Computer Architecture

Computer Architecture

Gérard Blanchet
Bertrand Dupouy

iSTE
WILEY

First published 2013 in Great Britain and the United States by ISTE Ltd and John Wiley & Sons, Inc.

ISTE Ltd
27-37 St George's Road
London SW19 4EU
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

Library of Congress Control Number: 2012951898

British Library Cataloguing-in-Publication Data
A CIP record for this book is available from the British Library
ISBN: 978-1-84821-429-3

FSC
www.fsc.org
MIX
Paper from
responsible sources
FSC® C013604

Printed and bound in Great Britain by CPI Group (UK) Ltd., Croydon, Surrey CR0 4YY

Table of Contents

Preface

This book presents the concepts necessary for understanding the operation of a computer. The book is written based on the following:

– the details of how a computer's components function electronically are beyond the scope of this book;

– the emphasis is on the concepts and the book focuses on the building blocks of a machine's architecture, on their functions, and on their interaction;

– the essential links between software and hardware resource are emphasized wherever necessary.

For reasons of clarity, we have deliberately chosen examples that apply to machines from all eras, without having to water down the contents of the book. This choice helps us to show how techniques, concepts and performance have evolved since the first computers.

This book is divided into five parts. The first four, which are of increasing difficulty, form the core of the book: "Elements of a basic architecture", "Programming model and operation", "Memory hierarchy" and "Parallelism and performance enhancement". The final part, which comprises appendices, provides hints and solutions to the exercises in the book as well as programming models. The reader may approach each part independently based on their prior knowledge and goals.

Presentation of the five parts:

1) Elements of a basic architecture:

– Chapter 1 takes a historical approach to present the main building blocks of a processor.

– Chapter 2 lists in detail the basic modules and their features, and describes how they are connected.

– Chapter 3 focuses on the representation of information: integers, floating-point numbers, fixed-point numbers and characters.

2) Programming model and operation:

– Chapter 4 explains the relationship between the set of instructions and the architecture.

– Chapter 5 provides a detailed example of the execution of an instruction to shed some light on the internal mechanisms that govern the operation of a processor. Some additional elements, such as coprocessors and vector extensions, are also introduced.

– Chapter 6 describes the rules – polling, direct memory accesses and interrupts – involved in exchanges with peripherals.

3) Memory hierarchy:

– Chapter 7 gives some elements – hierarchy, segmentation and paging – on the organization of memory.

– Chapter 8 presents cache memory organization and access methods.

– Chapter 9 describes virtual memory management concepts, rules and access rights.

4) Parallelism and performance enhancement:

– Chapter 10 gives an introduction to parallelism by presenting pipeline architectures: concepts, as well as software and hardware conflict resolution.

– Chapter 11 gives the DLX architecture as an example.

– Chapter 12 deals with cache management in a multiprocessor environment; coherence and protocols (MSI, MEI, etc.).

– Chapter 13 presents the operation of a superscalar architecture conflict, the scoreboarding and Tomasulo algorithms, and VLIW architectures.

5) Complementary material on the programming models used and the hints and solutions to the exercises given in the different chapters can be found in the appendices.

PART 1

Elements of a Basic Architecture

Chapter 1

Introduction

After providing some historical background, we will highlight the major components of a *computer* machine [MOR 81, ROS 69, LAV 75]. This will lead us to describe a category of calculators that we will refer to as *classic architecture machines*, or *classic architecture uniprocessors*. We will examine the functions performed by each of their modules, and then describe them in greater detail in the following chapters.

1.1. Historical background

1.1.1. *Automations and mechanical calculators*

The first known mechanical calculators [SCI 96] were designed by Wilhelm Schickard (1592–1635) (≈1623), Blaise Pascal (≈1642) and Gottfried Wilhelm Leibniz (1646–1716) (≈1673): they operate in base 10 through a gear mechanism.

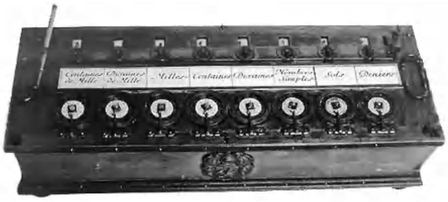


Figure 1.1. *Blaise Pascal's Pascaline*

It is up to the user to put together series of operations. The need for a sequence of processes that is automated is what will eventually lead to the design of computers.

The sequencing of simple tasks had already been implemented in the design of music boxes, barrel organs, self-playing pianos, in which cylinders with pins, cam systems and perforated paper tapes determined the melody. The loom, designed by Joseph-Marie Jacquard (1752–1834), is another example of an automaton. A series of perforated cards indicates the sequence of elementary operations to perform: each hole allows a needle to go through, and the tetrahedron that supports the cards rotates at the same pace as the shuttle which carries the thread that is woven. Introduced in the years 1804–1805, Jacquard's invention was formally recognized by France as being of a public benefit in 1806. In 1812, there were 11,000 such looms in France [ENC 08]. Some can still be found in operation in workshops around Lyon.

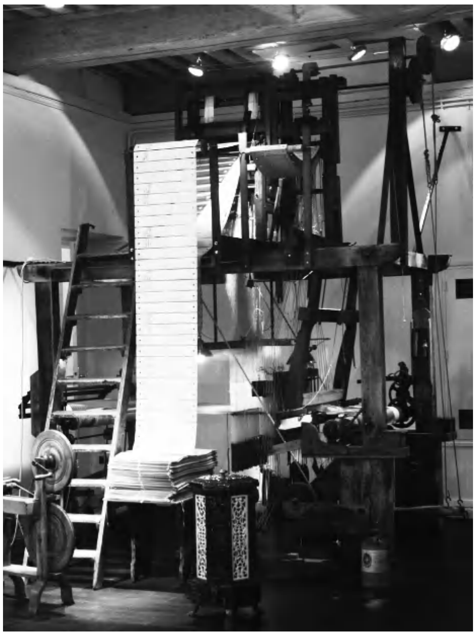


Figure 1.2. *An example of Jacquard's loom, courtesy of "La Maison des Canuts", Lyon, France*

This system provides a first glance at what will later become devices based on programmable automatons, or calculators, dedicated to controlling industrial processes.

Charles Babbage (1792–1871) was the first to undertake the design of a machine combining an automaton and a mechanical calculator. Having already designed a calculator, the Difference Engine, which can be seen at the Science Museum in London, he presented a project for a more universal machine, at a seminar held in Turin in 1841. His collaboration with Ada Lovelace (the daughter of Lord Byron) allowed him to describe a more detailed and ambitious machine, which foreshadows our modern computers. This machine, known as the *analytical engine* [MEN 42], autonomously performs sequences of arithmetic operations. As with Jacquard's loom, it is controlled by perforated tape. The user describes on this "program-tape" the sequence of operations that needs to be performed by the machine. The tape is fed into the machine upon each new execution. This is because Babbage's machine, despite its ability to memorize intermediate results, had no means for memorizing programs, which were always on some external support. This is known as an *external program* machine. This machine introduces the concept of *memory* (referred to by Babbage as the *store*) and of a *processor* (the *mill*). Another innovation, and contrary to what was done before, is that the needles, which engaged based on the presence or the absence of holes in the perforated tape, do not directly engage the *output devices*. In a barrel organ, a note is associated with each hole in the tape; this is formally described by saying that the output is *logically equal* to the input. In the analytical engine, however, we can already say that a program and data are coded.

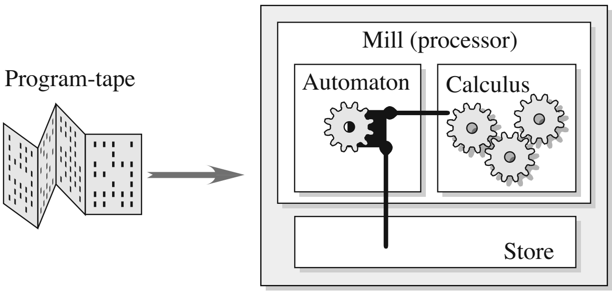


Figure 1.3. *Babbage's analytical engine*

This machine is divided into three distinct components, with different functions: the *automaton–calculator* part, the *data* and the *program*.

While each *row* of the perforated tape contains data that are "logical" in nature – the presence or the absence of a hole – the same cannot be said for both the automaton, which is purely mechanical, and the calculation unit, which operates on base 10 representations.

1.1.1.1. *Data storage*

The idea that it was necessary to automatically process data took hold during the 1890 census in the United States, a census that covered 62 million people. It was the subject of a call for bids, with the contract going to Herman Hollerith (1860–1929). Hollerith suggested using a system of perforated cards already used by certain railway companies. The cards were 7.375 by 3.25 inches which, as the legend goes, correspond to the size of the $1 bill at the time. The *Tabulating Machine Company*, started by Herman Hollerith, would eventually become *International Business Machines* (IBM), in 1924.

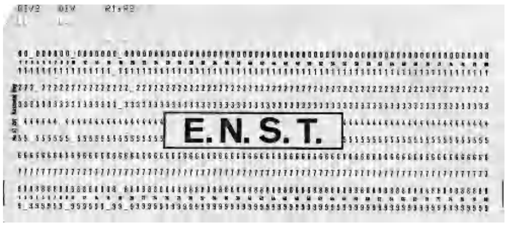


Figure 1.4. *A perforated card: each character is coded according to the "Hollerith" code*

In 1937, Howard Aiken, of Harvard University, gave IBM the suggestion of building a giant calculator from the mechanical and electromechanical devices used for punch card machines. Completed in 1943, the machine weighed 10,000 pounds, was equipped with accumulators capable of memorizing 72 numbers, and could multiply two 23-digit numbers in 6 s. It was controlled through instructions coded onto perforated paper tape.

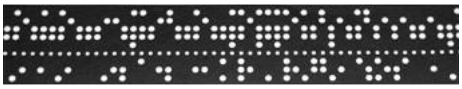


Figure 1.5. *Perforated tape*

Despite the knowledge acquired from Babbage, this machine lacked the ability to process conditional instructions. It did, however, have two additional features compared to Babbage's analytical engine: a clock for controlling sequences of operations and registers, a type of temporary memory used for recording data.

Another precursor was the *Robinson*, designed in England during World War II and used for decoding encrypted messages created by the German forces on *Enigma* machines.

1.1.2. *From external program to stored program*

In the 1940s, research into automated calculators was a booming field, spurred in large part by A. Turing in England; H. Aiken, P. Eckert and J. Mauchly [MAU 79] in the United States; and based in part on the works of J.V.Atanasoff (1995[†]) (*Automatic Electronic Digital Computer* (AEDQ) between 1937 and 1942).

The first machines that were built were electromechanical, and later relied on vacuum tube technology. They were designed for specific processes and had to be rewired every time a change was required in the sequence of operations. These were still externally programmed machines. J. von Neumann [VON 45, GOL 63] built the foundations for the architecture used by modern calculators, the *von Neumann* architecture.

The first two principles that define this architecture are the following:

– The universal applicability of the machines.

– Just as intermediate results produced from the execution of operations are stored into memory, the operations themselves will be stored in memory. This is called *stored-program* computing.

> The elementary operations will be specified by *instructions*, the instructions are listed in *programs* and the programs are stored in *memory*. The machine can now go through the steps in a program with no outside intervention, and without having to reload the program every time it has to be executed.

The third principle that makes this calculator an “intelligent” machine, as opposed to its ancestors, is the *sequence break*. The machine has decision capabilities that are independent from any human intervention: as the program proceeds through its different steps, the automaton decides the sequence of instructions to be executed, based on the results of tests performed on the data being processed. Subsequent machines rely on this basic organization.

Computer designers then focused their efforts in two directions:

– *Technology*: using components that are more compact, perform better, with more complex functions, and consume lower energy.

– *Architecture*: parallelization of the processor’s activities and organization of the memory according to a hierarchy. Machines designed with a *Harvard* architecture, in which access to instructions and to data is performed independently, meet this condition in part.

Figure 1.6 presents the major dates and concepts in the evolution that led to what is now called a *computer*. Note that without the methodological foundation provided by *Boolean algebra*, the first computer would probably not have emerged so quickly. This is because the use of this algebra leads to a *unification* of the representations used for designing the components and coding the instructions and data.

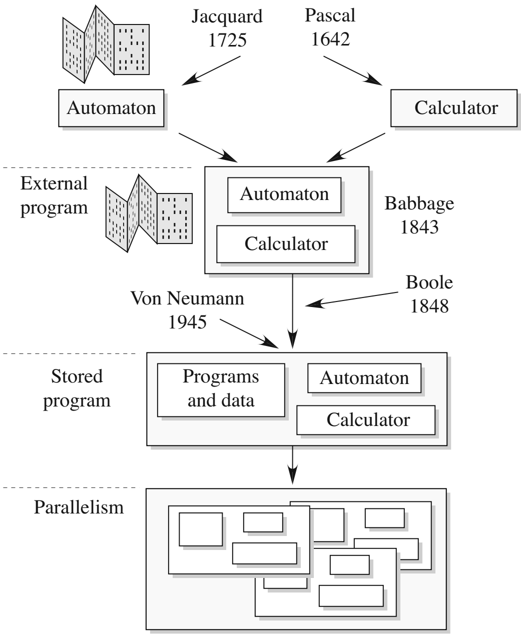


Figure 1.6. *From externally programmed to parallel computing*

1.1.3. *The different generations*

Since the *Electronic Discrete Variable Automatic Computer* (EDVAC) in 1945, under the direction of J. von Neumann [VON 45, GOL 63] (the first stored-program calculator), hundreds of machines have been designed. To organize the history of these machines, we can use the concept of *generations* of calculators, which is based essentially on technological considerations. Another classification could just as well be made based on software criteria, associated with the development of languages and operating systems for calculators.

1.1.3.1. *The first generation (≈1938–1953)*

Machines of this era are closer to laboratory prototypes than computers as we picture them today. These machines consist of relays, electronic tubes, resistors and other discrete components. The ENIAC, for example, abbreviated form for *Electronic Numerical Integrator And Computer*, was made up of 18,000 vacuum tubes, consumed around 150 kW, and was equipped with 20 memory elements (Figure 1.7[1]).

[1] `http://ftp.arl.army.mil/ftp/historic-computers`

Figure 1.7. *A photograph of ENIAC*

Because of the difficulties in the calculation part of the work, the processes were executed in series by operators working on a single binary element.

Being very energy-intensive, bulky and unreliable, these machines had an extremely crude programming language, known as *machine language*. Program development represents a considerable amount of work. Only one copy of each of these machines was made, and they were essentially used for research purposes. This was the case with the ENIAC, for example, which was involved in the research program for developing the *Bomba* [LAU 83], a machine used for decrypting messages during World War II.

1.1.3.2. *Second generation (≈1953–1963)*

The second generation saw the advent of machines that were easier to operate (the IBM-701, among others). *Transistors* (the first of which dates back to 1947) started to replacing vacuum tubes. Memory used ferrite toroids, and *operating systems*, the first tools designed to facilitate the use of computers, were created. Until then, machines were not equipped with development environments or with a user interface as we know them now. Pre-programmed input–output modules, known as Input Output Control Systems (IOCS) are the only available tools to facilitate programming. Each task (editing, processing, etc.) is executed automatically. In order to save time between the end of a job and the beginning of another, the *batch processing* system is introduced, which groups together jobs of the same type. At the end of each task, the operating system takes control again, and launches the next job. Complex programming languages were created, and become known as *symbolic coding systems*. The first FORTRAN (*FORmula TRANslator*) compiler dates back to 1957 and is included with the IBM-704. The first specifications for COBOL (*COmmon Business Oriented Language*) were laid out in 1959 under the name

COBOL 60. Large size applications in the field of management are developed. Magnetic tapes are used for archiving data.

1.1.3.3. *Third generation (≈1964–1975)*

The PLANAR process, developed at FAIRCHILD starting in 1959, makes it possible to produce integrated circuits. This fabrication technique is a qualitative breakthrough: reliability, energy consumption and size being dramatically improved.

Alongside the advances in hardware performance came the concept of *multiprogramming*, the objective of which is to optimize the use of the machine. Several programs are stored in the memory at the same time, making it possible to quickly switch from one program to another. The concept of input–output device independence emerges. The programmer no longer has to explicitly specify the unit where the input–output operation is being executed. Operating systems are now written in high-level languages.

Several computer operating modes are created in addition to batch processing:

– *Time sharing*, TS, lets the user work interactively with the machine. The best known TS system, *Compatible Time Sharing System* (TSS), was developed at *Massachusetts Institute of Technology* (MIT) and led to the Multics system, developed collaboratively by MIT, *Bell Labs* and *General Electric*.

– *Real time* is used for industrial process control. Its defining feature is that the system must meet deadlines set by outside stimuli.

– *Transaction processing* is mainly used in management computing. The user communicates with the machine using a set of requests sent from a workstation.

The concept of *virtual memory* is developed. The joint use of drives and memory, which is seamless for the user, creates the impression of having a memory capacity far greater than what is physically available. The mid-1960s see the advent of the IBM-360 calculator series, designed for general use, and equipped with an operating system (OS/360) capable of managing several types of jobs (*batch processing*, *time sharing*, *multiprocessing*, etc.).

This new era sets the stage for a spectacular increase in the complexity of operating systems. Along with this series of calculators emerges the concept of compatibility between machines. This means that users can acquire a more powerful machine within the series offered by the manufacturer, and still hold on to their initial software investment.

The first multiprocessor systems (computers equipped with several "automaton–calculation" units) are born at the end of the 1960s. The development of systems for machines to communicate with one another leads to computer *networks*.

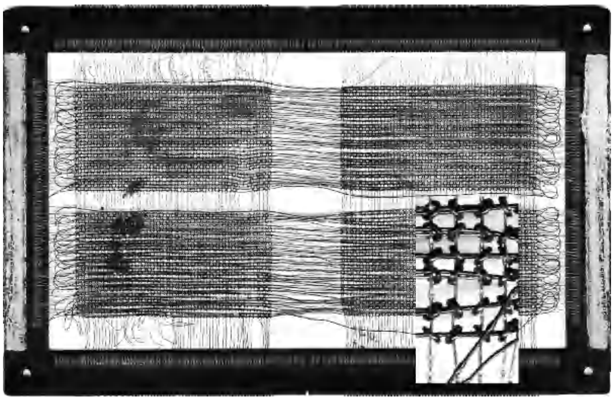


Figure 1.8. *In the 1970s, memory still relied on magnetic cores. This photograph shows a* $4 \times (32 \times 64)$ *bit plane. Each toroid,* $\approx$*0.6 mm in diameter, has three wires going through its center*

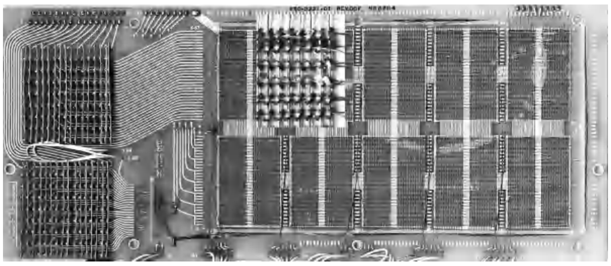


Figure 1.9. *The memory plane photographed here comprises twenty 512-bit planes. The toroids have become difficult to discern with the naked eye*

In the early 1970s, the manufacturing company IBM adopted a new policy (*unbundling*) regarding the distribution of its products, where hardware and software are separated. It then becomes possible to obtain IBM-compatible hardware and software developed by companies in the service industry. This policy led to the rise of a powerful software industry that was independent of machine manufacturers.

1.1.3.4. *Fourth generation (≈1975–)*

This fourth generation is tied to the systematic use of circuits with large, and later very large, scale integration (LLSI and VLSI). This is not due to any particular technological breakthrough, but rather due to the dramatic improvement in fabrication processes and circuit design, which are now computer assisted.

The integration of the different processor modules culminated in the early 1970s, with the development of the microprocessor. Intel® releases the I4004. The processor takes up only a few square millimeters of silicon surface. The circuit is called a *chip*. The first microcomputer, built around the Intel® I8080 microprocessor, came into existence in 1971.

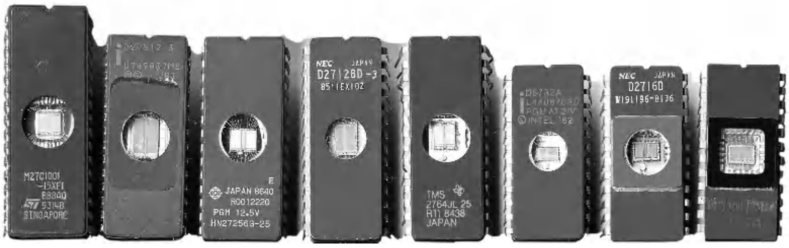


Figure 1.10. *A few reprogrammable memory circuits: from 8 kbits (≈1977) (right-hand chip) to 1 Mbit (≈1997) (left-hand chip) with no significant change in silicon surface area*

Figure 1.11. *A few old microprocessors: (a) Motorola*$^{\text{TM}}$ *6800 (1974, ≈6,800 transitors), Intel*$^{\text{TM}}$ *I8088 (1979, ≈29,000), Zilog*$^{\text{TM}}$ *Z80 (1976, ≈8,500), AMD Athlon 64X2 (1999, from ≈122 millions to ≈243 millions); (b) Intel i486 DX2 (1989, ≈1.2 million), Texas Instruments*$^{\text{TM}}$ *TMX320C40 (1991, ≈650,000)*

The increase in the scale of integration makes it possible for anybody to have access to machines with capabilities equivalent to the massive machines from the early 1970s. At the same time, the field of software development is exploding.

Designers rely more and more on parallelism in their machine architecture in order to improve performance without having to implement new technologies (*pipeline*, *vectorization*, *caches*, etc.). New architectures are developed: *language machines*, *multiprocessor* machines, and *data flow* machines.

Operating systems feature network communication abilities, access to databases and distributed computing. At the same time, and under pressure from microcomputer users, the idea that systems should be *user friendly* begins to take hold. The ease of use and a pleasant feel become decisive factors in the choice of software.

The concept of the virtual machine is widespread. The user no longer needs to know the details of how a machine operates. They are addressing a virtual machine, supported by an operating system hosting other operating systems.

The "digital" world keeps growing, taking over every sector, from the most technical–*instrumentation*, *process command*, etc.–to the most mundane–*electronic payments*, *home automation*, etc.

1.2. Introduction to internal operation

1.2.1. *Communicating with the machine*

The three internal functioning units – the automaton, the calculation unit and the memory unit that contain the intermediate results and the program – appear as a single module accessible to the user only through the means of communication called *peripheral units*, or *peripherals*.

The data available as machine inputs (or outputs) are only rarely represented in binary. It can exist in many different formats: as text, an image, speech, etc. Between these sources of data and the three functional units, the following must be present:

– sensors providing an electrical image of the source;

– preprocessing hardware that, based on this image, provides a signal usable by the computer by meeting the electrical specifications of the connection (e.g. a filter, followed by a sampling of the source signal, itself followed by a link in series with the computer);

– *exchange units* located between the hardware and the computer's core.

Exchange units are a part of the computing machine. The user ignores their existence.

> We will adopt the convention of referring to the system consisting of the processor (calculator and automaton) + memory + exchange units as the *Central Processing Unit* (CPU).

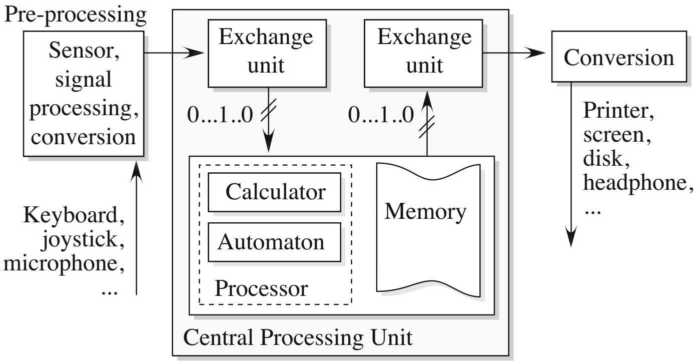


Figure 1.12. *User–machine communication*

It is important to note that the symbols “0” and “1” used in Figure 1.12 to represent data are notations used by convention. This facilitates the representation of *logic* values provided by the computing machine. They could have been defined as “α” and “β”, “ϕ” and “E”, etc.

What emerges from this is a modular structure, the elements of which are the *processor* (calculation and automaton part), the *memory*, the *exchange units*, and connections, or *buses*, the purpose of which is to connect all of these modules together.

1.2.2. *Carrying out the instructions*

The functional units in charge of carrying out a program are the *automaton* and the *calculator*:

– the automaton, or *control unit*, is in command of all the operations;

– the module tasked with the calculation part will be referred as the *processing unit*.

Together, these two modules make up the *processor*, the “intelligent” part of the machine.

The basic operations performed by the computer are known as *instructions*. A set of instructions used for achieving a task will be referred to as a *program* (Figure 1.13).

> Every action carried out by the computing machine corresponds to the execution of a program.

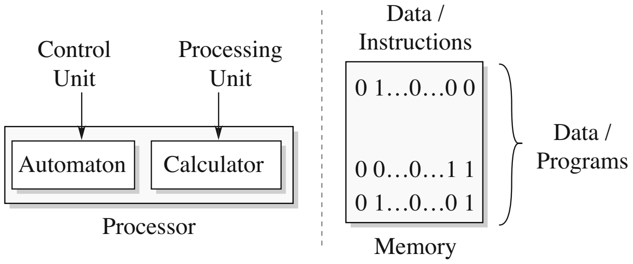


Figure 1.13. *Processor and memory*

Once it has been turned on, the computer executes a "fetch-execution" cycle, which can only be interrupted by cutting its power supply.

The fetch operation consists of retrieving within the memory an instruction that the control unit recognizes – *decodes* – and which will be executed by the processing unit. The execution leads to (see Figure 1.14) (1) a local processing operation, (2) something being read from or written into memory, or (3) something being read from or written into an exchange unit. The control unit generates all of the signals involved in going through the cycle.

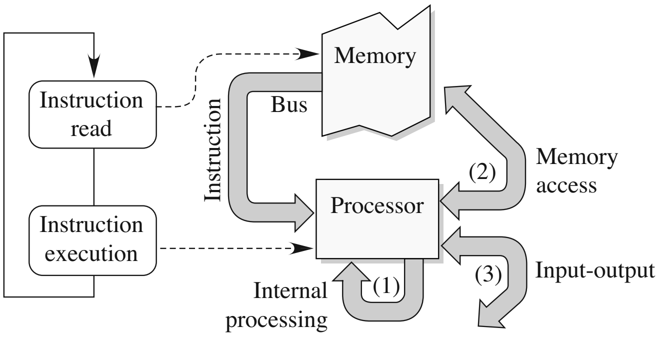


Figure 1.14. *Accessing an instruction*

1.3. Future prospects

Silicon will remain the material of choice of integrated circuit founders for many years to come. CMOS technology, the abbreviated form for *complementary metal-oxide semiconductor* (and its derivatives), long ago replaced TTL (*transistor-transistor Logic*) and ECL (*emitter-coupled logic*) technologies, even inside *mainframes*, because of its low consumption and its performance capabilities.

The power supply voltage keeps dropping – 3.3, 2.9, 1.8 V are now common – while the scale of integration increases with improvements in etching techniques (half-pitch below 30 nm) and the use of copper for metallization. There are many improvements in fabrication processes. *Computer-aided design* (CAD), helps reduce development time and increases circuit complexity. The integration of test methods as early as during the design phase is an advantage for improving fabrication yields.

– It is the architecture that is expected to greatly enhance the machine performance. Parallelism is the main way to achieve this. It can be implemented at the level of the processor itself (operator parallelism and data parallelism) or on the higher level of the machine, by installing several processors which may or may not be able to cooperate. The improvement of the communication links becomes significant to transmit data. Protocols and bus technology are rapidly evolving to meet this objective.

– User-friendly interfaces are now commonplace. Multi-task operating systems are used on personal machines. Operating systems now have the ability to use the machine's hardware resources.

– The cost of a machine also includes the cost of the software designed for operating it. Despite the emergence of improved software technology (*object-oriented* approach, etc.), applications are developed and maintained with lifecycles far beyond the replacement cycle of hardware.

– As early on as the design phase, machines are equipped with the means of communication which facilitate their integration into *networks*. Working on a machine does not imply that the processing is performed locally. Similarly, the data handled can originate from remote sites. While this concept of delocalization is not new, the concept that access should be transparent is quite recent (*distributed* computing).

Throughout this book, the word *architecture* will refer to the organization of the modules which comprise the computing machine, to their interconnections, and not to how they are actually created using logic components: gates, flip-flops, registers or other, more complex elements.

Chapter 2

The Basic Modules

This chapter presents the main modules that constitute a computer: the memory, the processor and the exchange units. We will review the operation of registers and counters used in the execution of instructions. We will not discuss in detail the internal workings of the different modules, and will instead focus on their functions.

2.1. Memory

Memory contains all of the data used by the processor. There are two types of data: a series of instructions and data on which to execute these instructions.

> The information stored in a memory has no inherent significance. It only becomes meaningful once it is used. If it is handled by a program, it consists of *data*, and if it is read by the control unit to be executed, it consists of *instructions*.

For example, if we “double-click” on the icon of a word processor, then this program, stored on a drive, is transferred to memory using a program called a *loader*, which sees it as “data”, until it becomes a “program” to the user.

2.1.1. *Definitions*

Memory is a juxtaposition of *cells* called *memory words*, made up of m electronic elements that can exist in two stable states (the *flip-flop* logic function) referred to as

"0" and "1". Each cell can therefore code 2^m different elements of information. Each of the m elements will be called a *bit*, abbreviated form for *binary digit*. If a memory word contains m bits, these are usually numbered from 0 to $m - 1$, from the *least significant bit* (LSB), to the *most significant bit* (MSB).

Each cell is uniquely identified by a number – its *address*. The information in the memory is accessed by setting the n lines of the address buses. The binary configuration of these n lines codes the address of the word, whose m bits of content can then be read by the data bus. This is achieved using a $n \to 2^n$ decoder located between the address bus and the actual memory, so that the values of the addresses scale from 0 to $2^n - 1$.

In Figure 2.1, the configuration $\{00 \ldots 0010\}$ leads to the selection of the memory word with the address 2, the content of which becomes accessible to the data bus.

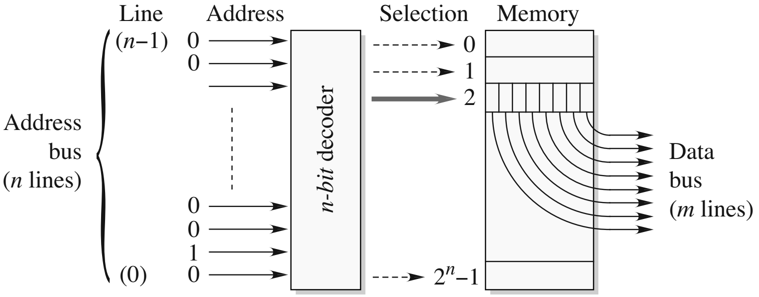


Figure 2.1. *Memory*

The delay between the setting of the address and the time when the data becomes available on the data bus is called the *access time*. The information stored in a cell is called the *content* of the address:

– a memory cell is denoted by M, where M is the cell's address;

– the content of the memory word with address M is denoted by $[M]$.

The name used for these memory words depends on their length. While a *byte* always refers to an eight-bit cell, the same is not true for a *word*, which can consist of 16 bits, 32 bits, etc. A 32-bit word is sometimes called a *quadlet*, and a half-byte is a *nibble* or *nybble*.

When each byte has an address, the memory is said to be *byte addressable* and each of the bytes in a 32-bit memory word can be accessed individually. The addresses of words are multiples of four.

The unit of *memory size* is generally a byte, in which case it is expressed in *kilobytes* (kB), *megabytes* (MB) or *gigabytes* (GB), with:

1 kilobyte = 2^{10} bytes = 1,024 bytes,
1 megabyte = 2^{20} bytes = 1,048,576 bytes,
1 gigabyte = 2^{30} bytes = 1,073,741,824 bytes.

2.1.2. *A few elements of technology*

2.1.2.1. *Concept*

Memory design relies on any of the following three technologies:

– Each basic element consists of a flip-flop: this technique is implemented to produce what is known as *volatile* and *static* memories: volatile because once the power supply is shut down, the information is lost, and static because the information is stable as long as the power supply is maintained.

– The basic element is comparable to a capacitor, which can be either charged or not charged: this concept applies to *volatile* and *dynamic* memories (Figure 2.2). The term dynamic means that the information is lost whenever the capacitor discharges.

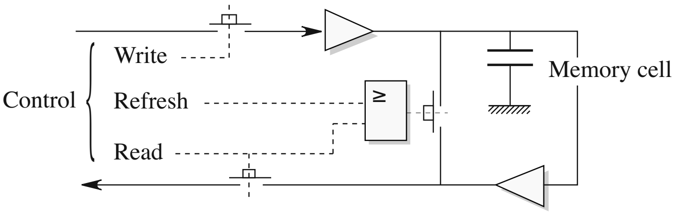


Figure 2.2. *Concept of the dynamic memory cell: the content of the cell must periodically be refreshed, something which is also achieved when the element is read*

This type of memory must periodically be *refreshed*, a task which can be accomplished using a read or write cycle.

– Producing the cell requires a phenomenon similar to the “breakdown” of a diode: this is how the so-called *non-volatile* memory is made. Once the content has been *programmed*, it exists permanently regardless of whether the power is on or off.

Volatile memories are usually known as RAM (*Random Access Memory*) and non-volatile memories as ROM (*Read-Only Memory*). Depending on the techniques, other terms such as DRAM (*Dynamic RAM*), PROM (*Programmable ROM*), F-PROM (*Field-Programmable ROM*), REPROM (*REProgrammable ROM*) and EAROM (*Electrically Alterable ROM*), are sometimes used.

2.1.2.2. *Characteristics*

The following are the main characteristics of memory:

– The access time, i.e. the time it takes from the moment an address is applied to the moment when we can be certain that the retrieved data is *valid*. This time is measured in nanoseconds (1 nanosecond $= 10^{-9}$ s), generally in the range of 1 to 100 ns. This parameter is not sufficient for measuring the machine's "memory performance". With modern computers, a memory access can sometimes be performed according to a complex protocol, and it is difficult to determine a value that is independent of the machine's architecture.

– The *size*, expressed in thousands, millions, or billions of bytes. Currently, the most common sizes are above 4 GB for personal computers, tens of GB for servers, and greater still for large machines (*supercomputers*).

– The *technology* which falls into two categories:

- rapid memories that rely on *transistor-transistor logic* (TTL) technology;

- *Complementary MOS* (CMOS) technologies, which have very low energy consumption and allow for high-scale integration (personal computers on the market today have memory components equipped with several billion bytes).

– The *internal organization* which allows more or less rapid access to information. Examples of available memory include VRAM (*Video RAM*), EDO (*Extended Data Out*), *synchronous RAM*, etc.

2.2. The processor

2.2.1. *Functional units*

The processor houses two functional units: the *control unit* and the *processing unit* (see Figure 2.3).

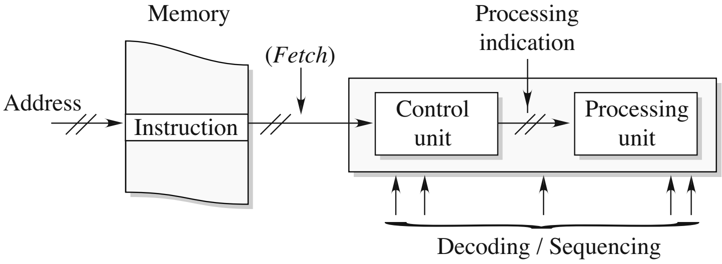


Figure 2.3. *Execution of an instruction*

2.2.1.1. *The control unit*

The control unit has three functions (see Figure 2.3):

– It *fetches* instruction stored in the memory: it sets the address of the instruction on the address bus, then, after a delay long enough to ensure that the address is in fact stable on the data bus (or the instruction bus, for machines equipped with distinct buses), it loads the instruction it has just obtained into a register.

– It *decodes* the instruction.

– It indicates to the processing unit which arithmetic and logic processes need to be performed, and generates all of the signals necessary to the execution of the instruction. This is the *execution* step.

2.2.1.2. *The processing unit*

The processing unit ensures the execution of elementary operations "specified" by the control unit ("processing indications"). The information handled and the intermediate results are stored in memorization elements internal to the processor, known as *registers*.

No operation is made directly with the memory cells: they are instead copied into registers, which may or may not be accessible to the programmer, before being processed.

The control unit, the processing unit and the registers are connected to each other, which makes it possible to load or read in parallel all of their bits. These *communication lines* are known as *internal buses* (Figure 2.4).

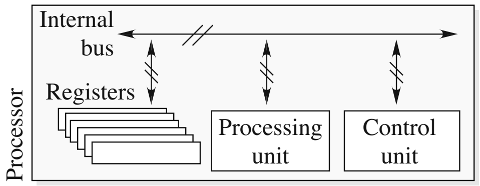


Figure 2.4. *Internal bus*

2.2.2. *Processor registers*

2.2.2.1. *Working register*

Working registers are used to store results without accessing the memory. This simplifies the handling of data (there is no need for managing addresses) and the time it takes to access the information is shorter. In most classic architecture processors, the number of registers is in the order of 8 to 32.

2.2.2.2. *Accumulator register*

In many classic architecture machines, the set of instructions favors certain registers known as *accumulator registers* or simply *accumulators*. In Intel® x86 series microprocessors, for example, all of the instructions for transfers between memory and registers, or between registers, can affect the accumulator register, denoted by AX. The same is not true for registers indexed SI or DI, which offer more limited possibilities for handling information.

2.2.2.3. *Stack pointer*

Stack pointers are mainly used in the mechanism for calling functions or subroutines. The stack is a part of the memory which is managed (Figure 2.5) according to the content of the *stack pointer* register.

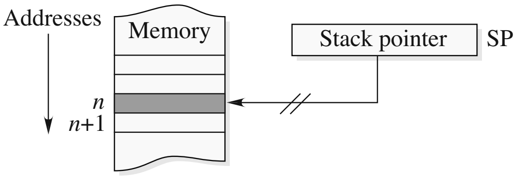


Figure 2.5. *The stack*

The stack pointer provides the address of a memory word called the *top of the stack*. The top of the stack usually has the same address as the word in the stack with the smallest address n.

The stack pointer and the stack are also handled by special instructions or during autosaves of the processor state (calls and returns from subroutines, interrupts, etc.).

EXAMPLE 2.1.– Let us assume that in an initial situation (Figure 2.6), a stack pointer contains the value $1{,}000_{16}$, which is the address of the memory word at the top of the stack. The stacking is done on a single memory word. The stack pointer is *decremented* and the information is stored at the top of the stack, in this case at the address 0FFF_{16} (Figure 2.6).

In certain machines, the stack is created using rapid memory distinct from the main memory (the memory containing data and programs). This is known as a *hardware stack*.

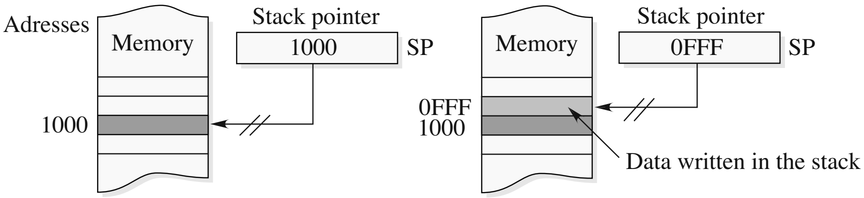


Figure 2.6. *Writing in the stack*

2.2.2.4. *Flag register*

The *flag register*, or *condition code register*, plays a role that becomes clear when performing a sum with a carry. When summing two integers coded in several memory words, the operation must be performed over several summing instructions, each instruction involving only a word. While there is no need to consider a carry for the first sum, since it involves the least significant bits of the operands, the same cannot be said of the following sums. The carry resulting from the execution of a sum is stored in the flag register. Its role is to memorize the information related to the execution of certain instructions, such as arithmetic instructions and logic instructions.

REMARKS 2.1.–

– Flags are used by conditional branching instructions, among others.

– Not all instructions lead to flag modifications. In Intel® x86 series microprocessors, assignment instructions (transfers from register to register, or between memory and a register) do not affect flags.

– The minimum set of flags is as follows:

- the *carry* flag C or CY;

- the *zero* flag Z, which indicates that an arithmetic or logic operation has produced a result equal to zero, or that a comparison instruction has produced an equality;

- the *parity flag* P, which indicates the parity of the number of 1 bit in the result of an operation;

- the *sign* flag, or *negative* flag S, which is nothing but a copy of the most significant bit in the result of the instruction that has just been executed;

- the *overflow* flag V, an indicator which can be used after executing operations on integers coded in two's complement (see Remark 2.2).

REMARK 2.2.– If we perform sums on numbers using the two's complement binary representation, it is important to avoid any error due to a carry which would propagate to the sign bit and change it to an erroneous value. Consider, for example, numbers coded in four bits in two's complement. Let C and S be the carry and sign flags, respectively, and let c be the carry generated by the sum of the three least significant bits (Figure 2.7).

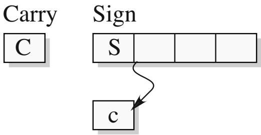


Figure 2.7. *Sign and carry*

Let us examine each possible case in turn, starting with the case of two positive numbers A and B.

1) Let the values of the operands be 3 and 5:

$$\begin{array}{ll} A & = 0\,0\,1\,1 \rightarrow +3 \\ B & = 0\,1\,0\,1 \rightarrow +5 \\ A+B & = 1\,0\,0\,0 \rightarrow -8 \end{array}$$

The result obtained is wrong. Note that c takes on the value 1, while C and S take on the values 0 and 1, respectively. In the case where we get the right result, for example with $A = 2$ and $B = 3$, then c, C and S take on the value 0.

2) In the case of two negative numbers:

$$\begin{array}{lr} A = & 1\,0\,1\,0 \rightarrow -6 \\ B = & 1\,0\,1\,1 \rightarrow -5 \\ A+B = & 1\,0\,1\,0\,1 \rightarrow +5 \end{array}$$

For these values of A and B that lead to an erroneous result, note that c, C and S take on the values 0, 1, and 0, respectively. When we get the right result, the values of c, C and S become 1, 1 and 1, respectively.

3) For numbers of opposite signs, we find that no error occurs:

$$\begin{array}{lr} A = & 0\,1\,1\,0 \rightarrow +6 \\ B = & 1\,0\,1\,1 \rightarrow -5 \\ A+B = & 1\,0\,0\,0\,1 \rightarrow +1 \end{array}$$

We can sum up the results of all possible cases by drawing up a Karnaugh map where overflows are represented by the number 1. The flag indicating these cases is denoted by OV:

	00	01	11	10	CS
0	0	0	ϕ	1	
1	ϕ	1	0	0	
c					OV

The overflow bit OV is then equal to $\mathrm{C} \otimes \mathrm{c} = \overline{\mathrm{C}}\,\mathrm{c} + \overline{\mathrm{c}}\,\mathrm{C}$.

REMARK 2.3.– There is a flag, known as the *half carry* flag, which is usually not directly useable by the programmer. It is used for the propagation of the carry when working with BCD coded numbers (see Chapter 3). Consider the sum of the two numbers 18_{10} and 35_{10} coded in BCD (each decimal figure is converted to four-digit binary):

Base 10	BCD code
18	0001 1000
+35	0011 0101
= 53	0101 0011

Since the adder works in two's complement, it produces the result 0100 1101. The intermediate carry, or *half carry*, indicates that the result of summing the least significant digits is >9. It is used internally when it is preferable to have a result also coded in BCD using *decimal adjust* instructions. In the Intel® x86 processor, for example, the sum of the two previous 8 bits, BCD coded integers can be performed as follows (`al` and `bh` are 8-bit registers):

```
               ; al contains 18 (BCD) i.e. 0001 1000
               ; bh contains 35 (BCD) i.e. 0011 0101
add al,bh      ; al + bh --> al : 18 + 35 --> 4D
daa            ; decimal adjust of al: 4D --> 53
```

2.2.2.5. *Address registers*

Registers can sometimes be involved in memory addresses. The relevant registers can be either working registers or specialized registers, with the set of instructions defining the role of each resister. Some examples of these registers are:

– *indirection* registers, the content of which can be considered addresses;

– *index* registers, which act as an offset in the address calculation, specifically a value added to the address;

– *base* registers, which provide an address in the calculation of an address by indexing (the address of the operand is obtained by summing the contents of the base and index registers);

– *segment* registers (see Chapter 7), *limit* registers, etc.

2.2.3. *The elements of the processing unit*

The *processing unit* is the component tasked with executing arithmetic and logic operations, under the command of the control unit. The core of the processing unit is the *arithmetic and logic unit* (ALU). The ALU consists of a combinational logic circuit which performs elementary operations on the content of the registers: sums, subtractions, OR, AND, XOR, complement, etc.

2.2.3.1. *Arithmetic operations*

The sum/subtraction operators are achieved on a cellular level using "bit-by-bit" adders. The problem with designing fast adders comes from the propagation speed of the carry throughout all of the adder stages. A solution ([GOS 80]) consists of "cutting up" the n–n adder into several p–p adder blocks, so that it becomes possible to propagate the carry from a p–p block to another p–p block. The carry is calculated on a global level for each p–p adder. This is known as a *look ahead carry*. Another technique consists, for this same division into p–p adders, of propagating the carry throughout the p-p floor only if it is necessary (*carry–skip adders*).

Multiplication can also be achieved in a combinational fashion. The only obstacle is the surface occupied by such a multiplier, the complexity of which is of the order of n^2 1–1 adders.

2.2.3.2. *The ALU and its flags*

The processing unit is often defined as the set consisting of the ALU and a certain number of registers that are inseparable from it (including flag registers and accumulator registers).

The command lines define the operation that needs to be performed, including, both as input and as output of the ALU, two lines for the *carry*:

– CY_{in}, which originates from the output of the carry flip-flop;

– CY_{out} as input for the same flip-flop.

Other lines, referred to as *flags*, provide the state of the result of the operation: *overflow*, *zero*, etc.

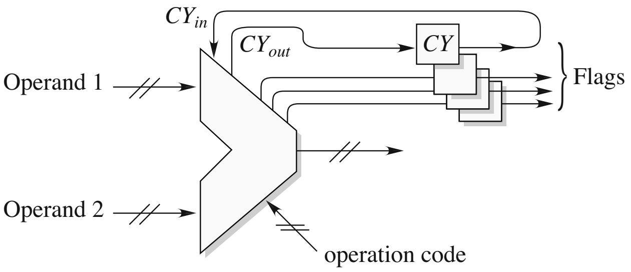


Figure 2.8. *The arithmetic logic unit*

REMARK 2.4.– Consider the case where a register is both the source and the destination of a sum. After executing the sum, the result should be located in the register. Consider the initial situation described in Figure 2.9.

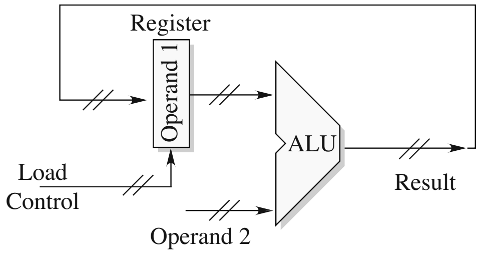


Figure 2.9. *Operational diagram*

Once the instruction is executed, the result is loaded in the register under the command of the control unit. However, in order for the operation to work properly, the information present at the register input cannot be modified when the register is loaded. This is not the case in the operational diagram shown in Figure 2.9. A simple method for avoiding this pitfall consists of inserting what is known as a *buffer* register between the register and the ALU (Figure 2.10).

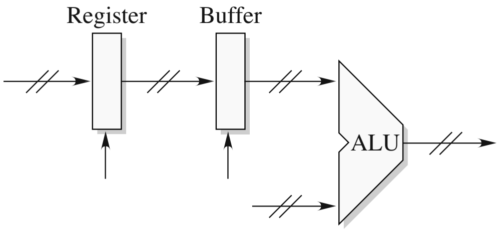


Figure 2.10. *The buffer register*

In the first step, the content of the register is transferred to the buffer, then, in the second step, the result is written into the register. We will encounter this scheme again in the example used for illustrating an instruction (Chapter 5). Later, we will consider the buffer–register system as a single register.

Any complex operation is performed using a combination of elementary operations available on the ALU and shift operations on registers. The multiplication operation, for example, will rely on sums and shifts if the specialized circuit used for its operation is not present in the ALU.

2.2.4. *The elements of the control unit*

The control unit sends control signals (also called *microcommands*) to ensure the proper execution of the decoded instructions.

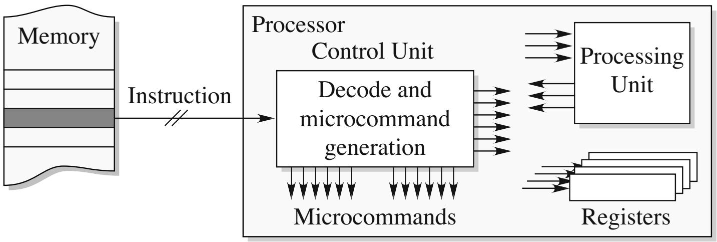


Figure 2.11. *Control unit*

The control unit consists of four modules:

– The *instruction register*, denoted by IR, receives the code for the instruction that the control unit has “fetched” from memory.

– The *program counter* (PC), or *instruction pointer* (IP), is a *programmable* counter containing the address of the instruction that needs to be executed. By “programmable”, we mean that it is possible to modify its content at any time without following the counting sequence. This is the case, for example, any time the sequentiality rule is violated (branches, subroutine calls, etc.). The instruction whose address is provided by the content of the program counter is placed in the instruction register, so that it can be decoded (Figure 2.12).

A 16-bit program counter enables access to 2^{16} program words. Given the Boolean nature of the information stored in memory (nothing looks more like an instruction than data), the content of the memory word is only an instruction if the program counter contains its address!

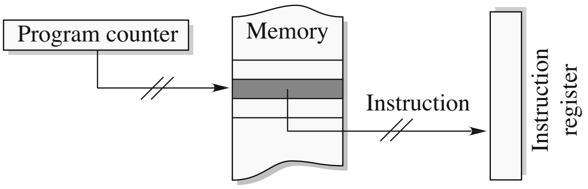


Figure 2.12. *Accessing an instruction*

– The *instruction decoder* is a circuit for recognizing the instruction contained in the instruction register. It indicates to the sequencer the sequence of basic instructions (*microcommand* sequence) that need to be performed to execute the instruction.

– The *sequencer* is the device that produces the sequence of commands, known as *microcommands*, used for loading, specifying the shift direction, or operations for the ALU, necessary for the execution of the instruction.

The sequencer's internal logic takes into account a certain number of events originating from the "outside", or generated by the exchange units or the memory: interrupts, holds, etc. The sequence of microcommands is delivered at a rate defined by the *internal clock* (Figure 2.13).

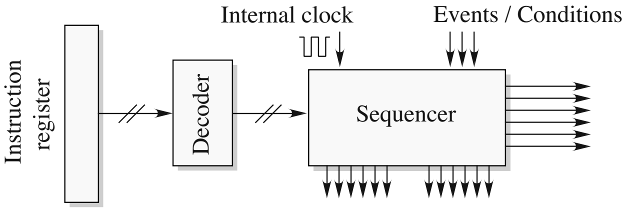


Figure 2.13. *Decoder/sequencer*

The speed of this clock determines the speed of the processor, and more generally, of the computer, if the elements surrounding the processor can keep up the pace. The phrase "1.2 GHz microcomputer" indicates that the basic clock frequency is 1.2 GHz. This specification is an indication of the performance of the processor, not of the machine.

2.2.5. *The address calculation unit*

Calculating addresses requires an arithmetic unit. The calculation can be performed by the ALU, as was the case in certain first generation microprocessors.

This method has the advantage of reducing the number of components in the central processing unit. On the other hand, the additional register operations and the resulting exchanges end up being detrimental to the machine's performance. The calculation of addresses is more often performed by an arithmetic unit specific to the task.

2.3. Communication between modules

The processor is physically connected to the memory and the exchange units by a set of lines. These connections form the so-called *external bus*, not to be confused with the *internal bus* connecting the ALU, the registers, etc. In the case of typical processors, the external bus consists of a data bus, an address bus, and a control bus.

The number of wires of the data bus gives us the number of bits that can be transferred to the memory in one access cycle. The transfer speed is called the bus *bandwidth*, usually given in bytes per second. This bandwidth depends to a large extent on the electrical characteristics, on the method used for performing transfers (synchronous and asynchronous modes), and on the bus exchange protocol.

PC-type microcomputers have a main bus (*Peripherals Component Interconnect*, PCI) for exchanges with the memory and modern peripheral cards. They are also equipped with dedicated buses, in particular the bus dedicated to transfers to the video memory (*Accelerated Graphics Port*, AGP).

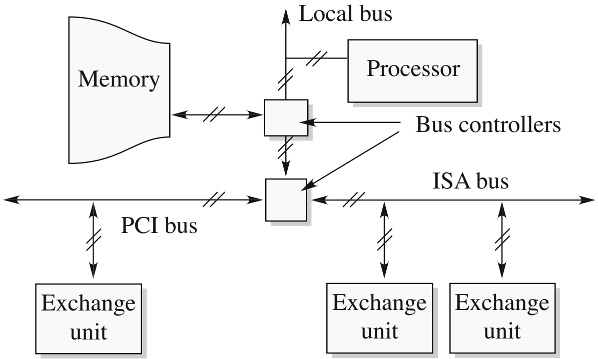


Figure 2.14. *Local data bus and external data bus: the connection between the local bus and the PCI bus depends on the generation of the machine, on the type of PCI bus used (PCI or PCI Express), and on the presence of northbridge and southbridge bus controllers*

So far, we have only discussed the transfer of information from the *data bus*. No such transfer can occur without first gaining access to this information. That is the role of the *address bus*. The processor provides this bus with the addresses of the elements it wants access to. The width, that is the number of lines, of the address bus determines the *addressable memory space*.

Other lines are necessary for the whole system to work properly, for example a read/write indicator, or lines indicating the state of the exchange units. The set of lines conveying this information is referred to as the *control bus*, or *command bus* (Figure 2.15).

The system comprising the data, address, and control buses constitutes what is called the external bus

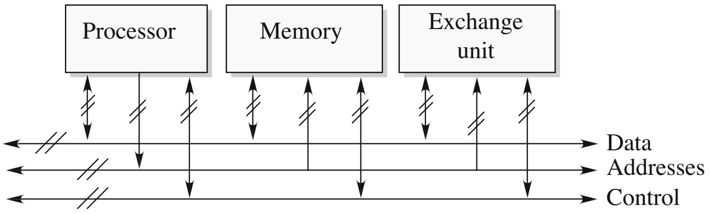


Figure 2.15. *Address, data and control buses*

In an effort to limit the number of communication lines (the sum of the bus widths, the number of electrical supply lines, and the clock signals), the solution that was chosen is to *multiplex* these bus lines, meaning that the same physical lines are used as both address and data lines.

On this external bus, it is possible to connect several processors or exchange units likely to take control of the buses – *multiprocessor* systems or *direct memory access* devices (section 6.3.2). When an exchange unit controls the buses, the other units have to be protected. This is done with the help of *three-state drivers*, which can exist in three states: 0, 1 and *high impedance*. This last state is equivalent to a physical disconnection of the line (Figure 2.16).

2.3.1. *The PCI bus*

The PCI bus is used in a very broad sense as an intermediate bus between the processor bus – the *local* bus – and the input–output buses [ABB 04]. It has been available in different versions – 32 or 64 bits, various information signaling voltages, supported bandwidths, etc. – since 1992, the year of its inception by Intel®.

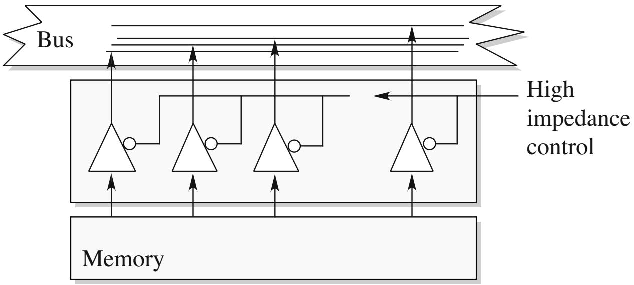


Figure 2.16. *Three-state drivers*

The PCI bus conveys data between the processor and the memory and any input–output devices by taking into account the presence of caches and the fact that they need to be updated (Chapter 8.) A bus is dedicated to the transfer of addresses and data (AD bus), whereas a control bus (C/BE# bus) indicates the phase of the bus. Figure 2.17 describes a read transfer between the processor and the memory. This diagram is very simplified, since not all of the signals involved in the communication protocol are shown.

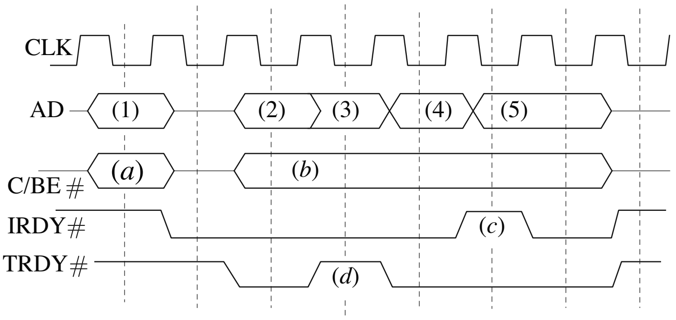


Figure 2.17. *Example of a read operation on a PCI bus: on the C/BE# bus, the memory read command is generated (a) and the address is sent (1) over the AD bus. IRDY# indicates that the processor (the initiator) is ready to receive the data sent by the memory (the target) in (2), (4), and (5) under the command of the C/BE# bus (b). Cycle (3) is a wait state signaled by the switch of TRDY# to 1 in (d). The switch of IRDY# to* 1 *in (c) indicates the end of the transfer*

A parity check system on the set of lines AD and C/BE# is implemented. The PAR line shows the parity "calculated" by the devices sending the information (address or data) and the SERR# and PERR# lines indicate a parity error provided by the target of the exchange and its source, respectively, if it happens to be necessary.

Each additional board is equipped with a software/hardware device (*firmware*), accessible in a particular addressing space (the *PCI configuration space*), which makes it possible to know if it is present when the machine starts, to know the type of resources (interrupts, memory and input–outputs) and to obtain a description of the memory space, or of the input–output space, which is available on the board.

Chapter 3

The Representation of Information

When representing data or communicating with the computer, the user relies on symbols – numbers, letters and punctuation – which together form *character strings*.

For example, the lines:

```
1   i = 15 ; r = 1.5 ;
2   car1 = "15" ; car2 = "Hello" ;
3   DIR > Result.doc
4   cat Result.tex
```

are interpreted as the instructions for a program (lines 1 and 2) or as commands sent to the operating system (lines 3 and 4). This *external representation* of data is not directly understandable to the machine. It requires a translation (*coding*) into a format that the computer can use. We will refer to the result of this operation as an *internal representation*. It involves in every case the use of logic symbols denoted by 0 and 1 (section 3.2.2). For reasons of clarity, and/or ease of specification, *logical* data are most often divided into sets of eight elements called *bytes*.

In this chapter, we will distinguish the following types: numbers, integer or real, and characters.

Consider again the previous example: internal coding of the data “15” will be different depending on whether the data are stored in the `i` variable or the `car1` variable. One is *numerical*, the other a *character string*. The external representation is the same for the two entities, despite the fact that they are fundamentally different in nature. The human operator is able differentiate them and can adapt their process according to context or using their knowledge of the object. Machines, on the other hand, require different internal representations for objects of different types. In this chapter, we present the rules that make it possible to convert *external format* → *internal format*, without making assumptions about the final implantation of the result in the machine's memory.

3.1. Review

3.1.1. *Base 2*

The *representation* of a number x in base b is defined as the vector $\{a_n, \ldots, a_0, a_{-1}, \ldots, a_{-p}\}$ such that:

$$x = \sum_{i=-p}^{n} a_i b^i \qquad [3.1]$$

where the a_i are integers between 0 and $(b-1)$. The index i and the number b^i are the *position* and the *weight*, respectively, of the symbol a_i. The coefficients a_n and a_{-p} are the *most significant* and *least significant* digits, respectively. In the case of base 2 (*binary coding*), the coefficients a_i may assume the values 0 or 1 and will be referred to as *bits*, shorten form for *binary digit*.

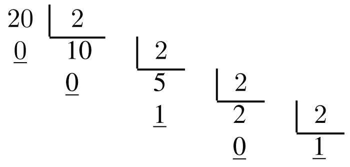
EXAMPLE 3.1.–

– The binary number $1010 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 = 10_{10}$.

– The change from base 10 to base 2 is done by a *series of divisions* by 2 for the integer part and a *series of multiplications* by 2 for the fractional part. Consider the number 20.375 in base 10.

We can switch to its base 2 representation by doing the following:

1) For the integer part, x_{INT}, we perform the series of divisions:



The result is given by the last quotient and the previous remainders. We get $x_{\text{INT}} = 10100_2$.

2) The fractional part x_{FRAC} is obtained by a series of multiplications by 2. Thus, for 0.375:

$$\begin{aligned} 0.375 \times 2 &= \underline{0}.75 \\ 0.75 \times 2 &= \underline{1}.5 \\ 0.5 \times 2 &= \underline{1} \end{aligned}$$

The digits of the fractional part are given by the integer parts of results of the multiplications by 2. Here, we get $x_{\text{FRAC}} = 0.011_2$. In the end, $20.375_{10} = x_{\text{INT}} + x_{\text{FRAC}} = 10100.011_2$. Note that the base 2 representation is not necessarily finite, even if the base 10 representation is. Thus, 0.2_{10} will give us $0.001100110011\ldots$. The number of digits in the fractional part is infinite.

The error introduced by limiting the number of bits of the representation is known as the *truncation error*. The error from rounding up or down in this representation is the *rounding error*.

For example, $0.2_{10} \approx 0.0011001100$ by truncation and $0.2_{10} \approx 0.0011001101$ by rounding.

3.1.2. *Binary, octal and hexadecimal representations*

In the binary case, which is of particular interest to us, the two logic states used for the representation of data are symbolized by 0 and 1, whether the data are numerical or not. For practical reasons, the external representation often uses base 8 (*octal* representation) or 16 (*hexadecimal* representation).

The octal system uses eight symbols: 0, 1, 2, 3, 4, 5, 6, 7. Changing from base 2 to base 8 ($= 2^3$) is done instantly by dividing the binary digits into sets of three, "starting from the decimal point". For example:

$$1011101.01101_2 = 1 \mid 011 \mid 101.011 \mid 010 = 135.32_8$$

Base 16 uses the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Changing from base 2 to base 16 ($= 2^4$) is done the same way, by dividing the binary digits into sets of four, "starting from the decimal point". For example:

$$1011101.01101_2 = 101 \mid 1101.0110 \mid 10 = 5\text{D}.68_{16}$$

These bases not only give us a condensed representation of binary digits, but also make it possible to instantly convert to base 2.

3.2. Number representation conventions

Representations of numerical values require the use of rules that have been adopted to facilitate hardware or software processing. We will not mention all of the existing conventions, thus only the most common conventions are discussed here.

3.2.1. *Integers*

3.2.1.1. *Classical representation: sign and absolute value*

In this first convention, the representation uses a sign bit ($0 \rightarrow +$, $1 \rightarrow -$) and the absolute value of the number coded in base 2. Table 3.1 gives examples of 16-bit representations with this convention.

Decimal value	Sign/absolute value	Hexadecimal representation
$+2$	0/000 0000 0000 0010	0002
$+0$	0/000 0000 0000 0000	0000
-0	1/000 0000 0000 0000	8000
-3	1/000 0000 0000 0011	8003

Table 3.1. *Sign and absolute value representation*

In this representation, there is a $+0$ and a -0. Circuits designed to process operations on numbers coded this way have a higher complexity than those used for processing numbers in *two's complement*.

3.2.1.2. *One's complement representation*

Negative numbers $\tilde{x}$ with absolute value x are coded in n bits so that $x + \tilde{x} = 2^n - 1$ ("bit-by-bit" complement). Note that "0" can be represented by both $00\ldots0$ and $11\ldots1$.

3.2.1.3. *Two's complement representation*

This representation does not have the above-mentioned drawbacks. The *two's complement* coding of a negative number is done by taking the complement of its absolute value with respect to 2^n, where n is the number of bits used for the coding. The result of the binary sum of a number and its complement is n and a 1 carried over. If we define $\tilde{A}$ as the two's complement of A:

$$A + \tilde{A} = 2^n$$

Table 3.2 shows the correspondence between a few base 10 numbers and their base 2 representation using 16 binary digits.

Decimal value	Binary coding
2	0000 0000 0000 0010
1	0000 0000 0000 0001
0	0000 0000 0000 0000
-1	1111 1111 1111 1111
-2	1111 1111 1111 1110

Table 3.2. *Two's complement representation*

Switching from a positive integer to the negative integer with the same absolute value, and the other way around, is done by complementing the number bit-by-bit, and then adding 1 to the result of the complementation.

This is because, by definition:

$$\tilde{A} = 2^n - A = (2^n - 1) - A + 1$$

and $((2^n - 1) - A)$ is obtained by a *bit-by-bit complementation* of A.

EXAMPLE 3.2.– To obtain the 16-bit two's complement representation of the integer -10_{10}, we start with the binary representation of 10_{10}, that is $0000\ 0000\ 0000\ 1010_2$, and complement it:

$$(2^{16} - 1) - 10 = \overline{0000\ 0000\ 0000\ 1010} = 1111\ 1111\ 1111\ 0101$$

we then add 1:

$$1111\ 1111\ 1111\ 0110_2, \text{ i.e. FFF6}_{16} \text{ or } 177\,766_8$$

Common programming languages represent integers using 16, 32 or 64 bits in two's complement. It is easy to confirm that, in a 16-bit representation, any integer N is such that:

$$-32\,768_{10} \leq N \leq 32\,767_{10}$$

3.2.1.4. *Binary-coded decimal*

In some cases, rather than a binary representation, it is preferable to use coding that makes it easier to change from the decimal coding to the machine representation. A commonly used code is known as *binary-coded decimal* (BCD).

Each digit of a number represented in base 10 is coded using four bits. This representation, or other representations that share the same concept, is often used in languages used for management, such as COBOL. Although its internal process during operations is more complex than for numbers using the two's complement convention (due to the fact that hardware computational units usually operate on the two's complement representation), BCD coding presents a significant advantage in management: the absence of a *truncation error* (see section 3.2.2) when changing from base 10 to the internal representation.

The sign is coded separately, with the allocation of a half-byte assigned with a value different from the codes used for digits.

There are several BCD codes: BCD-8421 or BCD, Excess-3 BCD (XS-3, 0 is coded as 0011, 1 as 0100, etc.), and BCD-2421 (0–4 are coded the same way as in BCD and 5–9 are coded from 1011–1111). The last two codes are symmetric and the "bit-by-bit" complementation leads to the complement with respect to 9 of the corresponding decimal representation. In the BCD "comp-3" representation (a *packed decimal* representation by IBM), the + sign is coded as a C (1100) and − as a D (1101). The code for the sign is set as the least significant 4-bit code.

EXAMPLE 3.3.– The integer −1289 is represented by:

$$-1289_{10} = \underbrace{0000}_{0} \cdots \underbrace{0000}_{0}\ \underbrace{0001}_{1}\ \underbrace{0010}_{2}\ \underbrace{1000}_{8}\ \underbrace{1001}_{9}\ \underbrace{1101}_{\text{sign}} \text{comp-3 BCD}$$

3.2.2. *Real numbers*

There are two methods for representing real numbers: the *fixed-point* and *floating-point* representations.

3.2.2.1. *"Fixed-point" representation*

The *fixed-point* representation is often used in fields where the only machines available deal with integers and where calculation speed is crucial (e.g. in signal processing). Although tempting, the solution consisting of coding numbers in the "scientific" or "floating" format results in a significant increase in processing time and energy consumption.

The "fixed-point" representation arbitrarily places the decimal point between two digits of the binary representation. A representation in which k bits are allocated to the fractional part is known as a Q_k representation. It is obviously necessary to first analyze the dynamics of the values used to avoid any saturation that could be caused by the processing.

EXAMPLE 3.4.– The number 5.75 can be coded in $N = 16$ bits in Q_8 in the following way:

$$00000101\ .\ 11000000$$

where the point is represented symbolically to indicate the coding.

The corresponding integer is the integer that would be obtained by rounding the *multiplication* of the real number by 2^k. With this convention ($k = 8$), the number -5.75 would be coded as:

$$-5.75 \times 2^8 = -1472_{10} = \text{FA40}_{16} = 1111\ 1010\ .\ 0100\ 0000$$

From an operational perspective, the sum of two Q_k numbers is Q_k and their product is Q_{2k}. Note also that the significant figures are on the most significant bit side of the result, and not on the least significant bit side! This explains the presence in certain processors of *multiplication-shift* operators to have a Q_k after multiplication.

EXAMPLE 3.5.– Consider numbers coded in 4 bits in Q_3 fixed-point. We are going to perform a “signed” multiplication. Retrieving the result requires a shift to the right by three positions. There are three possible cases, which we will now examine.

1) First, let both numbers be positive:

$$\begin{cases} 0.5 \\ \times \\ 0.5 \end{cases} \rightarrow \begin{cases} (0.5 \times 8 = 4 =)0\,100 \\ \times \\ 0\,100 \end{cases} = 0001\,0000 \xrightarrow{3} 0010 = 0.25$$

2) With the same coding rules, we now consider two numbers of opposite signs:

$$\begin{cases} 0.5 \\ \times \\ -0.5 \end{cases} \rightarrow \begin{cases} 0\,100 \\ \times \\ 1\,100 \end{cases} = 1111\,0000 \xrightarrow{3} 1110 = -0.25$$

3) With the same coding rules, for two negative numbers:

$$\begin{cases} -0.5 \\ \times \\ -0.5 \end{cases} \rightarrow \begin{cases} 1\,100 \\ \times \\ 1\,100 \end{cases} = 0001\,0000 \xrightarrow{3} 0010 = 0.25$$

In practice, the main format used is Q_{15}, which requires shifting the numbers being processed so that their moduli are less than 0.5.

In this type of coding, managing the position of the point is therefore left as a task for the programmer, who must increase the number of bits of the representation to maintain the same precision. To perform P additions, $\log_2 P$ bits are added to the representation. Multiplications present a more difficult challenge. The result of the product of two numbers coded in N bits is represented by $2N$ bits. Performing a series of truncations can lead to disastrous results.

3.2.2.2. *Floating-point representation*

This type of representation is used when a variable x is declared, among other possibilities, in any of these formats:

```
float x ;  in C or Java
single x ;  in Basic
x : Float ; in Ada
```

in a few common high-level programming languages. The value stored in the cell with the name *x* will then be coded according to the format:

$$< \text{sign} >< \text{absolute value} > \times 2^{\text{exponent}}$$

The principle of coding, in other words the evaluation of the three values "sign", "absolute value" and "exponent", of a number x, is as follows:

1) We convert $|x|$ to binary.

2) The field associated with the absolute value is determined by *normalization*. This normalization consists of bringing the most significant 1 to the bit right before, or right after the decimal point. For example, the number $x = 11.375_{10} = 1011.011_2$ can be normalized in either of the two following ways:

- $1011.011_2 \rightarrow 0.1011011_2$ (format 1)

- $1011.011_2 \rightarrow 1.011011_2$ (format 2)

3) The shift imposed by the normalization corresponds to multiplications or divisions by two, and must therefore be compensated, which sets the value of the exponent. In the present case:

- $1011.011_2 = 0.1011011_2 \times 2^4 \rightarrow \text{exponent} = 4$

- $1011.011_2 = 1.011011_2 \times 2^3 \rightarrow \text{exponent} = 3$

The exponent must then be represented in binary.

This leads us to Remark 3.1.

REMARK 3.1.–

– Since the first digit before or after the point is always 1, it is not necessary for it to appear in the internal representation. This saves 1 bit in the representation, referred to as the *hidden bit*.

– This prohibits any representation of a number equal to zero. We must therefore find a code for the exponent that solves this problem. Consider the previous example once again, but choosing the following, arbitrary, rules:

- We decide to choose the normalization (format 1) that gives us $0.1011011_2 \times 2^4$. Since we do not keep the most significant 1, the remaining part is 011011. We will refer to this value as the *significand* or *mantissa*.

- The exponent is coded in *two's complement*, which in this case gives us 100_2.

- We choose as a convention for sign: 0 for the $+$ sign and 1 for the $-$ sign.

- The number will be coded in 32 bits, with 1 bit for the sign, 23 bits for the significand and 8 bits for the exponent. The resulting code, in the <sign/significand/exponent> format is then:

0 | 011 0110 0000 0000 0000 0000 | 0000 0100

With this convention, the exponent can assume values between -128_{10} and $+127_{10}$. The problem of coding "0" can be solved by assigning a value of $+127$ or -128 to the exponent, the significand's value being of no importance. For purposes of symmetry, let us choose -128.

To know if a variable is equal to 0, all we now need to do is to check if the value of the *representation* of its exponent is -128. However, the zero test of a number should be independent of its representation. This is why we can decide to add the constant 128 to the value of the exponent in order to obtain its code. In doing so, it is possible to verify whether any number is zero by testing the equality of the value <exponent> with 0.

In Table 3.3, we provide a few internal representations of the exponent in the case where the number of bits allocated to that exponent is eight:

REMARK 3.2.–

– The choice of the representation is the result of a compromise between the extent of the representable or *dynamic* range (specified by the number of bits in the exponent) and the *precision* needed (specified by the number of bits in the significand).

– The number of bits and the coding used for the significand and exponent fields are set by convention.

– The actual location of the different bit fields in memory depends on how they are used and on the processor they are implanted in.

Binary coded exponent	Exponent in base 16	Exponent in base 10
1000 0010	82	2
1000 0001	81	1
1000 0000	80	0
0111 1111	7F	-1
0111 1110	7E	-2
0111 1101	7D	-3

Table 3.3. *Examples of exponent coding*

3.2.3. *An example of a floating-point representation, the IEEE-754 standard*

This standard, recommended by the IEEE (Institute of Electrical and Electronics Engineers), was an initiative from Intel® under the direction of William Kahan[1]. IEEE-754 became an industrial standard in 1985 under the name *binary floating-point representation*. This standard defines two types of floating-point representations, known as *single precision* (*short real*) and *double precision* (*long real*).

– In the first case, the representation uses 32 bits. The significand is coded in 23 bits, m_0 to m_{22}, and the exponent in 8 bits, e_0 to e_7. The value of the number is coded as:

$$\text{value} = (-1)^{\text{sign}} \times 2^{(E_{INT}-127)} \times (1.\text{significand})$$

where E_{INT} is the value of the exponent's internal representation.

Sign	Exponent $e_7 \cdots e_0$	Significand $m_{22} \cdots m_0$
31	30 23	22 0

Table 3.4. *Single precision floating-points*

The standard also provides for the representation of non-numerical information. Table 3.5 shows some elements of this standard. NaN (*not-a-number*) is used for coding data that are not numerical in nature (e.g. when the division $0/0$ occurs).

[1] `http://www.cs.berkeley.edu/~wkahan/`

Exponent E_{INT}	Significand m	Value v
$E_{\text{INT}} = 255$	$m \neq 0$	$v = NaN$
$E_{\text{INT}} = 255$	$m = 0$	$v = (-1)^{sign} \times \infty$
$0 < E_{\text{INT}} < 255$		$v = (-1)^{sign} \times 2^{(E_{\text{INT}}-127)} \times (1.m)$
$E_{\text{INT}} = 0$	$m \neq 0$	$v = (-1)^{sign} \times 2^{(E_{\text{INT}}-126)} \times (0.m)$
$E_{\text{INT}} = 0$	$m = 0$	$v = 0$

Table 3.5. *Elements of the IEEE-754 standard*

The case of NaN is special. For example, when performing the multiplication $0 \times \infty$, certain processors code the result as NaN, while others simply trigger an exception (see chapter 6 on inputs and outputs).

NaN can be coded in two formats ($m \neq 0$):

$\phi 111\ 1111\ 1\underline{1}\phi\phi\ \phi\phi\phi\phi\ \phi\phi\phi\phi\ \phi\phi\phi\phi\ \phi\phi\phi\phi\ \phi\phi\phi\phi$

$\phi 111\ 1111\ 1\underline{0}\phi\phi\ \phi\phi\phi\phi\ \phi\phi\phi\phi\ \phi\phi\phi\phi\ \phi\phi\phi\phi\ \phi\phi\phi\phi$

The first case is known as *quiet NaN* (qNaN), the second as *signaling NaN* (sNaN). qNaNs are processed as "normal" significands in most cases, while sNaNs trigger an exception as soon as they are used.

– The double precision representation uses 64 bits. The significand is coded in 52 bits, m_0 to m_{51}, and the exponent in 11 bits, e_0 to e_{10}. The value of the number coded this way is:

$$\text{value} = (-1)^{\text{sign}} \times 2^{(E_{INT}-1023)} \times (1.\text{significand})$$

Sign	Exponent $e_{10} \cdots e_0$	Significand $m_{51} \cdots m_0$
63	62 52	51 0

Table 3.6. *Double precision floating-points*

EXAMPLE 3.6.– We wish to code 2.5_{10} as a *short real* type floating-point in the IEEE-754 standard. The different steps required are the following:

– convert 2.5_{10} to binary → 10.1_2;

– normalize → 1.01×2^1 (exponent 1, significand 01);

– calculate the representation of the exponent E_{INT}:

$$E_{\text{INT}} = exponent + 127 \rightarrow E_{\text{INT}} = 128$$

– convert it to binary → $E_{\text{INT}} = 10000000$;

– represent the sign bit, in this case *s* = 0.

The machine representation of 2.5_{10} is therefore:

$0 \mid 100\ 0000\ 0 \mid 010\ 0000\ 0000\ 0000\ 0000\ 0000$

that can be expressed in hexadecimal as: $40\ 20\ 00\ 00_{16}$.

In the C language, the floating-point representation can be extracted as follows:

```
#include <stdio.h>
int main (int argc, const char * argv[]) {
    union { unsigned char repchar[4];
            float repfloat;} number;
    short k;
    number.repfloat=2.5;
    printf("Floating: %f\n",number.repfloat);
    for (k=0;k<4;k++)
        printf("%d %2d\n",k,number.repchar[k]);
    return 0;
}
```

In the previous case, and for a "little-endian" representation, we get:

```
Floating: 2.500000
0  0
1  0
2 32
3 64
```

3.2.4. *Dynamic range and precision*

An important question in floating-point representation involves:

– the *precision*, specified by the endianness of the last bit used in the representation;

– the *dynamic range*, defined as the interval between the smallest and the largest positive numbers that can be represented.

Table 3.7 shows the different bounds of the IEEE standard. In the IEEE-754 standard, the largest number that can be represented in single precision is $1.11 \cdots 1_2$

$\times\, 2^{127}$ (23 1's after the decimal point). The maximum value for the internal representation of the exponent is FE_{16}, since FF_{16} is reserved for the case of infinite values or "non-numerical values". The precision is specified by the number of significant figures.

	Single precision	Double precision
Largest positive number	$2^{127}(2-2^{-23}) \approx$ $3.40282346 10^{38}$	$21023(2-2^{-52}) \approx$ $1.7976931348623157\ 10^{308}$
Smallest positive number	$2^{-126} \approx$ $1.17549436 10^{-38}$	$2^{-1022} \approx$ $2.2250738585072014\ 10^{-308}$
Largest negative number	$-2^{126} \approx$ $-1.17549436 10^{-38}$	$-2^{1022} \approx$ $2.2250738585072014\ 10^{-308}$
Smallest negative number	$-2^{127}(2-2^{-23}) \approx$ $3.40282346 10^{38}$	$-21023(2-2^{-52}) \approx$ $-1.7976931348623157\ 10^{308}$

Table 3.7. *A few minimum and maximum values*

3.2.5. *Implementation*

The processing of numbers represented in floating-point is usually performed by specialized computational units known as *floating-point units*. With processors lacking such capabilities, it is necessary to write the arithmetical functions that perform at least the four basic operations. Consider the case of a multiplication involving real numbers. The sequence of operations is as follows:

– The exponents are added. A sum that exceeds 7F_{16} results in a floating-point overflow. A sum below 80_{16} results in a floating-point underflow.

– The significands are multiplied.

– The result is normalized. This can cause an overflow or an underflow.

It is also necessary to make sure there are no special cases to consider. If the two operands are -1.0×2^n and -1.0×2^m with $m + n = 7\text{F}_{16}$, the result will lead to an overflow, since we get $0.5 \times 2^{m+n+1}$.

3.2.6. *Extensions of the IEEE-754 standard*

Because of rounding errors caused by the binary floating-point representation, a standard known as IEEE-854-1987 (*IEEE Standard for Radix-Independent Floating-Point Arithmetic*) was proposed, which generalized the previous standard.

The numbers x are coded according to the format $x = \pm c \times b^{exp}$. In fact, the standard indicates that $b = 2$ or $b = 10$. This is why it is referred to, wrongly, as the *decimal floating-point* (DFP) representation, for which b is restricted to 10.

The IEEE-754 standard was later amended to include IEEE-854-1987. It now carries the name IEEE-754-2008 [IEEE 08]. This standard emphasizes not only floating-point coding but also their "arithmetic behavior" during operations (rounding effects, calculations involving $\pm\infty$, the handling of exceptions, etc.).

The DFP representation defines three formats – 32, 64 and 128 bits – and two codes – DPD (*densely packed decimal*) and BID (*binary integer decimal*). The number of significant numbers and the values of the exponent are shown in the following table.

	32 bits	64 bits	128 bits
Significant figures	7	16	34
exponent e	$-95 \le e \le 96$	$-383 \le e \le 384$	$-6143 \le e \le 6144$

The hardware implementation of floating-point numbers in RFD was never completed. The lack of applications (essentially in management computing) and the expected energy consumption are such that only a software implementation was completed.

3.3. Character representation

3.3.1. *8-bit representation*

Traditionally, characters (letters, numbers and punctuation symbols) were *coded* in a byte. The most commonly used codes are ASCII (*American Standard Code for Information Interchange*), proposed by Bob Bemer in 1961, which uses seven bits, plus an eighth bit to check for parity (Table 3.8), and the EBCDIC (*Extended Binary Coded Decimal Interchange Code*), which first appeared on IBM/360 machines in 1964, and was implanted on eight bits.

To find the hexadecimal ASCII code of a character, we have to read the most significant digit in the top line, and the least significant digit in the left column. For example, the character <G> corresponds to the code 47_{16}. Table 3.9 shows the codes for the set of North American characters.

Certain codes are reserved for specific uses: control of the display, managing data flow for communication between machines, etc. In the ASCII code, the codes 0 to $1F_{16}$ are reserved for this purpose (Table 3.10).

Least	Most significant							
significant↓	0	1	2	3	4	5	6	7
0	NUL	DLE		0	△	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	△	2	B	R	b	r
3	ETX	DC3	△	3	C	S	c	s
4	EOT	DC4	△	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	△	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	△	k	△
C	FF	FS	,	<	L	△	l	△
D	CR	GS	-	=	M	△	m	△
E	SO	RS	.	>	N	△	n	~
F	SI	US	/	?	O	_	o	DEL
△: character not defined by the standard (national choices)								

Table 3.8. *Table ASCII*

Least	Most significant							
significant ↓	0	1	2	3	4	5	6	7
0	NUL	DLE		0	@	P	‘	p
⋮	...							
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
⋮	...							
7	BEL	ETB	’	7	G	W	g	w
⋮	...							
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Table 3.9. *Specificities of the ASCII code*

Code	Symbol	Significance
04	<EOT>	End of transmission
07	<BEL>	Audible bell
0C	<FF>	Form feed (causes the the screen to go blank on a screen terminal)
0E	<SO>	Produces a complement to the existing set of characters
1B	<ESC>	Escape character

Table 3.10. *A few of the control characters*

Code 7, for example, denoted by BEL, corresponds to the terminal alarm. These codes are usually directly accessible on the keyboard by simultaneously pressing <CONTROL> and a letter. <CONTROL> and <A>, for example, yields code 1, and <CONTROL> and <G> yields code 7, which corresponds to the “beep”, etc.

These codes are also usable from a program written in a high-level language by simply “printing” them, just like alphanumerical characters. The control characters we mentioned can only be interpreted by what is known as an ASCII type terminal.

Example 3.7.– The character string `Computer` followed by a move to the next line will be coded with the sequence of bytes:

43, 6F, 6D, 70, 75, 74, 65, 72 (hexadecimal)

Note that the move to the next line is coded using two characters: 0D which is the *Carriage Return*, the return to the beginning of the line, and 0A which takes us to the next line (*Line Feed*). Line changes in text files created by the editor in MS-DOS, Microsoft Corp.’s “historical” operating system, are coded this way. This is because certain terminals from the era of the birth of the “PC” were electromechanical, and required these two commands to return the carriage to the beginning of the line and trigger the rotation of the printer drum.

Example 3.8.– Table 3.11 shows examples of internal representations, where the integers and real numbers are coded in 32 bits, in two’s complement for the integers and in the IEEE-754 format for the floating-point representations. The characters are coded in the ASCII format.

3.3.2. *Modern representations*

8-bit character coding as it was originally practiced does not provide a way to code all of the characters in all of the alphabets (e.g. ő and ǒ). A solution was to introduce “national code pages”. But again, this solution did not prove satisfactory. The Unicode consortium[2] developed a standard that covers every other existing ISO (International Organization for Standardization) system. This standard offers more than simple coding. It “names” the characters, and describes processing models that can be applied to them and the semantic links between them.

Data being coded	Number of occupied bits	Internal representation (hexadecimal)
Number 1 as an integer	32	0000 0001
Number -1 as an integer	32	FFFF FFFF
Floating-point 1	32	exp. 7F, significand 0, sign 0
Floating-point -1	32	exp. 7F, significand 0, sign 1
Character 1	8	31
Characters $+1$	16 (2 characters)	2B 31
Characters -1	16 (2 characters)	2D 31
Characters 1.0	24 (3 characters)	31 2E 30
Characters -1.0	32 (4 characters)	2D 31 2E 30

Table 3.11. *Examples of internal representations*

The three main “vector” fonts, which replaced the original *bitmap* fonts, are PostScript, TrueType and OpenType. This last font, introduced in 1996, is a superset of the previous two. The XƎTEX [GOO 10] documentation gives an interesting history of the creation of these fonts and of the relevant technical characteristics.

OpenType accepts Unicode coding in more than four hexadecimal digits and typographic rules – which are not alway implemented – for the use of *glyphs* (substitutions, position, ligature rules, etc.). Many programs now accept this to represent characters, or ideograms, from a great variety of languages.

EXAMPLE 3.9.– In traditional Chinese, the word “microprocessor” is written as 处理器 in the STSong font, which can be expressed phonetically as “chǔ lǐ qì”. The three ideograms are coded as 5904_{16}, 7406_{16} and 5668_{16}, respectively. As another example, “luck” is written as:

运气

in simplified Chinese with the STKaiti font.

[2] http://unicode.org/

3.4. Exercises

Exercise 3.1 (Integer representation) (Hints on page 327)

In a highlevel language, write a program that iteratively increments a variable declared as an integer. Define a stop condition that will give you the highest possible value with this code. Use the value of this number to infer how many bits are used for coding it.

Exercise 3.2 (Single precision floating-point representation) (Hints on page 328) Verify that the internal representation

```
║0  0
║1  0
║2 72
║3 64
```

corresponds to $x = 3.125$.

Exercise 3.3 (Fixed-point and floating-point representations) (Hints on page 328) Consider the real number $x = -3.82$ represented as a single precision floating-point number on a PC-type machine with an x86 series Intel® processor.

1) Write the internal representation of x in hexadecimal.

2) Write the Q_{12} fixed-point representation of x in 16 bits.

Exercise 3.4 (Short floating-point representation) (Hints on page 328)

The documentation for the Texas Instruments DSP TMS320C30® [TEX 89] describes a short floating-point representation as follows:

In the short floating-point format, floating-point numbers are represented by a two's-complement 4-bit exponent field (e) and a two's-complement 12-bit mantissa field (man) with an implied most-significant non-sign bit.

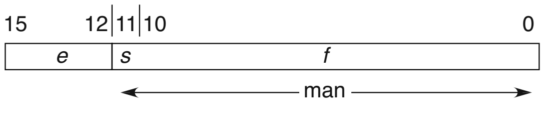


Figure 3.1. *Short floating-point format*

Operations are performed with an implied binary point between bits 11 and 10. When the implied most-significant non-sign bit is made explicit, it is located to the *immediate left of the binary point. The floating-point two's-complement number x in the floating-point format is given by:*

$$x = \begin{array}{ll} 01.\mathrm{f} \times 2^{e} & \textit{if}\, s = 0 \\ 10.\mathrm{f} \times 2^{e} & \textit{if}\, s = 1 \\ 0 & \textit{if}\, e = -8,\ s = 0,\ \mathrm{f} = 0 \end{array}$$

The following reserved value must be used to represent zero in the short floating-point format: "$e = -8$, $s = 0$, $f = 0$".

An exponent equal to -8 is actually reserved for special cases.

1) Write the representations for the maximum and minimum positive and negative values.

2) Write the representation of $x = 10^{-2}$.

3) Write the representation of $x = -64.3_{10}$.

PART 2

Programming Model and Operation

Chapter 4

Instructions

As we have seen in previous chapters, any activity of a machine corresponds to the execution of a *program*. This program comprises of a series of *instructions* defining the elementary operations that the machine has to execute. These operations require gaining access to operands that may be found in memory, in working registers, in the registers of exchange units, etc.

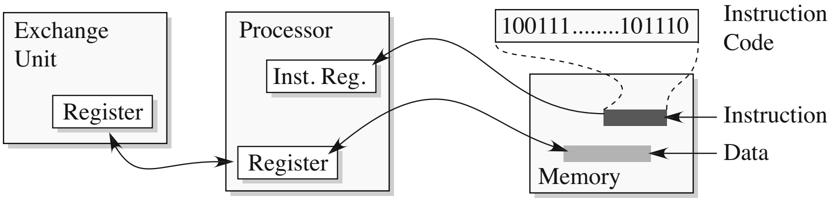


Figure 4.1. *Accessing the instructions and the operands*

The instruction code must identify three types of information: the operation performed, the operands and the means of gaining access to these operands. In practice, the user only rarely has the opportunity or the need to access the binary instruction codes, which together form the *machine language*.

EXAMPLE 4.1.– On the Intel I8086® microprocessor, for example, the instruction coded as $\texttt{0100 0000}_2$, i.e. 40_{16}, means “increment the ax register”, while 43_{16} means “increment the bx register”.

Rather than using this obscure representation, we rely on mnemonics that clarify the role of each instruction. In the previous example, the instructions will be denoted by `inc ax` and `inc bx`, respectively. Writing a program in mnemonics requires the use of a "translator", known as an *assembler*, with the corresponding language being referred to as an assembly language.

When the user needs to work "close to the machine", they require some basic information, such as working registers, in order to gain access to data, all of the available instructions and the size of the program counter. This information is sometimes known as a *programming model*.

4.1. Programming model

In what follows, we describe the programming model for the Intel I8086®. Using this dated processor makes it easier to understand the examples given at the end of this chapter. The more recent processors from Intel Corp. are based on this processor, at least with respect to the set of instructions.

4.1.1. *The registers of the I8086*

The Intel I8086® is equipped with the following registers:

– `ax`, `bx`, `cx`, `dx`, the-16 bit working registers; the most and least significant bits for each of these registers can be accessed using the mnemonics `ah`, `al`, `bh`, `bl`, etc.;

– `si` (Source Index), `di` (Destination Index), the 16-bit index registers;

– `bp` (Base Pointer), the 16-bit stack address register;

– `sp` (Stack Pointer), the 16-bit stack pointer;

– `ip` (Instruction Pointer), the 16-bit program counter, which is used for addressing 64 kB of memory;

– the flag register (overflow, zero, carry, auxiliary carry, parity, sign) and other control bits (direction, etc.);

– `cs` (code segment), `ds` (data segment), `ss` (stack segment), `es` (extra segment), the 16-bit "segment" registers involved in the construction of addresses in memory.

We will describe these registers in detail in the examples provided in the following.

The flag register consists of six 1-bit indicators:

– Z is set to 1 when the destination operand has been set to zero or when the result of a comparison operation is "equality";

– S, the sign bit, is a copy of the most significant bit of the result operand;

– O, the overflow bit (page 24), is modified by most arithmetic operations (operations in two's complement) and logic operations;

– C, the carry bit, is modified by most arithmetic and logic operations;

– P, the parity bit, provides information on the number of 1 bit in the result of certain 8-bit arithmetic or logic operations;

– A, the auxiliary carry bit, is used by the processor for instructions that handle BCD coded data (`DAA` and `DAS` instructions).

4.1.2. *Address construction and addressing modes*

4.1.2.1. *Address construction*

The segmentation system used by Intel® allows access to 1 MB (2^{20} bytes) of memory, although the registers are coded in 16 bits, which limits the address space to 64 kB. Access to the entire memory space is done using four registers: `cs`, `ds`, `ss` and `es`. The instructions provide 16-bit addresses. A mechanism adds to these addresses the content of the segment register offset by 4 bits to the left. The final address found on the external bus is therefore 20 bits in length:

$$\text{Final address} = (2^4 \times \text{ segment register}) + \text{ offset} \qquad [4.1]$$

where the offset is the relative address in the segment.

EXAMPLE 4.2.– If the program counter (the `ip` register) contains $1A00_{16}$, and if the `cs` register contains 2010_{16}, the address generated on the external bus for accessing this instruction will be (Figure 4.2):

$$(2010_{16} \times 10_{16}) + 1A00_{16} = 20100_{16} + 1A00_{16} = 21B00_{16}$$

Each instruction is associated by default with one or two segment registers. As a result, a memory write operation by the `mov` instruction will only modify a word in the data segment located by `ds`.

EXAMPLE 4.3.– The instruction `mov [bx],al`, which is used for copying the content of the 8-bit `al` register into the memory word whose address is in `bx`, triggers the address calculation (Figure 4.3):

$$(16 \times [\texttt{ds}]) + \texttt{bx}$$

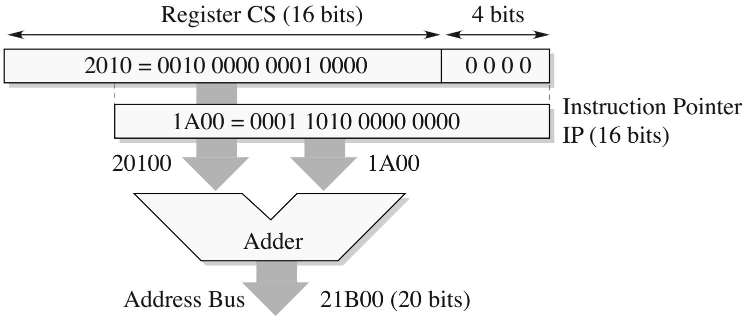


Figure 4.2. *Calculation of the instruction address*

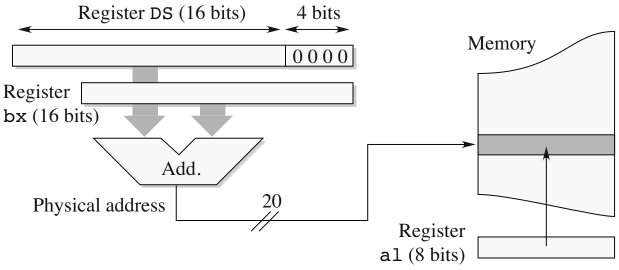


Figure 4.3. *Calculation of the address and writing of the operand*

4.1.2.2. *Addressing modes*

The different ways of accessing data in a Intel I8086® are limited by the specialization of the registers. Unlike many other processors whose registers can be used and handled in an ordinary fashion, Intel® defines the roles attributed to the registers. The addressing modes are described in Appendix A.1. Most of the instructions have two operands. The first operand constitutes the destination and the second the target. `add bx, ax`, for example, corresponds to the operation `bx = bx + ax`.

In the following examples, we will describe the operation and coding of a few instructions.

EXAMPLE 4.4. (Immediate access) – Consider the instruction `mov bx,1000H`, equivalent to `bx = 0x1000;` in Java. This instruction is coded as `BB0010`:

	1011 w reg	data (1/2)
Code	1011 1 011	0000 0000 0001 0000

The w bit (=1) indicates that we are dealing with the transfer of a 16-bit word. The bx register is numbered 011 and the value of the operand (1000) appears with the least significant bit first. The data field can be one or two bytes in length (1/2 indication).

The code for the instruction mov bl,0FFH is:

	1011 w reg	data (1/2)
Code	1011 0 011	1111 1111

The w bit indicates the transfer of a byte. The bl register is numbered 011 and the value of the operand (FF) appears in the second byte of the code. The offset field can be one or two bytes in length (1/2 indication).

EXAMPLE 4.5. (Direct access)– Consider the instruction mov bx,[1000H], which triggers the transfer of a 16-bit word, located at the address 1000H in a memory segment, to bx. The code for this is 8B1E0010:

	1000 10 d w	mod reg r/m	disp(0/2)
Code	1000 10 1 1	00 011 110	0000 0000 0001 0000

The d bit is set to 1, which indicates a transfer from memory to register. This transfer involves a 16-bit word (w=1). mod=00 and r/m=110 correspond to direct addressing (the address is provided “directly” in the instruction). The register number is 011 (bx). The “offset” field can be zero or two bytes in length (0/2 indication).

EXAMPLE 4.6. (Indirect access)– In this type of access, the address is constructed from the content of the registers. Consider the instruction mov al,[bx+si+10] for reading a byte stored in memory. The corresponding code is 8A4010.

	1000 10 d w	mod reg r/m	disp(0/2)
Code	1000 10 1 0	01 000 000	0001 0000

The d bit is set to 1, which indicates a transfer from memory to register. This transfer involves a byte (w=0). mod=01 corresponds to an indirect access with an 8-bit offset. reg=000 codes al and r/m=000 give the “mode” ds:[bx+si+disp]: the address calculation uses the ds segment and the address is calculated through the sum (bx+si+10).

The default segment, ds, can be changed: if we wish to access the code segment, we simply need to provide mov al,cs:[bx+si+10]. This instruction actually consists of two instructions:

```
cs:                ; instruction which modifies the default segment
                   ; only applies to the following instruction
mov al,[bx+si+10]  ; transfer
```

4.2. The set of instructions

In what follows, we will concern ourselves only with instructions for the architecture we have chosen.

4.2.1. *Movement instructions*

Movement instructions include transfers between registers, from memory to register and from memory to memory. We saw a few examples of these previously. There are other instructions of this kind. The `movsb` (*move string byte*) instruction, for example, is used for transferring `cx` bytes from memory to memory.

EXAMPLE 4.7.– The following excerpt from a program transfers 16 bytes from memory to memory:

```
cld                  ; direction
lea si,sourcestring ; source string address --> si
lea di,targetstring ; target string address --> di
mov cx,10H           ; number of transfers
rep movsb            ; transfer of cx bytes
```

`cld` (*clear direction flag*) indicates that addresses are incremented. The transfer is done from `ds:si` to `es:di` by incrementing `si` and `di` at each iteration. `rep` is a *repeat string* instruction prefix. `lea` (*load effective address*) loads `si` or `di` with the address of the source and target strings, respectively.

Intel® made the choice of not modifying flags during the movement instructions. `mov ax,0`, for example, does not set the Z flag. This is not the case for all processor designers. In the Motorola M68000® processor, for example, which dates back to the same era as the Intel I8086®, the `move` instruction modifies the zero flag Z and the sign flag N.

4.2.2. *Arithmetic and logic instructions*

4.2.2.1. *Arithmetic instructions*

Arithmetic instructions – adding, subtracting, multiplying and dividing – act on integers.

EXAMPLE 4.8.– The following instructions execute the sum of two 32-bit integers. The two 32-bit operands are denoted by `oper1` and `oper2`:

```
lea si,oper1    ; operand 1 address in si
lea di,oper2    ; operand 2 address in di
mov ax,[si]     ; memory --> ax (least significant)
add ax,[di]     ; add least significant words (16 bits)
mov [bx],ax     ; storage in memory, ax --> memory
mov ax,[si+2]   ; memory --> ax (most significant)
adc ax,[di+2]   ; add most significant bits
mov [bx+2],ax   ; storage in memory
```

Summing the 16 least significant bits produces a carry that is used by the addition adc. Obviously, C must not have been modified in the meantime.

inc (increment), dec (decrement) and neg (sign change in two's complement) are some of the arithmetic instructions offered by every designer.

Multiplications and divisions can be *signed* (imul, idiv) or *non-signed* (mul, div).

4.2.2.2. *Logic instructions*

Logic instructions are "bit-by-bit instructions"; four of them are: and, or, xor and not.

EXAMPLE 4.9.– Manipulating of a byte:

```
mov al,[bx]      ; memory --> al
and al,7FH       ; most significant bit <-- 0
not al           ; complementation
```

The value (byte) read in memory is loaded into al. The most significant bit is set to zero by a "logic AND" with the constant 0111 1111. The result is then complemented "bit-by-bit".

REMARK 4.1.– The instructions mov ax,0 and xor ax,ax yield the same result. However, the former is coded in three bytes (code B8 00 00), and the latter in two bytes (code 31 C0). This means that the first instruction is "slower" than the second, but does not modify the flags.

4.2.2.3. *Comparisons*

Comparisons are performed by subtracting the operands without propagating the result, but with a flag update. We will later discuss the use of these instructions for conditional branching.

4.2.3. *Shift instructions*

There are eight *shift* instructions, including shifts and rotations, the shifting direction and management of the “output” bit (Figure 4.4).

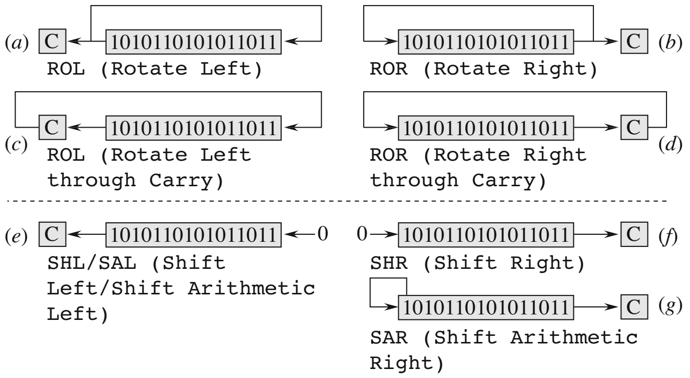

(a)
C
1010110101011011
ROL (Rotate Left)
(b)
ROR (Rotate Right)
(c)
ROL (Rotate Left through Carry)
(d)
ROR (Rotate Right through Carry)
(e)
0
SHL/SAL (Shift Left/Shift Arithmetic Left)
(f)
SHR (Shift Right)
(g)
SAR (Shift Arithmetic Right)

Figure 4.4. *The types of rotations are – (a), (b), (c), (d) – and shifts are – (e)– (f), (g). The carry bit is denoted by C. The logic and arithmetic shifts to the left are identical, whereas there is a difference between the shifts to the right, due to the necessary “sign propagation” in the arithmetic case*

EXAMPLE 4.10.– The following sequence performs the multiplication by two of a 32-bit integer:

```
mov ax,[bx]        ; least significant operand --> ax
shl ax,1           ; ax*2 --> ax and most significant bit --> CY
mov [si],ax        ; result --> memory
mov ax,[bx+2]      ; most significant operand --> ax
rcl ax,1           ; ax*2 --> ax and recovery of CY in
                   ; least significant bit
mov [si+2],ax      ; result --> memory
```

The operand is at the address given by bx. The shift is performed in two phases by a shift and a rotation at the address given by the register si.

REMARK 4.2.– In the Intel I8086®, the number of shifts is necessarily equal to one since it is given “by immediate value”. If we wish to specify another value, we use the cl register. This limitation was eliminated in later processors.

4.2.4. *Branching*

4.2.4.1. *Unconditional branching*

A branch is *unconditional* if the instruction provides the branching address:

– *As an immediate and absolute value:* With an Intel I8086®, for example, when the processor starts, the pair <CS register:program counter> is initialized as F000:FFF0. This address is the location for a branching instruction, such as `jmp F000:E05B` (code `EA5B0E00F0`), which is used to "jump" to the address of the beginning of the start-up process. The pair <CS:program counter> is directly loaded with F000:E05B.

– *As a relative value:* Except for very specific cases such as the case we just saw, the address is coded in the instruction as a *shift* relative to the program counter that corresponds to the following instruction (branching to the following instruction is done using a zero shift).

EXAMPLE 4.11.– The following sequence of instructions illustrates the operation of a `jmp` instruction. The instruction codes are shown on the left:

```
EB 02      jmp nxt      ; branch towards the mov instruction
                        ; EB=jmp, shift=2
04 03      add al,3     ;
88 C4 nxt: mov ah,al    ; result --> memory
```

The shift is calculated relative to the current value of the program counter. Since the counter is already incremented during the actual execution of the `jmp` command, PC points to `add`. We branch to the following instruction; in other words, to `mov` with the address <address of `add` + shift (size of the `add` = 2)>.

– *Through an indirection register:* The instruction `jmp [bx]` triggers a branch to the address (relative to the actual address) given by the content of `bx`.

EXAMPLE 4.12.– In the following sequence, the branch address (the offset of the `nxt` tag) is loaded by `lea` into `bx`. The instruction `jmp [bx]` then executes the branch to `nxt`. The addresses are shown in the first column, followed by the instruction codes.

```
0100 8D 1E FD 00     lea bx,nxt
0103 ff 27           jmp [bx]
0105 41              inc cx
...
0200 05 01      nxt: ...
```

4.2.4.2. *Conditional branching*

Conditional branches allow for the implementation of program structures, such as {`if ... then ... else ...`}, {`while ... end`} or {`do ... until`}. The

Intel I8086® offers a set of 16 branches using flags: `je` (`jump if equal`), `jne` (`jump if not equal`), `jae` (`jump if above or equal`), `jc` (`jump if carry=1`), etc.

EXAMPLE 4.13.– The following sequence reads a 16-bit parameter in the stack. If this parameter is zero, the CY flag is set to 1, otherwise it is set to 0:

```
8b 46 08          mov ax,[bp+8]
3d 00 00          cmp ax,0      ; compare ax with 0
eb 03             je nxt1       ; jump if equal
f9                stc           ; set carry bit
eb 01             jmp nxt2      ;
f8          nxt1: clc           ; clear carry bit
            nxt2: ...
```

REMARK 4.3.– `bp` serves as a pointer in the stack segment. It is managed by the user. The program fetches a piece of information in the stack at the address `bp`+8.

The comparison operation could have been done with `or ax,ax` (code `09C0`). This instruction modifies the Z flag, thus indicating whether the result, i.e. `ax`, is zero.

4.2.4.3. *Calling functions*

There are two instructions for calling functions `call` and `int` (*interrupt*), which are associated with the instructions `ret` and `iret` (*interrupt return*). We will mention `int` again in Chapter 6. In the Intel I8086®, calling a function relies on the stack mechanism described on page 22. When a `call adrtrt` is executed:

– the processor saves the *return address*, in other words the address of the instruction that follows the `call adrtrt`, in the stack;

– it loads the program counter, and sometimes CS depending on the type of `call` (near or far `call`), with the processing address `adrtrt`;

– at the end of the process, a `ret` instruction, which is another branching instruction, recovers from the stack the return address that was saved there to load it into PC.

EXAMPLE 4.14.– The following function returns the number of 1 bit in a byte. The input parameter is `ah`, the output parameter `dl`:

```
50          func: push ax   ; save ax
51                push cx   ; save cx
30 d2             xor dl,dl ; bit counter <-- 0
```

```
b9 0800        mov cx,8  ; total number of iterations
f0 ec   mylp: shr ah,1  ; shift right by 1 bit in CY
80 d2 00       adc dl,0  ; sum
e2 f9          loop mylp ; as long as cx>0 go to mylp
59             pop cx    ; restore cx
58             pop ax    ; restore ax
c3             ret       ; recover return address
```

The main program is as follows:

```
mov  ah,53  ; input parameter
call func
```

REMARK 4.4.– A push instruction stores its operand at the "top of the stack" and decrements the stack pointer. pop performs the opposite operation: it recovers the value at the top of the stack and stores it in the operand, then increments the stack pointer. This mechanism is flexible and powerful: it avoids issues of managing placement in memory for saving registers modified in the functions. Some processors do not offer this easy option.

The `loop mylp` instruction is a conditional branch instruction described by:

```
dec  cx
jnz  mylp ; if cx not zero, goto mylp
...
```

4.2.5. *Other instructions*

Modern processors are equipped with a much more diverse set of instructions that goes hand-in-hand with the increasing complexity of the hardware architecture:

– As we saw previously, the arithmetic instructions only deal with integers: section 5.4.1 will show how real numbers in floating point are handled by an arithmetic coprocessor.

– Vector processing capabilities have been added to the existing architecture: section 5.4.2 presents the MMX and SSE instructions used for such devices.

– Intel Westmere® processors [INT 10] were given additional instructions designed for facilitating Advanced Encryption Standard (AES) encrypting and decrypting.

– Instructions that can be used to synchronize tasks or accesses to critical resources make it simpler to write multitask systems and applications. The Intel I80386®, for example, includes the BTS (`Bit Test and Set`) instruction, referred to as *atomic* because it executes several non-interruptible operations: a bit transfer from a memory operand to the CY flag, then the setting of this same bit to 1.

4.3. Programming examples

The following program, written in C, calls a function written in an assembly language to perform the sum of two 16-bit integers. The target machine is an Intel® Core2 Duo$^{\mathrm{TM}}$. This example illustrates the problems we face when passing parameters between high-level language and assembly language.

```
/* expleparam.c */
#include <stdio.h>
int main() {
    void mysum(short, short, short*);
    static short a=4, b=7, c;
    static short *ptrc=&c;
    mysum(a,b,ptrc);
    printf("%d\n",*ptrc);
    return 1;
}
```

The function `mysum(short, short, short*)` has two arguments `short` (16 bit) and a 32-bit pointer. The compiler generates three 32-bit instructions to place operands at the "top of the stack" (`pushl` instruction) in this order: pointer on the result, the operand b, and then the operand a. After executing `call` (stacks the return address in 32 bits), the stack looks as follows:

Address in the stack		Content of the stack
n	←	return address in 32 bits
$n+4$	←	operand a in 32 bits
$n+8$	←	operand b in 32 bits
$n+12$	←	pointer toward result in 32 bits

The assembly language program that performs the sum is the following (here we are using the AT&T® syntax, which is quite different from the Intel® syntax, though the latter can still be used with the help of the directive `.intel_syntax`):

```
     .text
.globl _mysum
_mysum:
     pushl     %ebp           ; ebp --> stack
     movl      %esp, %ebp     ; esp --> ebp
     pushl     %eax           ; save eax
     pushl     %edx            ; save edx
     movzwl    12(%ebp), %eax  ; b --> eax
     addw      8(%ebp), %ax    ; a + ax --> ax
     movl      16(%ebp), %edx  ; result address
     movw      %ax, (%edx)     ; write result
     popl      %edx            ; restore
     popl      %eax            ; restore
     popl      %ebp            ; restore
     ret
```

After the `pushl %ebp`, the stack contains:

Address in the stack	Content of the stack
m ←	saved copy of `ebp` in 32 bits
$m + 4 = n$ ←	return address in 32 bits
$m + 8$ ←	operand a in 32 bits
$m + 12$ ←	operand b in 32 bits
$m + 16$ ←	pointer toward result in 32 bits

– the instruction `movzwl 12(%ebp),%eax` loads the operand b into `eax` (32 bits) by setting the 16 most significant bits to 0;

– the instruction `addw 8(%ebp),%ax` executes the sum;

– the instruction `movl 16(%ebp),%edx` loads `edx` with the pointer indicating the result;

– the instruction `movw %ax,(%edx)` stores the result in the address `[edx]`;

– after executing the instruction `ret`, the value of the stack pointer has to be reset to the value it had before calling the function. This is because the parameters are still on the stack and the stack pointer does not have the correct value. In C, `esp` is incremented by adding the number of bytes taken up by the parameters, in this case 24. This is one of the specificities of the language used. A basic compiler does nothing, and it is the programmer's responsibility to leave the stack the way they found it. This is done with the instruction `ret 24`, which is the parametrized version of `ret`.

The executable is generated by the command:

```
gcc -Wall expleparam.c mysum01.s -o myexec
```

where `mysum01.s` is the assembly language program file.

In the Intel® syntax, we would have:

```
     .intel_syntax
     .text
.globl _mysum
_mysum:
     push   ebp
     mov    ebp, esp
     push   eax
     push   edx
     xor    eax,eax
     mov    ax, [ebp+12]
     add    eax, [ebp+8]
     mov    edx, [ebp+16]
     mov    [edx], ax
     pop    edx
     pop    eax
     pop    ebp
     ret
```

4.4. From assembly language to basic instructions

The development of a program requires the following steps (Figure 4.5):

– Writing the source files S_1, S_2, etc.

– Invoking translation programs – an assembler for the assembly language, a compiler for high level languages, etc. – to go from the symbolic representation to binary and to assign addresses to the variable names. The files created in this phase are known as *object files*. Here they will be denoted by O_1, O_2, etc.

– Calling the *linker*, which takes as input the object files, completes them by consulting “libraries” provided with the languages (B_1, B_2, etc.), and creates a binary executable (or rather loadable) file denoted by Ex.

In order for a file to be executed, it has to be loaded into memory. This is achieved by “typing” its name on the keyboard or “double-clicking” on its icon: the *loader* will find it on the drive and load it into memory.

4.4.1. *The assembler*

Assembly language is a symbolic representation of the binary coding of the machine instructions. This binary coding is the only form of information that the processor can work with. The assembly is charged, not only with the translation in binary code, but also with the assignment of an address to each symbol (variable names and tags) used by the program.

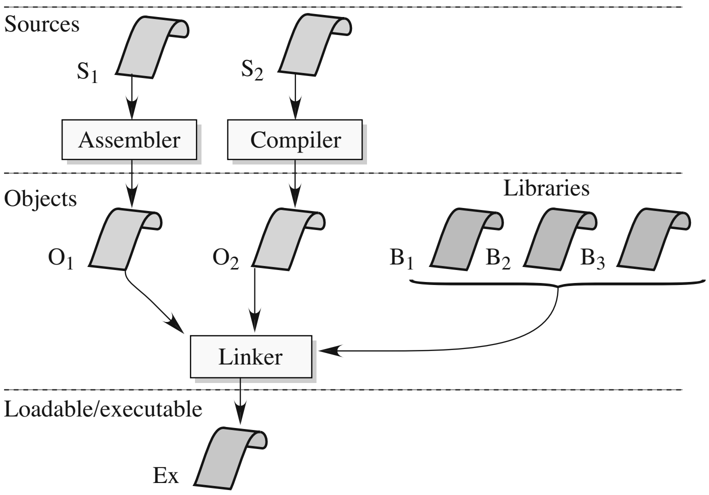


Figure 4.5. *Application development*

The assembly language also features directives and pseudo-instructions used for reserving memory cells, initializing them or facilitating the task of writing programs.

Among other things, pseudo-instructions make it possible to manage the allocation of memory for variables: size, alignment and type. The following example presents some of the pseudo-instructions commonly used with the Intel I386® processor assembly language.

```
Char_One:
.byte 0x31     ; a byte with character "1"
Int16:
.word 0x0400   ; 16 bits integer = 1024
Int32:
.long 2048     ; 32 bits integer = 2048
My_Struct:
.align 4       ; aligning on an address multiple of 4
.word 0        ; 16 bit word = 0
.word 0,0      ; 2 16 bit word initialized with 0
Text:
.align 4       ; aligning on an address multiple of 4
.byte 0x61, 0x68, 0x3f ; 3 bytes with the string "ah?"
.ascii "The animals"  ; string
.byte 0               ; byte = 0
```

Directives are indications given to the assembler and the linker: they provide information for building the object file. They do not give rise to a translation or to a reservation of memory cells.

```
;----------------------------------------------------
; The symbol "INIT" wil be replaced by 0xbf397df
;----------------------------------------------------
INIT = 0xbf397df
;----------------------------------------------------
; global makes the symbol "start" visible from other files
;----------------------------------------------------
.text
.globl start
start:
...
;----------------------------------------------------
; Calling an external procedure proc_ext. The latter is not
; defined in the current file. The link editor will look
; for it in the object files used to build the .exe file.
;----------------------------------------------------
.extern proc_ext
...
call proc_ext           ; calling proc_ext
...
```

The programmer has other opportunities at their disposal for facilitating the writing of the source file, for example *macroinstructions* and *conditional assembly* instructions.

4.4.2. *The assembly phases*

As we have seen, the assembler has the following tasks: "translating from symbolic representation to binary representation", "allocating memory addresses to the operands" and "memorizing all of the symbols used by the program". This work is done in two steps known as *passes*. The first pass is in charge of translating and initiates the address calculations, which are completed in the second pass.

4.4.2.1. *First pass*

The objective in this phase, in addition to the translation, is to construct the *symbol table*. This table assigns a value to each mnemonic (variable or tag) used by the program. This value is not, immediately, the memory address where the variable or tag will be inserted. The assignment process involves the following steps:

– The assembler initializes the counter to zero.

– This counter is incremented every time a line in the source file is read, and these lines are analyzed one by one.

– For each symbol that is encountered, a new entry, a *cross-reference*, is added to the symbol table. If the symbol is a tag, it is assigned the value of the counter.

Allocating an address to a symbol is done as soon as possible. As long as a memory address is not associated with a variable or a tag, it is represented by the number of the input it was assigned to in the symbol table. At the end of the first pass, the symbol table is completely updated; each mnemonic used by the programmer is now associated with an address.

4.4.2.2. *Second pass*

During this phase, the assembler goes once again through the source file using the symbol table created during the first pass. If the line contains an instruction, the assembler reads in the symbol table the values associated with the variables, completes the binary coding of this instruction and stores this code in the object file. If errors have been detected, the building of a possible text file is continued, but the building of the object file is abandoned and it is destroyed. References that have not been dealt with are processed in this phase: the variables that were represented by their entry in the symbol table are now represented by their addresses.

4.4.3. *The linker*

The linker combines the object files and the libraries to create an executable file. This file can be complemented with additional information, for example information necessary to the use of a *debugger*.

Object files are composed of several sections, for example:

– the program (*text segment* or *code segment*);

– the data initialized by the user (*data segment*).

In order to combine the address fields of the different object modules, the linker builds a table containing the name, size and length of all the object modules, then assigns a loading address to each object file.

It collects the symbol tables so it can build a unique object module equipped with a single symbol table. The addresses of the variables are recalculated. Hidden information (*glue*), often ignored by the programmer, are added for the start of the program, for interfacing with the system (object file `crt0.o` in the UNIX system) and for final modifications.

The linker consults libraries to process references that have not been addressed, meaning those that do not appear in any of the object files cited by the user. The objects found in the library are then added to the object file being built.

The resulting executable file can be quite large. In order to reduce its size, we can replace the object code of a library function by a pointer. This pointer gives the address of an executable module which will be used during the execution of the program. This module will not be integrated into the executable code, and will furthermore be shared by all of the programs that rely on this function. This is known as *dynamic linking* and *shared libraries*. At Microsoft®, the phrase *dynamic linkable library* (DLL) is used. When the function is called, all or part of the library is loaded into memory.

4.4.4. *When to program in assembly language*

The architectures of modern processors are increasingly complex, making it very difficult to program in assembly language. This is particularly true in the case of the *very large instruction word* (VLIW) and *explicitly parallel instruction computing* (EPIC) architectures, for which the developer must know in all its details the architecture of the processor and the relevant constraints.

Assembly programming is only used as a last recourse when faced with strict constraints involving the hardware (real-time, signal and communication processing, embedded applications, etc.). In most cases, high-level language compilers produce sufficiently high performance code.

We can mention situations where assembly language programming, or at least knowledge of this language, is imperative:

– debugging programs when confronted with difficult issues: the use of pointers, stack management, etc.;

– the use of registers or special instructions in the case of system programming: changes of context, low-level input–output processing, writing the system *boot* or the file `crt0.s` in UNIX, etc.;

– the need for performance when the assembly code generated during a compilation does not turn out to be efficient enough.

Even when an assembly language is used as a last recourse, it is often used in the form of *in-line* instructions, i.e. instructions inserted within a program written in high-level language.

Chapter 5

The Processor

We saw in Chapter 2 that, upon examination, the functions performed by a processor can be divided into three units (Figure 5.1):

– the *control unit*, which is responsible for:

- fetching instructions from memory,
- decoding these instructions,
- sequencing microcommands to ensure the operation of recognized instructions.

– the *processing unit*, which produces the results of the elementary arithmetic and logic operations;

– the *registers*, which are used for storing into memory the information handled by the processing unit.

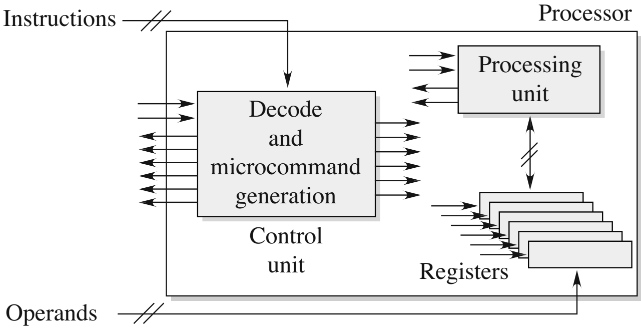


Figure 5.1. *The components of a processor*

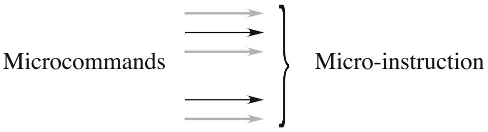


Figure 5.2. *Microcommands and microinstructions*

In the context of microcommands sent by the sequence, we will use the following terms:

– *microinstruction*, a Boolean vector representing the logic state of these microcommands originating from the control unit;

– *micro-operation*, the execution of the microinstruction (Figure 5.2).

After presenting an example of the execution of an instruction, this chapter will describe in detail the components of the control unit, as well as the concepts involved in its design.

5.1. The control bus

We have emphasized the repetitive nature of a processor's tasks: read instruction, decode, execute, read instruction, etc. This cycle can be altered by external signals. These signals are transmitted over the lines of the *control bus*. We have already mentioned the case of the wait line (Figure 5.3).

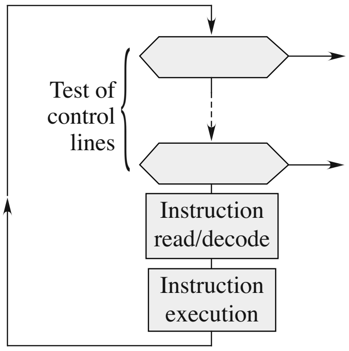


Figure 5.3. *Test of the control lines*

Other signals originating from the processor, indicating its internal state or providing information to external devices (the example of acknowledgments is shown in Figure 5.4), travel on this same bus.

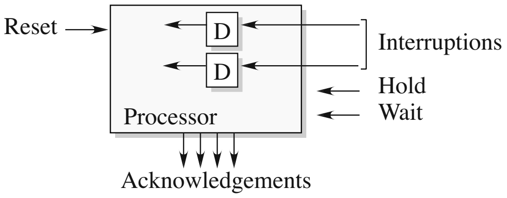


Figure 5.4. *The control lines*

The state of the control lines is usually memorized by flip-flops, which then reflect the event that activated them.

5.1.1. *Reset line*

The *reset* line causes the processor to "reboot". When this line changes state, the various flip-flops and internal registers are initialized, then the booting address is obtained:

– *directly*: the *program counter* is initialized with a given value (this is the case for Intel I8080® microprocessors, where the program counter is reset to zero, and the Intel I8086®, where the *physical* boot address is $FFFF0_{16}$);

– *indirectly*: the *program counter* is loaded with a value that the processor fetches from memory at a fixed address (e.g. in $FFFE_{16}$ and $FFFF_{16}$ in the 8-bit Motorola MC6800 microprocessor, Figure 5.5).

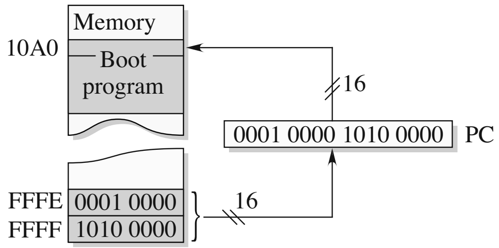


Figure 5.5. *Booting in the case of the Motorola M6800*

5.1.2. *Hold line*

The *hold* line is used when a unit other than the processor (another processor or an exchange unit) attempts to take control of the buses. If this attempt is successful, the processor must abstain from accessing the buses. The protocol uses two lines of the control bus: the *HOLD* line and the *HOLD Acknowledge* line. In response to this hold request, the processor sends an acknowledgment indicating that the data and address buses are switched to the "high impedance" state. The processor is then said to be in *idle mode*.

5.1.3. *Wait control line*

The *wait control* line turns out to be necessary, as we have already seen, when the processor – or another input–output device – attempts to gain access to a memory with a long access time. This *WAIT control line* or *READY line* allows the processor to insert *wait cycles* or *wait states* during read or write operations, while waiting for the required data to become available, or the input–output unit to be ready.

5.1.4. *Interrupt lines*

Interrupt lines are used for taking external events into account. A sequence of instructions can be executed in response to this request. The processing of these events known as *interrupts* will be discussed in Chapter 6 on input–outputs.

5.1.5. *Conceptual diagram*

An operation diagram inspired from the Motorola MC6800 microprocessor can be represented as shown in Figure 5.6.

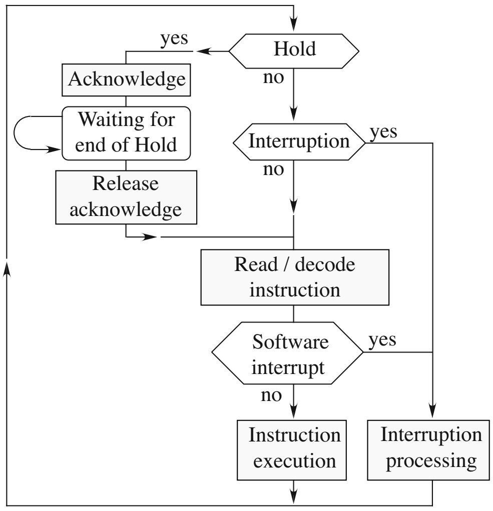


Figure 5.6. *Example of an operation diagram*

A complete phase of executing the instruction is referred to as an *instruction cycle*. The length of this cycle depends on the machine's architecture.

5.2. Execution of an instruction: an example

We are going to illustrate what we have just seen with an example based on the internal operation of the Intel I8080® microprocessor (Figure 5.7) [INT 79]. The instruction we chose is `add M` according to the notation defined by Intel®.

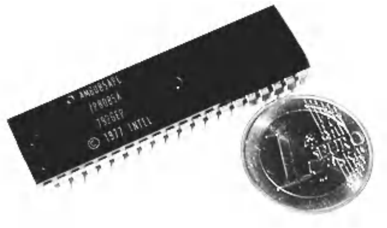


Figure 5.7. *The Intel® I8085, a more modern version of the I8080®, had simpler electronics, in particular a single 5 V power supply and an integrated clock generator, a few additional interrupts and a multiplexed address-data bus*

This instruction performs the sum of the contents of the A register (accumulator) and the memory word with the address provided by the content of the HL register (used as an address register). The result is stored in register A (Figure 5.8).

This instruction gives rise to two memory accesses: one to read the instruction and the other to read the operand into memory.

This example was chosen because the operation of the Intel I8080® microprocessor is very simple, and because a description of every step in every instruction can be found in the documentation that was made available by Intel Corp.

`add M` is divided into three machine cycles: M1, M2 and M3; each cycle divided into "states" T_1, T_2, etc, according to the following scheme:

		M1				
		T1	T2	T3	T4	T5
`add M`	Code 86_{16}	PC OUT Status	PC++	INST → IR	(A) → ACT	☒

⟶

⟶

M2			M3		
T1	T2	T3	T1	T2	T3
HL OUT Status	Data	→ TMP		(ACT)+(TMP) → A	☒

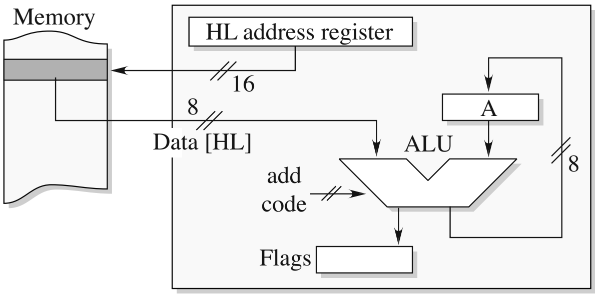


Figure 5.8. *Diagram showing the operation of the* `add M` *instruction*

The simplified internal structure of the machine is shown in Figure 5.9.

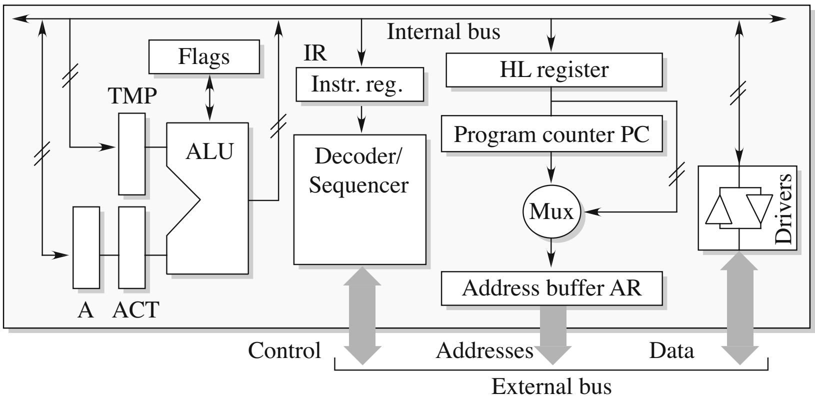


Figure 5.9. *Simplified structure of the CPU*

The figure shows the main registers: instruction register IR (8 bits), flags (8 bits), program counter (16 bits), register HL (16 bits), register A (8 bits) and two buffer registers referred to as ACT (8 bits) and TMP (8 bits). The address buffer AR (16 bits) temporarily memorizes the address that needs to be presented to the address bus. Bi-directional *drivers* are used to supply power to the data bus. These are *three state* drivers.

5.2.1. *Execution of the instruction*

Intel® defined a sequencing in terms of *machine cycles*. This means that each instruction is divided into main cycles (one to five). The sum instruction considered here consists of three machine cycles. Each of these consists of two to five *elementary cycles* ("states") defined by the basic clock. The microcommand involved in the execution of each elementary cycle is denoted by μ^i_{jk}, where j is the machine cycle number, k is the elementary cycle number and i is the line number.

1) *First machine cycle*: This corresponds to *fetching* the instruction in memory and consists of following four elementary cycles:

i) The content of the program counter PC is transferred to the address buffer AR. At the same time, the data bus provides an indication about the current machine cycle, in this case the instruction fetchs cycle. This phase, denoted by M1–T1 (Figure 5.10), involves the following microcode:

- μ^1_{11} commands multiplexer Mux to divert the content of PC to AR;

- μ^2_{11} commands the loading of buffer AR;

- μ^3_{11} indicates that we are in cycle T1. This information appears on a line of the control bus under the name SYNC and can be used by the external logic;

- μ^4_{11} commands the data bus *drivers* in the "send" direction, etc.

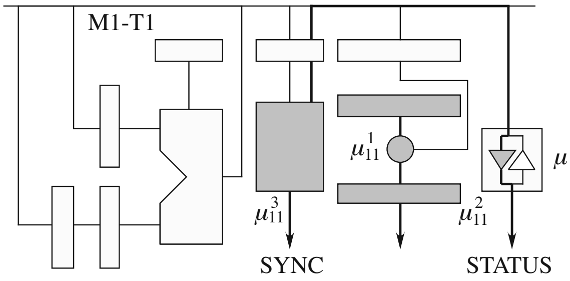


Figure 5.10. *Phase M1–T1*

These are certainly not all of the microcommands, but this list gives a good picture of the role signals play in ensuring the execution of an instruction.

ii) Phase M1–T2 consists of incrementing the content of the program counter. This anticipation of the program counter increment saves time when dealing with instructions that take up a single word and provides additional time for the memory to set itself (see next phase). Among the microcommands, we will mention μ^1_{12} that indicates the program counter "increment mode", μ^2_{12} that activates the incrementation and μ^3_{12} or $\overline{\mathrm{RD}}$ that validates access to the memory (Figure 5.11).

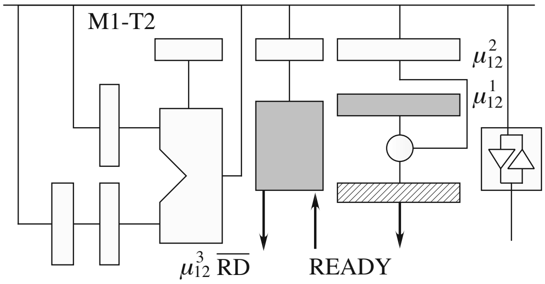


Figure 5.11. *Phase M1–T2*

During this phase, the READY line is tested. If it is set to 0, which means that the memory is not ready to deliver the instruction code, the processor introduces elementary wait cycles TW. This is the same phase where the hold lines are tested.

iii) In the wait phase M1–TW, the instruction expected on the data bus is not considered stable. It can therefore not be entered into the instruction register (Figure 5.12).

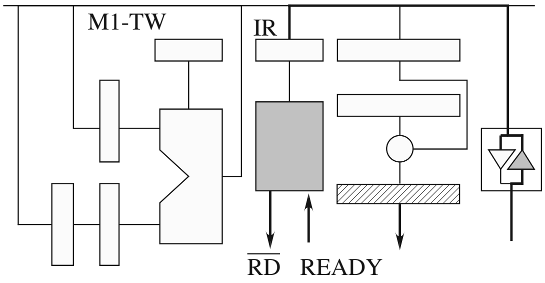


Figure 5.12. *Phase M1–TW*

iv) Phase M1–T3 (Figure 5.13) is the actual fetching of the instruction, which is loaded into the instruction register IR. Among the relevant microcommands, we can mention:

- μ_{13}^1 command for the data bus *drivers*;
- $\overline{\text{RD}}$ memory control signal;
- μ_{13}^2 loads the instruction register IR.

The memory word address containing the instruction has been on the bus since phase M1–T1.

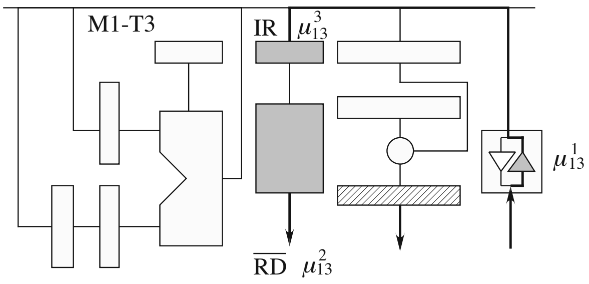


Figure 5.13. *Phase M1–T3*

v) Phase M1–T4 marks the beginning of the actual unfolding of the instruction. It corresponds to the transfer of the content of register A to buffer ACT, a transfer that, as we explained previously, plays a role in preventing operational mishaps (see Figure 2.10). The microcommand μ_{14}^{1} triggers the loading of ACT (the summing code is already placed on the arithmetic logic unit) (Figure 5.14).

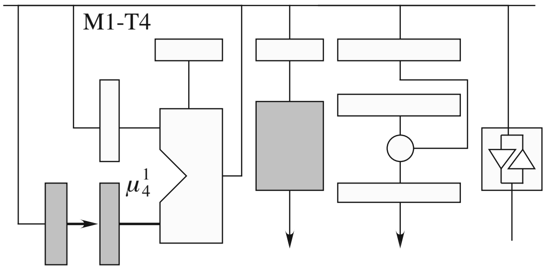


Figure 5.14. *Phase M1–T4*

2) *Second machine cycle*: This cycle is made up of three elementary cycles and corresponds to the machine cycle for reading the operand in memory.

i) Phase M2–T1 (Figure 5.15) consists of indicating the cycle type, *Memory Read*, and is identical in every way to phase M1–T1 (*Instruction Fetch*).

The address of the operand in memory, which is contained in HL, is stored in buffer AR.

ii) During phase M2–T2 (Figure 5.16), the READY line is tested and wait cycles are inserted if necessary.

iii) The wait phase M2-TW is identical in every way to Phase M1–TW.

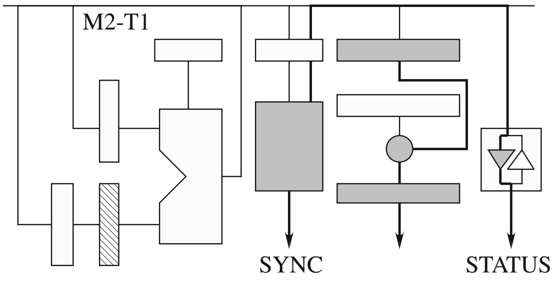


Figure 5.15. *Phase M2–T1*

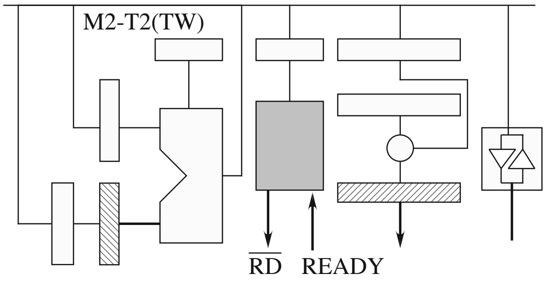


Figure 5.16. *Phase M2–T2*

iv) Phase M2–T3 (Figure 5.17) consists of fetching the second operand with the microcommands:

- μ^1_{23} command for the data bus driver;
- $\overline{\text{RD}}$ controls signal for the memory;
- μ^2_{23} for loading TMP.

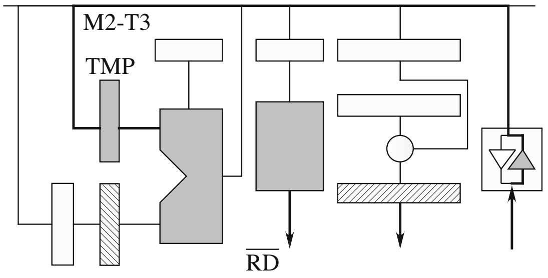


Figure 5.17. *Phase M2–T3*

The address of the memory word containing the operand has been on the address bus since phase M2–T1. Just as in the read operation, a read command is generated for the memory and the data bus driver is set. The only difference is the loading of the registers: TMP replaces IR.

3) *Third machine cycle*: This could be described as the instruction execution cycle. It consists of the following two elementary cycles:

i) During the first-to-last phase, M3–T1 (Figure 5.18), nothing happens for the current operation. It is introduced artificially in order for the following instruction to begin its cycle.

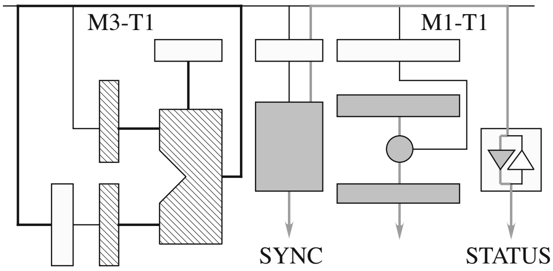


Figure 5.18. *Phase M3–T1 (M1–T1 of the next instruction in the program)*

ii) The final phase, M3–T2 (Figure 5.19), consists of obtaining the result of the summing operation from register A and setting the flags. Among the microcommands used, we can mention:

- μ^1_{32} for the loading of register A;
- μ^2_{32} for the loading of the flag register.

The overlap between consecutive operations significantly improves performance. For the summing instruction, we gain two elementary cycles out of the nine that are required (once if we include the wait cycles).

5.2.2. *Timing diagram*

We continue with the commands involved in the execution of the instruction studied in the previous section, and now represent them in a *timing diagram* depicted in Figure 5.20 This timing diagram is based on the I8080 and I8085 processors from Intel Corp.® processors. The $\overline{\text{RD}}$ line was actually introduced with the Intel I8085®. In the Intel I8080®, a DBIN (*Data Bus IN*) line was used for validating read transfers.

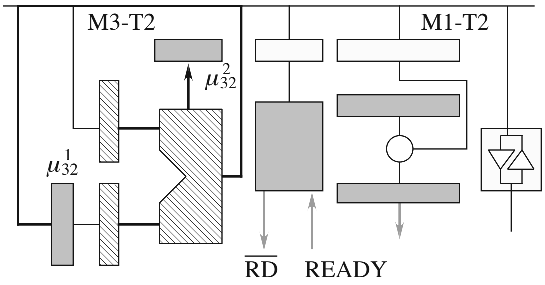


Figure 5.19. *Phase M3–T2 (M1–T2 of the next instruction in the program)*

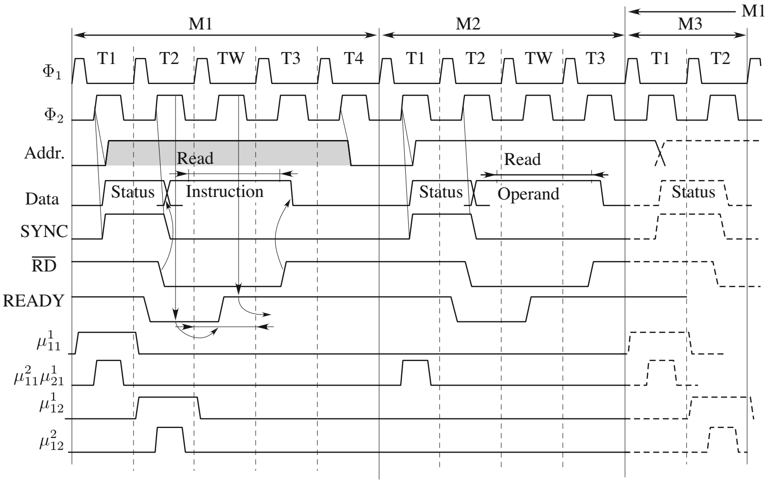


Figure 5.20. *Timing diagram*

Time is represented on the x-axis and the logic state of a few of the commands are represented on the y-axis. We want to take a closer look, in particular, at phases M1–T1 through M1–T3, which correspond to the fetching of the instruction.

1) During the M1–T1 phase, the microcommands involved are the following:

- μ_{11}^{1} command for multiplexer Mux which is assumed to be equal to 1 in M1–T1. or ,

- μ_{11}^{2} command to load buffer AR equal to Φ_2.

These commands are also used in fetching the operand during the phase where AR is loaded with the content of HL. We assume that $\mu_{11}^{1} = 0$.

2) During phase M1–T2, the address of the instruction (shaded area) is not yet *valid* on the address bus. To increment the program counter, the *increment* mode μ_{12}^{1}

is set to 1, then an impulse from the clock μ_{12}^{2} using Φ_2 is generated;

3) And so on.

5.3. Sequencer composition

The sequencer (Figure 5.21) is an automaton that generates the *microcommands*. This automaton, or *sequential* machine, has been taken as inputs:

– the instruction code or, if we make a distinction between *decoder* and *sequencer*, the outputs of the decoder;

– information on the state of the processor (in particular, on the flags);

– the state of the control bus lines;

– the signals from the clock.

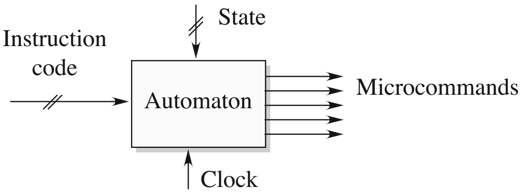


Figure 5.21. *The sequencer*

The design of this sequential machine can be achieved by:

– a traditional synthesis method based on hardwired automatons;

– a method based on the concept of microprogramming;

– a combination of these two methods.

5.3.1. *Traditional synthesis methods*

5.3.1.1. *Main ideas*

Traditional synthesis methods depend a great deal on the experience of the designer. They consist of coding the different phases of the execution of the instructions. To illustrate, we will consider again the instruction used as an example, simplifying even more the operation of the processor. Let C_n be the variable that uniquely identifies the instruction and ϕ_{ij} the variable that canonically codes phase M_i–T_j (Figure 5.22). It is of course out of the question to do a synthesis of all the instructions independently from one another. The task would be too difficult.

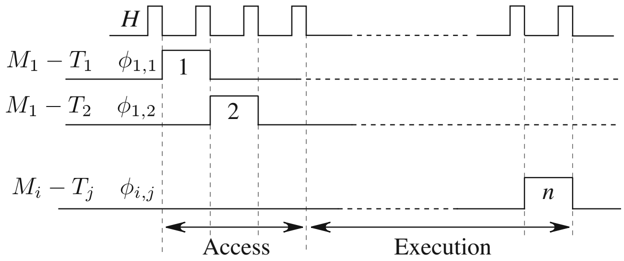


Figure 5.22. *The ϕ_{ij} variables*

The equation for the command of multiplex "PC/HL", for this instruction *alone*, can be written as:

$$\mu_{12}^2 = C_n \times \phi_{11} = \phi_{11}$$

because μ_{12}^2 is independent of C_n, since that phase executes itself regardless of the instruction. Noticing this *might* lead to a simplification (the simplest expression does not always correspond to the simplest design). The load command for register AR could be:

$$loadAR = \phi_{11} \times \Phi_2 + C_n \times \phi_{21} \times \Phi_2$$

The incrementation command for the *program counter* will be written as $incrPC = \phi_{12}$ due to the independence of *incrPC* with respect to the instruction code.

If we know how to write all of these commands as equations, our synthesis problem boils down to generating the ϕ_{ij}. We can then draw the *state diagram* of the corresponding sequential machine (Figure 5.23).

The functions that set the conditions for transitions in this graph depend on the instruction and the condition codes. Condition C_1, for example, will only be true for an "empty" instruction (instruction that does nothing or `nop`), whereas its complement will be true for all of the other instructions. Such an instruction would generate a sequence ϕ_{11}–ϕ_{12}–ϕ_{13}. The sequence "state 1 – state 2 – state 3" is incompressible and common to all of the instructions.

The synthesis process relies on models of variable complexity arrive at a coding of the states, followed by the logic equations for the state variables and the outputs. The diagram showing the operation of the resulting sequencer is shown in Figure 5.24.

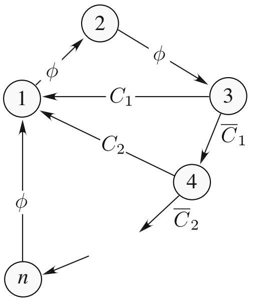


Figure 5.23. *The state diagram of the $\phi_{i,j}$ generator*

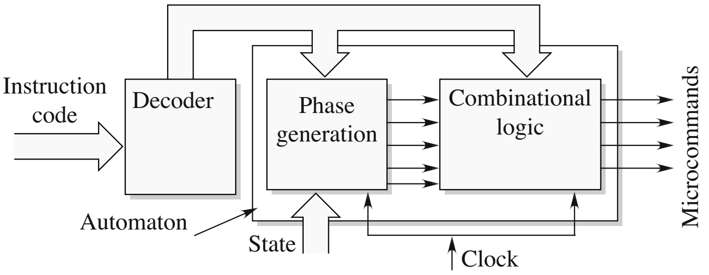


Figure 5.24. *Diagram showing the operation of the hardwired sequencer*

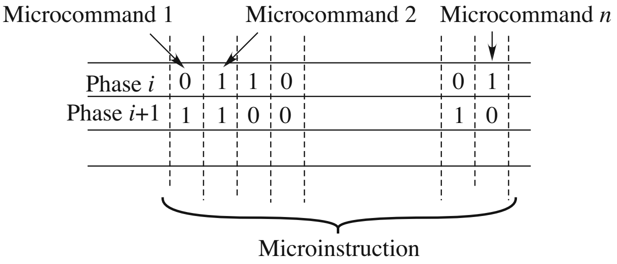


Figure 5.25. *Microcommands*

The main difficulties of this approach arise when dealing with the design (each modification requires doing the entire synthesis process over again), the optimization of the result (minimizing the number of elements) as well as the testing of this result.

5.3.2. *Microprogramming*

The second sequencer synthesis method consists of using a memory device – the *microprogram memory* – to implement the sequencer. For each phase, we can identify the microcommands involved in its design (Figure 5.25). These values are stored onto memory, and the command can be executed simply by scanning this memory.

This memory consists of:

– a *microcommand* part: at a given time, the commands can be represented by a Boolean vector called a *microinstruction*. Each binary element of a word in memory indicates the state of one of the microcommands in the *phase*;

– an *address*: the sequencer is a sequential machine. Its states are defined by the inputs of this microprogram memory [WIL 53]. The transitions are defined by the addresses corresponding to the following state.

This technique was proposed by *Wilkes* in 1951 [WIL 51]. At a given time, we have at our disposal (Figure 5.26):

– commands corresponding to the current phase;

– the address of the word containing the commands associated with the following state.

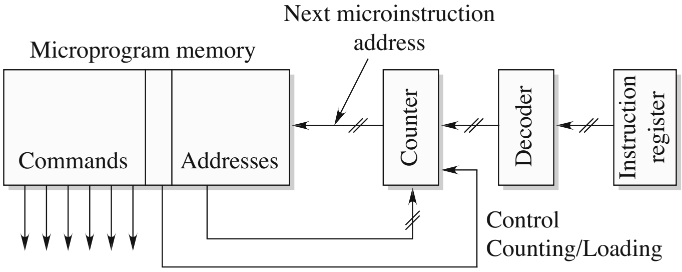


Figure 5.26. *General structure of the microprogrammed sequencer*

The *microcounter* is comparable to a program counter. It provides addresses in the microprogram memory, ensuring the sequential operation of the phases of the instruction. This *microprogramming* technique gives considerable flexibility in defining the set of instructions. In particular, it gives the user the possibility to create new ones if the set provided by the manufacturer is not sufficient. In that case, of course, the memory comprises of a non-volatile part defining the standard set of instructions, and a volatile part usable for *microcoding* new instructions.

5.4. Extensions

5.4.1. *Coprocessors*

Coprocessors provide extensions to the capabilities of the processor and are equipped, just as regular processors are, with a control unit and a processing unit. The processor and coprocessor share the same local bus and have common instructions, referred to as *escape instructions*, which extend the basic set of instructions.

EXAMPLE 5.1.– [I80387 coprocessor] Coprocessors from a series such as Intel® I80387 (*NPX, Numeric Processor Extension*), Cyrix and Weitek generate a significant gain in computational power. These coprocessors are equipped with a particularly wide variety of instructions – arithmetic functions, transcendental functions, trigonometric functions, etc. – and with eight additional 80-bit registers directly accessible to the processor. Exchanges between processor and coprocessor are completely invisible to the programmer.

Registers, which can be accessed through a stack mechanism, are referred to as ST for the top of the stack, ST(1), . . ., ST(7) (Figure 5.27).

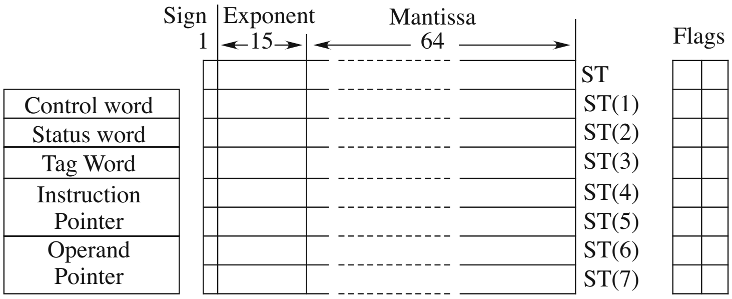


Figure 5.27. *Architecture of the coprocessor*

The instruction size is 2 bytes when the memory access is not required. Otherwise, the size is 4 bytes for the Intel® 8087, the Intel® 80287 and the Intel® 80387 in 16-bit mode, and six bytes for the Intel® 80387 in 32-bit mode. On the original Intel® 8087 processor, each instruction was preceded with an FWAIT instruction to ensure the synchronization of the execution of processor and coprocessor instructions.

Each coprocessor instruction is preceded by an escape byte “ESC”. The processor and the coprocessor decode the instruction. The processor calculates the address of the operand and fetches it. The data are then “captured” by the coprocessor on the data bus.

Internally, the coprocessor works in an extended 80-bit floating-point representation. Of these 80 bits, 64 are used for the significand, 15 for the exponent and 1 is a sign bit. What makes this coding different is that the first bit of the significand (the *hidden* bit in other floating-point representations) is apparent. The formats that are handled, and that are therefore subject to conversion when they are loaded into the coprocessor, are the following:

– 16, 32 or 64-bit integers coded in two's complement;

– 18-digit *packed BCD*. Bit 79 is the sign bit, and bits 72 through 78 are not used;

– single precision 32-bit floating-point: significand (23), exponent (8) and sign;

– double precision 64-bit floating-point: significand (52), exponent (11) and sign.

The two bit flags associated with each internal register, and combined in a 16-bit word, give the state of the content of each register: 00 for valid content, 01 for content equal to 0, 10 for QNaN, SNaN (page 45), infinite, denormalized and 11 for empty.

The presence of these flags makes it possible to optimize the operation of the coprocessor, in part by avoiding unnecessary calculations. The state word has the following structure:

	B	C3	T2	T1	T0	C2	C1	C0	···
···	ES	SF	PE	UE	OE	ZE	DE	IE	

Bits T0, T1 and T2 give the number of the register at the top of the stack, and bits C0 through C3 are the coprocessor's condition codes. The exception flags (see Chapter 6 on input–outputs) are: PE (precision), UE (underflow), OE (overflow), ZE (division by zero), DE (denormalized operand) and IE (invalid operation).

Interrupts 7, 9, 13 and 16 correspond to the coprocessor exceptions on the Intel® 80386.

EXAMPLE 5.2.– [Calculating a sum using the coprocessor] We wish to calculate the sum of two single precision floating-point numbers using the floating-point coprocessor described in the previous section.

The calling program, written in C:

```
#include <stdio.h>

int main () {
    void mysum(float, float, float*);
    float a=3.5, b=7;
    float c;
    mysum(a, b, &c);
    printf("%5.2f\n",c);
    return 1;
}
```

transmits to the stack address of the result (32 bits), then the value of operand b (32 bits) and finally the value of operand a (32 bits).

The assembly language program is as follows:

```
     .text
.globl _mysum
_mysum:
     pushl     %ebp
     movl      %esp, %ebp
     pushl     %ebx
     flds      12(%ebp)
     fadds     8(%ebp)
     movl      16(%ebp), %ebx
     fstps     (%ebx)
     popl      %ebx
     leave
     ret
```

The `flds` instruction loads a single precision floating-point number onto the top of the stack (register ST). `fadds` adds the top of the stack to the single precision floating-point number with the address given in the stack in `8(%ebp)`, and the result is stored in ST. `fstps` stores the top of the stack at the address provided (`%ebx`). The result for the executable named `addfloat` is:

```
mycomputer:$ addfloat
10.50
```

5.4.2. *Vector extensions*

Most programs apply identical processes to large quantities of data. The parallelization of these processes considerably improves processing speeds.

Processor architects have focused on developing modules known as *vector extensions*. Of these, we can mention Altivec by Motorola®, 3DNow! by AMD® and, at Intel®, the MMX technology (Pentium® II), SSE extensions (Pentium® III), SSE2 extensions (Pentium® 4, Xeon®), SSE3 extensions (Pentium® 4 with Hyper-Threading) and Supplemental Streaming SIMD Extensions 3 (Xeon® 5100 and Core® 2).

In the most recent extensions, the data in a 128-bit register can be interpreted as 128 bits, 16 bytes, eight 16-bit integers, four 32-bit integers or four IEEE-754 single precision floating-point numbers.

5.4.2.1. *The MMX extension*

The first generation of extensions (the MMX extensions) from Intel® used the registers of the floating-point coprocessor for the vector extension (Figure 5.28).

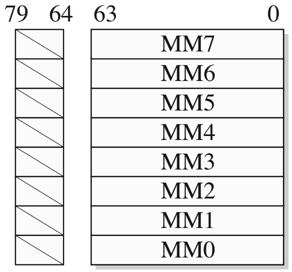


Figure 5.28. *The MMX extension*

"Quadruple word" registers (4×16 bits) are denoted by MM0, MM1, ..., MM7. When the operations deal, in parallel, with a set of bytes, words, or double words of a 64-bit register, we use the term *packed data*.

The set of instructions associated with the MMX consists of 57 instructions: transfers, arithmetic and logic operations, comparisons, conversions, shifts and setting to zero the MMX state register. Saturations are dealt with in "saturated mode", and sums are handled "bit-by-bit", so $7F_{16} + 1$ yields $7F_{16}$ and not 80_{16}.

EXAMPLE 5.3.– [MMX instructions] The instruction set offers several different types of sums, in particular:

– sum involving packed bytes with handling of the saturation (the “fixed-point” section on page 40): `paddsb mm0,mm1` simultaneously adds 8 bytes contained in `mm0` to those in `mm1` and stores the eight results in `mm0`;

– sum involving packed words without handling of the saturation: `paddw mm0,mm1` simultaneously adds four words contained in `mm0` to those in `mm1` and stores the eight results in `mm0`.

EXAMPLE 5.4.– [Multiplication–accumulation instructions] A multiplication–addition operation involving words facilitates the processing of operations such as the fixed-point convolution (section 5.4.3.1 on DSP). Its operation is summed up in Figure 5.29.

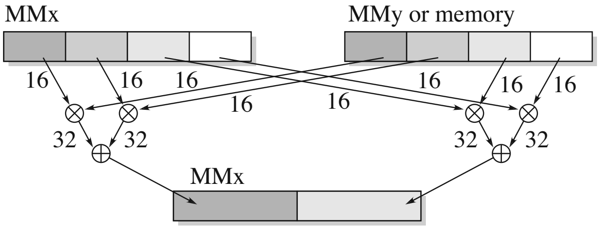


Figure 5.29. *The multiplication–addition operation*

Four 16-bit values $a_1, \ldots, a_4$ are stored in the MMx register, four 16-bit values b_1, $\ldots$, b_4 are stored in the MMy register. The `pmadwd` instruction yields $\sum_{k=1}^{2} a_k b_k$ in the 32 most significant bits of MMx and $\sum_{k=3}^{4} a_k b_k$ in the 32 least significant bits.

With a sum and a shift, we can accelerate the sum of four products.

The management of registers is not the same for STn floating-point and MMn registers. With the former, STn refers to the physical register with the number between 0 and 7 given by the top of stack (TOS) field of the state register. MM0 always refers to the physical register 0. At each execution of an MMX instruction, the TOS field is in fact reset to 0.

The `emms` (*Empty MMX Technology State*) instruction sets all the flags of the floating-point unit to “empty”. When working with the MMX, every sequence must end with this instruction, otherwise the data contained in the register could seem valid to the floating-point coprocessor.

During a task switch, the `CR0.TS` bit of the control register `CR0` is set to one. The first attempt to execute an MMX or floating-point instruction triggers a seven exception (*device not available*). This allows the system to save a copy of the task and restore it (FSAVE and FRSTOR instructions).

5.4.2.2. *SSE extensions*

The MMX extension has the drawback of only processing integer values (fixed-point numbers). Furthermore, since it is impossible to simultaneoulsy use the floating-point coprocessor and the vector extension, Intel® added a specific set of 128-bit registers (Figure 5.30), the SSE extension (*Streaming SIMD Extension*).

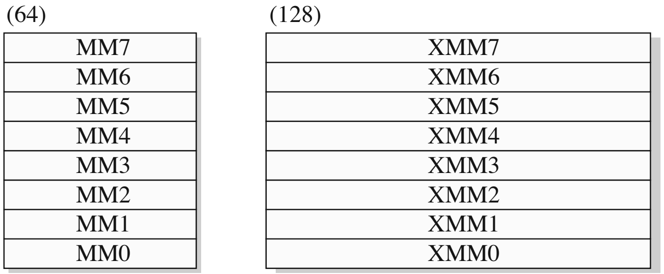


Figure 5.30. *The SSE extension*

The data being processed can be stored in memory or in one of the eight 128-bit registers of the extension, referred to as the XMM registers. At the same time, the SSE extension introduces a certain number of MMX instructions.

Intel® introduced, with the Pentium®-4, the second generation SSE2 in which the operand can be, as before, stored in memory or in XMM registers. It is possible to work with double precision floating-point data, with 64-bit packed integers, and with 128-bit integers.

EXAMPLE 5.5.– [Dot product on an SSE3 extension] Consider two four-element vectors v_1 (vect1) and v_2 (vect2). Their dot product is given by $S = \sum_{k=0}^{3} v_1(k)v_2(k)$.

```
#include <stdio.h>

int main () {
    void dotprod(float*, float*, float*);
    float vect1[]={1.0, 2.5, 3.0, 4.5};
    float vect2[]={5.3, 2.5, 2.1, -1.5};
    float result;
    dotprod(vect1, vect2, &result);
    printf("%f\n", result);
    return 1;
}
```

The registers xmm0 and xmm1 are used for storing the two vectors. The multiplication mulps performs a dot product by calculating the four products $p(k) = v_1(k)v_2(k)$. The horizontal addition haddps on the same register xmm0 first gives us $[p(0) + p(1), p(2) + p(3), p(0) + p(1), p(2) + p(3)]$ then $[S, S, S, S]$, in the second call, where S is the result of $p(0) + p(1) + p(2) + p(3)$.

The assembly language program is as follows:

```
     .text
.globl _dotprod
_dotprod:
     pushl     %ebp
     movl      %esp, %ebp
     pushl     %ebx
     movl      8(%ebp),%ebx
     movaps    (%ebx),%xmm0
     movl      12(%ebp),%ebx
     movaps    (%ebx),%xmm1
     mulps     %xmm1,%xmm0
     haddps    %xmm0,%xmm0
     haddps    %xmm0,%xmm0

     movl      16(%ebp), %ebx
     movss     %xmm0,(%ebx)
     popl      %ebx
     leave
     ret
```

We can test whether the extensions MMX, SSE, etc., are present using the cpuid instruction [INT 09]. The following program calls cpuid from machine instructions included in the program written in C. We are using here what is known as an *extended* assembler (*extended asm*) specific to the gcc compiler:

```
#include <stdio.h>

int main() {
    int b=0, a=0;
    char myform[]="edx=%x and ecx=%x \n";
    __asm__ ("mov $1, %%eax; "       /* 1 into eax */
            "cpuid; "
            "mov %%edx, %0; "
            "mov %%ecx, %1;"
            : "=r"(b), "=r"(a)       /* output */
            :                        /* no input */
            : "%eax", "%edx", "%edx" /* clobbered registers */
            );
        printf(myform,b,a);
    return 1;
}
```

The program returns, for the processor used in this example, the values 8e3bd and bfebfbff in the 32-bit registers ecx and edx, respectively. The interpretation of this result indicates that the processor is equipped with a floating-point unit, or FPU (edx.0), supports MMX instructions (edx.23), as well as SSE (edx.25), SSE2 (edx.26), instructions for saving and rapid restore of the FPU task (edx.24), and supports SSE3 instructions (ecx.0), SSSE3 (ecx.9) and SSE4.1 (ecx.19).

5.4.2.3. *The Altivec extension*

The Altivec® (the Motorola® name), or Velocity Engine® (Apple®), or VMX® (IBM®), is an extension found in the Power 6®, XBox 360®, PowerPC® or, in a similar format, in Sony's PS3®. Its programming model is shown in Figure 5.31.

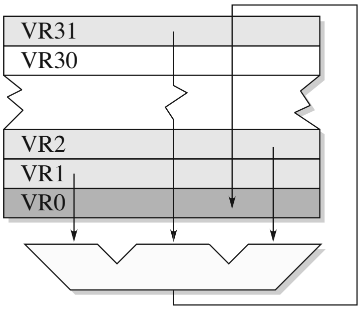


Figure 5.31. *The Altivec® – three sources and one target*

128-bit vectors can be manipulated as sixteen 8-bit integers, eight 16-bit integers (signed or not), four 32-bit integers or four single precision floating-point numbers (no double precision). The 162 instructions support three sources and one target (except for transfers). The Altivec® specification does not indicate either the operating mode (pipeline, parallel, etc.) or the number of processing units.

The Altivec® architecture supports "big-endian" and "little-endian" modes. In the big-endian mode, the transfer from memory to register of a quadruple word writes the word with the address EA in position 0 in the register (Figure 5.32).

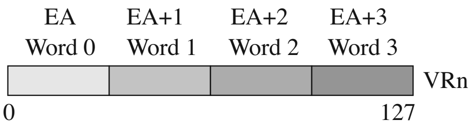


Figure 5.32. *Accessing a quadruple word*

5.4.3. *DSP and GPU*

Standard processors are not the best suited for certain specific tasks such as signal processing (filtering, equalization, etc.) or processing geometric objects to display them on a screen (lighting, texture, etc.).

5.4.3.1. *Processors specialized in signal processing*

Digital signal processors (DSPs) have an architecture adapted to, among other things, *filtering*, or *convolution*, operations. A convolution, requires the ability to execute a multiplication and a sum in a single clock cycle. Another important task, particularly in telecommunications, is *equalization*, which requires the implementation of a "shortest path"-type algorithm, therefore the ability, in one cycle, to perform a comparison and tag a node in a graph (Viterbi algorithm). For these specific operations, DSP are at least as powerful as ordinary processors, despite the relatively low clock rate imposed by low consumption constraints.

In simple terms, the filtering operations are equivalent to calculating, from the sequences $\{x_n\}$ and $\{h_n\}$, a third sequence of values $\{y_n\}$ as follows:

$$y_n = \sum_{k=0}^{N-1} h_k x_{n-k}$$

We therefore need to perform a loop $y := y + (h_k \times x_{n-k})$. If we want to perform each step of the loop in one cycle, we need a special instruction that simultaneously performs a multiplication and a sum (MAC instruction), and most of all we need an architecture that allows for three fetch operations in the same cycle: fetching the instruction, and fetching the two operands h_k and x_{n-k}. This can only be done with the help of a cache (Chapter 8) and two memory banks (instructions and data: Harvard architecture).

DSPs are extremely widespread because they are built into all cell phones. Certain architectures dedicated to stationary communication hardware (base stations) have VLIW (Very Long Instruction Word) architectures (Chapter 13) and hold considerable computational power.

5.4.3.2. *Processors specialized in image processing*

Graphics processing units (GPUs) are programmable calculation units adapted to process 3D images. They are characterized by architectures with very parallelized operations. The “geometric” description of a 3D image is transmitted to the display board and the GPU is tasked with calculating the 2D image, a series of pixels, which is displayed on the screen.

The basic geometric element is the *vertex*. A vertex is defined by its *homogeneous* coordinates x, y, z and w. The usual coordinates are then given by x/w, y/w and z/w. In practice, the vertex is not exactly a point in the traditional geometric sense. It has other attributes, such as diffuse and specular color, a vector orthogonal to the surface where the vertex sits, texture coordinates, etc.

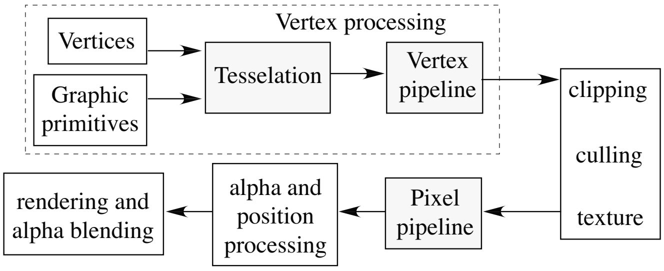


Figure 5.33. *Architecture of a graphics processor. Tessellation consists of dividing each triangle, or basic element, into sub-elements that can then be manipulated along the direction orthogonal to the surface. This helps achieve a more realistic final image.*

The processing (Figure 5.33) is divided into several steps (source: NVIDIA® [LAI 11]):

1) vertex processing (from vertices to screen space);

2) clipping (deleting unseen pixels);

3) pixel generation (triangle setup and rasterization);

4) deleting hidden pixels (occlusion culling);

5) calculating values for all pixels (parameter interpolation);

6) processing of colors, transparency, etc, (pixel shader), textures (in the frame buffer);

7) blending of the final image (pixel engine).

5.5. Exercise

Exercise 5.1 (Instruction sequencing) (Hints on page 330) – In the Intel I8080®, there are 10 possible types of machine cycles: *Instruction Fetch*, *Memory Read*, *Memory Write*, *Stack Read*, *Stack Write*, *Input Read*, *Output Write*, *Interrupt Acknowledge*, *Halt Acknowledge* and *Interrupt Acknowledge while Halt*. The corresponding codes are sent on the data bus during phase T1 of each machine cycle.

Consider the instruction "Transfer immediate value to memory", in which the address is given by the content of the HL register. Its operation is described as follows:

		M1				
		T1	T2	T3	T4	T5
`mvi M,data`	Code 36h	PC OUT Status	PC++	INST →IR	×	⊠

⟶

⟶

M2			M3		
T1	T2	T3	T1	T2	T3
PC OUT Status	PC++ B2 →	TMP	HL OUT Status	TMP→	DATA BUS

Graphically represent the execution of this instruction using diagrams like those in Figure 5.34.

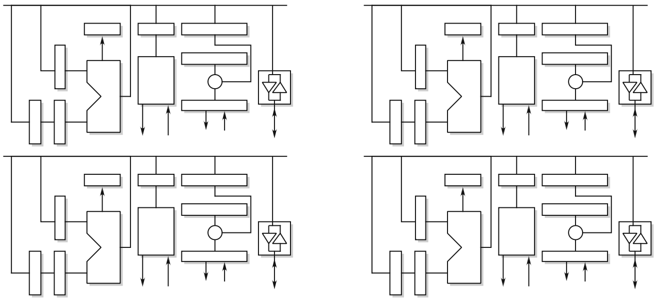


Figure 5.34. *Execution of an instruction, exercise*

Chapter 6

Inputs and Outputs

The use of personal computers has become commonplace, and we are now familiar with its elements: keyboard, screen, printer, mouse, hard drive, etc. These devices, or *peripherals*, are components that are directly tied to the processor. In this chapter, we propose to examine how they communicate with each other.

The few peripherals we mentioned have very different properties, both in terms of functions and electrically:

– The keyboard sends *characters* at a pace defined by the user, and each character is coded and transmitted to the processor.

– The screen is used for displaying text and/or images in color. The transmission of information between processor and monitor can be done in the form of a binary pulse train, a composite signal or a video signal, depending on the characteristics of the screen-graphics card combination.

– The printer is used for producing paper documents at speeds from a few tens of characters per second (*cps*) to several tens or hundreds of pages per minute.

– The drive units contain information stored in the form of blocks that can be transferred at speeds of several million *bytes per second*, etc.

The processor communicates with these peripherals via a *bus* that connects the different modules of the central processing unit (Chapter 2).

The electrical characteristics of the bus prohibit them from being directly connected to peripherals. These peripherals can instead be “attached” to the computer bus via devices called *exchange units*. The “exchange unit-peripheral” connection will be referred to as a *link* (Figure 6.1).

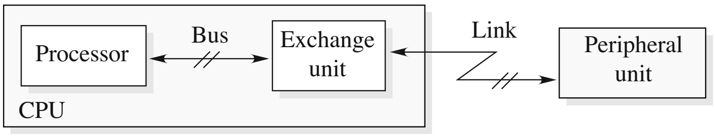


Figure 6.1. *Exchange units*

The simplest exchange units function merely as hardware adapters for electrical signals, but they can also be tasked with managing the entire exchange of information between the processor-memory unit and a peripheral (i.e. for example, the case, with exchange units that manage disks). More specifically, they can be divided into two categories:

1) hardware adapters:

- voltage levels;

- command mode (current or voltage);

- electrical power supply;

- waveshaping (levels, pulses);

2) functional adapters:

- data type (word, byte, block, etc.);

- transfer speed or *bandwidth*;

- protocol handling (combining and recognizing service information, handling all of the exchange procedures).

We will first give two examples emphasizing the role of exchange units in adapting the characteristics of the peripheral to those of the computer (section 1.2). We will then present the basic components of exchange units. Finally, we will describe the elements used for the connection and communication between the exchange units and the processor-memory unit (the dynamic aspect of exchange units), in other words:

– exchange unit addressing;

– handling transfers between the processor or the memory and the exchange units, where we will see that:

– on the one hand, exchange units are often responsible for part of the input–output tasks, so that the processor can devote itself to exchanges with the memory;

– on the other hand, they replace the processor in exchanges that it cannot handle, either from lack of speed (in the case of transfers with a hard drive, for example) or to free up its resources.

This description will emphasize one of the fundamental tools of computer operation: *interrupts*.

6.1. Examples

The first function of exchange units is to *interface* with the peripherals. The role of an interface is to *adapt* the electrical and logic characteristics of the signals sent and received by peripherals to those of signals transported on buses. The first example is an illustration of this function, while the second will show how the exchange unit can also take on part of the processor's tasks.

6.1.1. *Example: controlling a thermocouple*

The system in Figure 6.2 is composed of a processor and two peripheral units:

– an input unit consisting of a device for measuring temperature (a thermocouple equipped with a threshold logic), which supplies a 5 V voltage if the temperature is above a certain threshold, and 0 V if it is not;

– an output unit consisting of only a light-emitting diode that must light up when the temperature goes above said threshold, all this under the control of the computer.

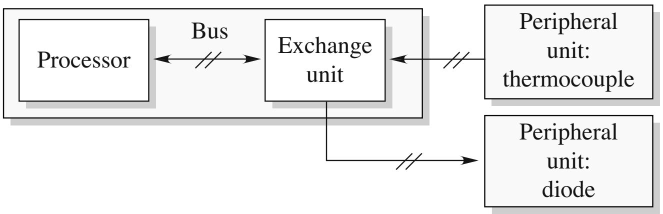


Figure 6.2. *Control device*

To control this device, the processor is equipped with an 8-bit bidirectional data bus and a 16-bit address bus. It also has an accumulator register denoted by A. Let us assume that the only tasks the computer must perform are reading the thermocouple and controlling the on and off switching of the diode according to the flow chart in Figure 6.3.

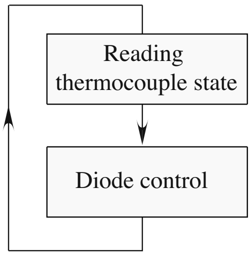


Figure 6.3. *Flow chart of the control system*

When the processor executes a program, the address, data and control buses go through changes in their states over time. The execution of a read or write operation (involving the memory or other devices) is done by presentation:

– *on the data bus*: of the value of the *data* we wish to transfer;

– *on the address bus*: of the *address* of the output or input unit.

Let us assume that the addresses of the input and output units are $140C_{16}$ and $140D_{16}$, respectively. The accumulator contains the data 32_{16}. If the processor executes an instruction to store the accumulator at the address $140C_{16}$, then the states of the data and address *buses*, during a very short time, are as follows:

Data bus								
Line number	7	6	5	4	3	2	1	0
Line state	0	0	1	1	0	0	1	0

Address bus																
Line number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Line state	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	0

A timing diagram of a write instruction is given in Figure 6.4. It shows the processor clock, the data bus, the address bus and the write line of the control bus. This line is used to validate any write operation generated by the execution of a write instruction (in memory or for any other output device).

The addresses are used for selecting the input–output unit. The only difference between the two is the least significant bit (line $A0$):

– input: 140C (hexa) = 0001 0100 0000 1100

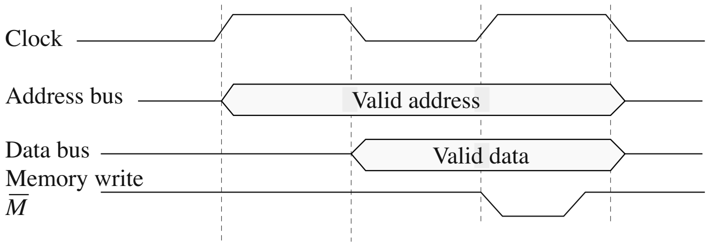


Figure 6.4. *Timing diagram*

– output: 140D (hexa) = 0001 0100 0000 1101

The 15 most significant bits are decoded (Figure 6.5).

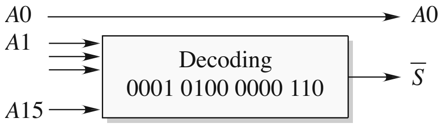


Figure 6.5. *Address decoding*

We will assume that selections are done always by the "logic state" 0 ($\overline{S}$) when the input is in the adequate configuration. This gives us the Karnaugh maps for the input and output selections S_e and S_s, respectively:

		$\overline{S}$	
	S_e	0	1
$A0$	0	0	1
	1	1	1

		$\overline{S}$	
	S_s	0	1
$A0$	0	1	1
	1	0	1

Table 6.1. *Karnaugh maps of the selections*

hence $S_e = \overline{S} + A0$, $S_s = \overline{S} + \overline{A0}$ and the address decoding diagram is shown in Figure 6.6.

The output selection takes place very quickly when the writing instruction is executed.

However, the light-emitting diode must remain on, or off, depending on the state of thermocouple. We will use an element of memory, in this case a D flip-flop, for controlling this lamp. Since the information that has to be stored in memory is only a 1 or a 0, a single bus line will suffice. Let us choose line $D0$. Loading the flip-flop uses the control lines for *memory write* and for *output select*. Note in Figure 6.7 that the load control (the drop in $\overline{M} + \overline{DS}$) happens when the data on the data bus is valid (Figure 6.8).

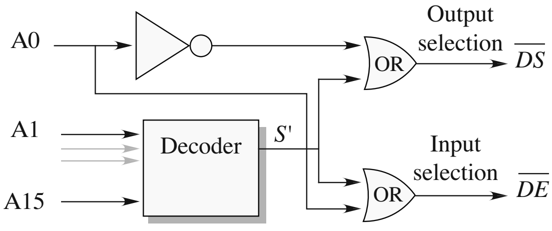


Figure 6.6. *Selection diagram*

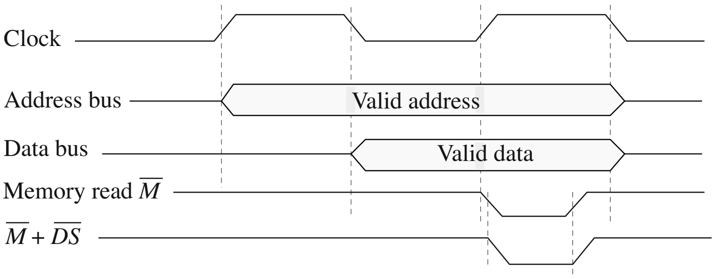


Figure 6.7. *Diode control*

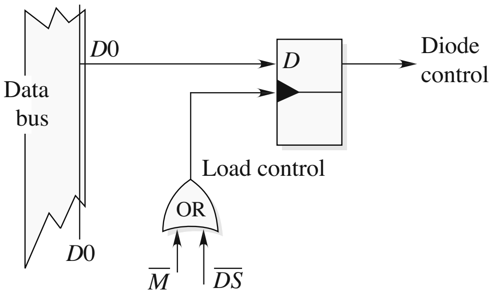


Figure 6.8. *Loading the control flip-flop*

Adapting electrical signals involves both the input and the output. Since the input unit delivers voltages of 0 or 5 V, there are no compatibility issues. We can indeed assume that the microprocessor uses these voltage levels or that there is a voltage adapter designed for such levels. For the output device, the voltage can be adapted by inserting a transistor set up as an *open collector*. This is necessary because if the light-emitting diode were to get its power directly from the microprocessor, or from any logic gate, for example diode that relies on TTL technology, the device could be destroyed. The current required would be too high for its output circuits. It is therefore necessary to introduce an element capable of providing the required amount of energy to power this diode (Figure 6.9).

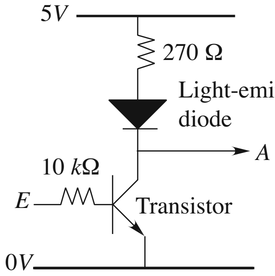


Figure 6.9. *Transistor setup*

We use a transistor – actually an integrated circuit containing a transistor – that functions as a control switch from its input E (the *base* of the transistor). When the input voltage is sufficient (in this case 5 V), the transistor switches to its *active* or *saturation* mode (closed switch). The potential at point A is set to a value close to 0 V. The current through the diode becomes sufficient for it to light up. In the opposite scenario, the transistor is said to be *cutoff* (open switch). The potential at point A is 5 V and no current is traveling through the diode, which remains off.

Therefore, if we wish to turn the diode on (respectively off), we need to load an even value (respectively odd) into the accumulator and send it via the data bus lines. Line $D0$ (least significant bit) will then take on the values 0 or 1.

To light the diode, two instructions are necessary:

– the first, which loads an odd value into the accumulator so that $D0$ is set to 1;

– the second, which transfers the content of the accumulator to the address 140D_{16}. The input unit is selected just like the output unit.

Let us consider the more general case of an input unit and an output unit that are accessible, via a bidirectional data bus and an address bus, according to Figure 6.10.

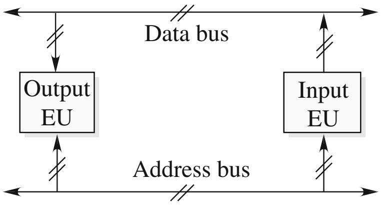


Figure 6.10. *The bus, and the input and output units*

Since each unit is accessible from the same data bus, the input unit runs the risk of deteriorating when, for example, a write operation is performed to the output unit. This is due to the fact that the information will be present on the outputs of the input unit. The solution that was chosen is to use three-state drivers. When the input unit is not selected (addressed), these outputs are set to high impedance. They are "disconnected" from the bus.

In the present case, the circuit diagram (Figure 6.12) will use one of these three-state drivers (Figure 6.11) as a switch.

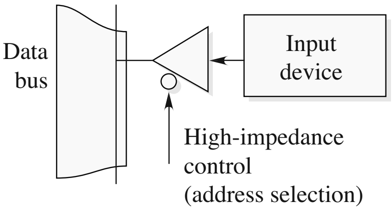


Figure 6.11. *The three-state driver*

Note that the same is true for memory "outputs" set to high impedance when they are not addressed.

The thermocouple state is read by loading into the accumulator the content of the address $140C_{16}$.

```
mbegin:
    mov A,(140CH) ; reading of the thermocouple state. A:=(140CH)
                  ; The bit is 0 if the temperature is less
                  ; than the chosen threshold,
                  ; and otherwise 1
    mov (140DH),A ; diode control (140DH):=A
                  ; (only the 0 bit is active)
    jmp mbegin
```

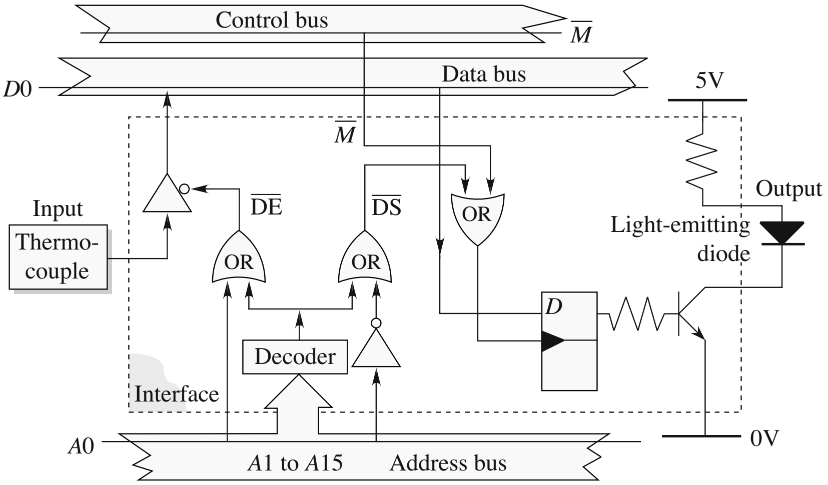


Figure 6.12. *Circuit diagram*

Note the poor use of the processor. A device that only detected changes in the state of the thermocouple, and warned the processor of these changes, could certainly free it up for other tasks. We will come back to this point in the section on interrupts.

6.1.2. *Example: serial terminal connection*

For a very long time, communicating with a computer was done using terminals that operated in text mode. We will focus here on the connection of a *serial-asynchronous* terminal to a computer (Figure 6.13). This peripheral is equipped with a screen and a keyboard, which allow it to display characters sent by the computer or, the other way around, to send the computer characters typed on the keyboard.

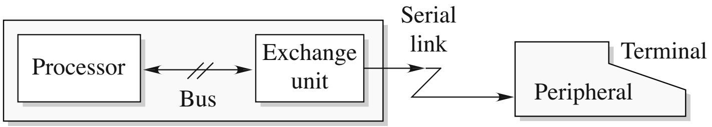


Figure 6.13. *Diagram of the connection scheme*

Suppose we wish to send the character “G” to the terminal, and that it functions with ASCII. The coding for this character is then 47_{16}.

The character can be sent in two steps:

– the character is first "sent" (execution of a program) in parallel mode on the data bus to the exchange unit;

– then transferred in *serial* mode from the exchange unit to the terminal. As a convention, we send the least significant bit first (Figure 6.14). The period of the clock signal is denoted by T.

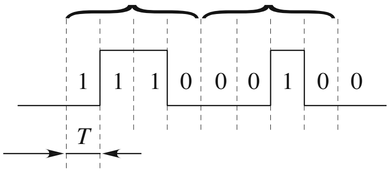


Figure 6.14. *Timing diagram of the transfer*

The terminal will only recognize the character it receives if the following two conditions are met:

– the first transmitted bit can be identified, to ensure what is known as *character synchronization*;

– the signal received is sampled at the proper rate (bandwidth).

The transmission protocol works according to the following rules:

– At rest, meaning when no character is being sent, the line has to be set to a potential such that its state can be distinguished from that of a line not connected to any terminal (logic state 1).

– The first bit is always a 0, which precedes the byte being transmitted. This is known as the *start bit*.

– We add, at the end of the character transmission, a certain number of 1 bits, known as *stop bit(s)*. The number of these stop bits is usually 1, 1.5 or 2 (1.5 means that the line remains set to 1 for a time $1.5 \times T$). This ensures that a transition $1 \to 0$ will indeed occur when the next bit is sent.

If we use two stop bits, the timing diagram will look like the diagram in Figure 6.15.

The exchange unit that handles the terminal is therefore in charge of the following functions:

– serializing the data provided by the processor;

– adding service information (in this case, start and stop bits);

– defining the transmission speed to adapt to the terminal being used.

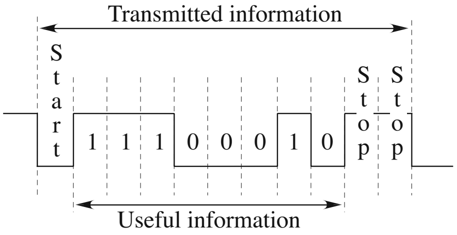


Figure 6.15. *Transmission with two stop bits*

The transmission rate is expressed in *bits per second*, although the term *bauds* is also often used (wrongly, since it actually expresses the modulation rate used). The receptor operates as follows: the controller samples a line at a frequency usually set to $16/T$ (this is one of the parameters that can be provided to the controller). As soon as the controller detects a 0, it waits for a time $T/2$ and samples the line again to check if the switch to 0 is not transient. If this 0 state is stable, it continues to sample the line at a frequency $1/T$ and loads the bits it receives in an offset register (Figure 6.16).

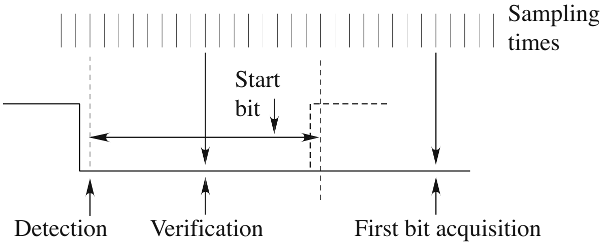


Figure 6.16. *Signal sampling by the controller*

The general scheme of this exchange unit is shown in Figure 6.17.

When a character is typed on the keyboard, the transmission diagram in the direction from terminal to processor is of course similar to the previous diagram (Figure 6.18).

In practice, the two input and output functions are integrated into a single electronic component referred to, depending on the manufacturer, as *Universal Asynchronous Receiver Transmitter* (UART), *Asynchronous Communications Interface Adapter* (ACIA), etc.

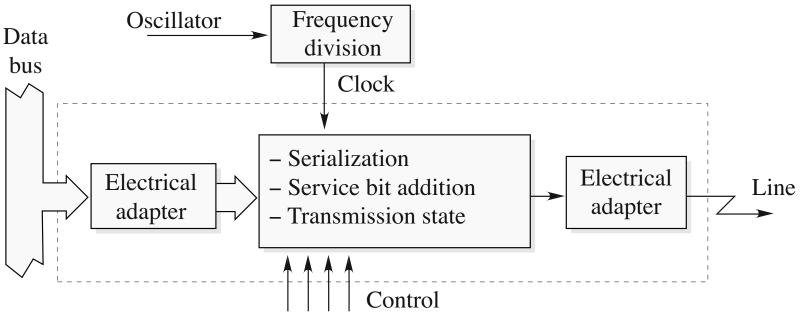


Figure 6.17. *Sending from the exchange unit*

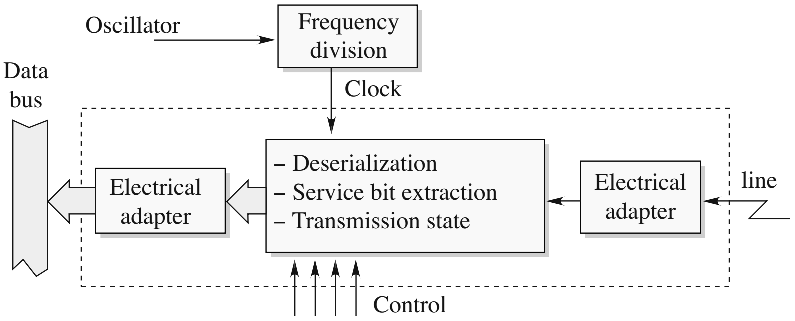


Figure 6.18. *Reception by the exchange unit*

These integrated controllers offer many different features, including:

– choosing the number of useful bits (from five to 16);

– choosing the number of stop bits to transmit;

– checking the parity of the word being transmitted (meaning the parity of the number of 1 bits in the character code, not the parity of the code itself), etc.

These parameters are defined during the execution of the controller's initialization program.

One possible diagram for such an exchange unit is shown in Figure 6.19.

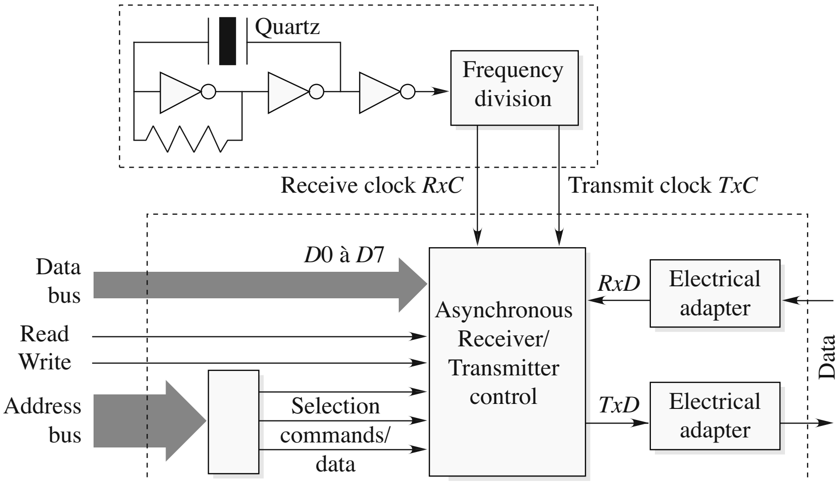


Figure 6.19. *Serial exchange unit*

6.2. Design and addressing of EU

6.2.1. *Design of exchange units*

In the case of serial *reception* discussed as an example, when the exchange unit has finished "de-serializing" a word, it must be able to indicate to the processor that a word is available (the processor can then fetch it). In the case of *sending*, the exchange unit must also be able to inform the processor that it is ready to receive a word before it can be sent.

In both cases, the exchange unit must also be able to indicate whether the exchange took place properly. All of this information can be gathered in a register known as a *status register*, where each bit carries a specific meaning, such as "transmitter ready", "receiver ready" and "parity error".

The exchange unit needs to be equipped with one or several *data registers* used by the processor to read or write the data being exchanged.

Because the exchange unit generally offers several services, each one in response to a command, it will also be equipped with:

– a *command register* used by the processor to read or write the task at hand: initializing the exchange unit, picking and starting a task, etc.;

– a *parameter register* used for providing the task characteristics: transmission rate, memory transfer addresses, number of stop bits, track and sector number on a disk, etc.

All of these components can be represented according to Figure 6.20.

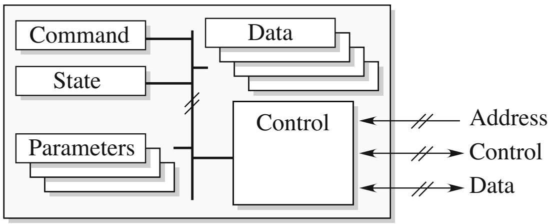


Figure 6.20. *Design of the exchange unit*

An exchange unit can be considered as a specialized processor with an instruction register consisting of a single command register.

The difference lies in the fact that the exchange unit is not usually searching for commands in memory in an autonomous fashion. It receives them under the control of the processor. This is what is known as a *slave unit*.

The complexity of exchange units varies greatly. In our first example (hardware adapting), we were dealing with a simple *interface*. Our second example is called a *serial controller*. *Input–output processors*, on the other hand, are much more complex. Examples of these include the *Redundant Array of Inexpensive Disks* (RAID) disk units, which ensure the duplication, storage and journaling of data.

6.2.2. *Exchange unit addressing*

Like memory cells, the registers of exchange units are identified by their address. These addresses can be thought of as part of the entire set of memory addresses, or as a distinct set. These two cases are known as *single* address space and *separate* address spaces, respectively.

6.2.2.1. *Single address space*

The registers of the exchange unit are handled like memory cells. The processor sends commands, parameters or data to the exchange unit by writing in its registers using write instructions in memory (`store`, `move`, etc.). Similarly, reading data or the status register (`load`) is achieved using read operations in memory (Figure 6.21).

For example, storing the value seven in the data register (address 10) of an exchange unit can be done as follows:

```
...
mov  #10,r0    ; r0 is initialized as 10
mov  #7,@r0    ; the number 7 is written at the address 10
...
```

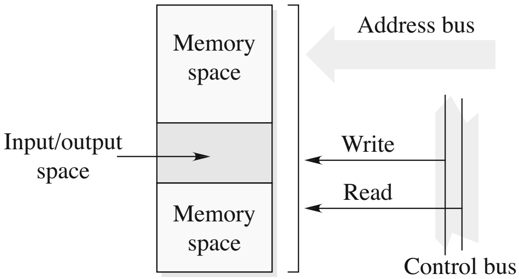


Figure 6.21. *Single address space and inputs–outputs*

To avoid any *access conflict* issues, there can be of course no memory installed at the address of the registers of the exchange units.

6.2.2.2. *Separate address spaces*

In the case of separate address spaces, the machine is given special instructions for inputs and outputs. No change is made to the address and data buses. The distinction lies in the command lines, which are different in the control bus (Figure 6.22).

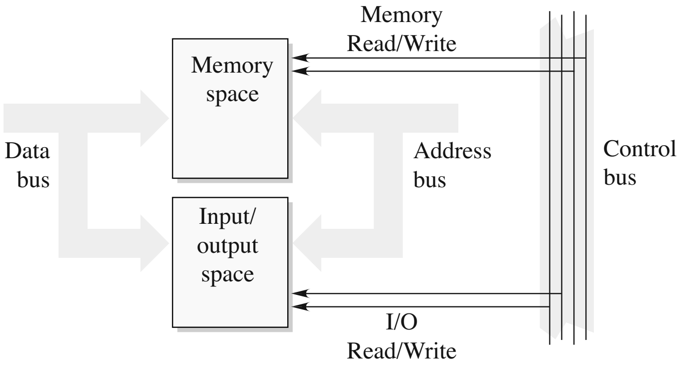


Figure 6.22. *Separate address spaces*

EXAMPLE 6.1.– The first computers with Intel® processors put the two previous concepts into practice:

– exchange unit registers are in a separate memory space accessible via input–output instructions (`in`, `out`, `outsb`, `outsw`, etc.). The physical size of this space is theoretically 64 kB (in PC-type machines, only the 10 least significant bits of the address were decoded);

– on the other hand, the area reserved for the video screen is located in the memory space. In the case of the *monochrome display adapter* (MDA) display mode, this space takes up 4 kB, starting from the physical address $B0000_{16}$ (for a total memory space of 1 MB). A character is displayed on the screen simply by writing in the memory space attributed to it. The video controller can read the content of this area and generate the signal, which will be used to display the corresponding screen image.

The following program writes an "A" on the screen in line 2, column 48:

```
...
mov   bx,0b000H             ; bx is initialized as b000 (hexa)
mov   ds,bx                 ; register ds := bx
mov   bx,100H
mov   byte ptr[bx],41       ; A (code 41 hexa) is written at address
                            ; bx+10H*ds = 0b0100H
                            ; (see chapter on memory)
...
```

6.3. Exchange modes

Data transfers between an EU and a processor involve three techniques:

1) *polling*;
2) *Direct Memory Access* or DMA;
3) *interrupts*.

6.3.1. *The polling exchange mode*

This mode is characterized by permanent polling of the exchange units by the processor. The processor reads the status registers, testing the bits involved in the exchange. Once this is done, the data received are read or written (Figure 6.23).

This work mode, known as either *polling* or *busy waiting*, is particularly simple to implement but has the drawback of taking up the entire activity of the processor for the intput–output operation.

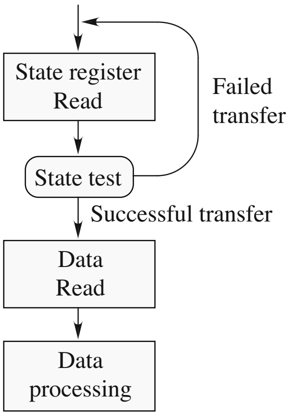


Figure 6.23. *The busy waiting process*

Consider again the example of serial-asynchronous transmission described previously. To send a character to the terminal, we need to at least test the status register bit that indicates whether the controller is ready to send. This bit indicates whether the shift register that must receive the character code is available or not. If we assume that the bit in question is bit 5 of the status register, the test is done as follows (if this bit is not equal to zero, the controller is not ready):

```
myloop:
   mov A,(StatReg) ; status read (the status register
                   ; has the address StatReg)
   and A,20H       ; bit 5 is singled out
   jmp nz,myloop   ; if bit 5 <> 0 we come back to test the status
```

6.3.2. *Direct memory access*

Direct memory access (DMA) consists of giving the exchange unit the possibility of reading or writing directly into memory without going through the processor. This method:

– saves the processor from handling this transfer task;

– may take over for the processor in cases where it is not fast enough to perform the *processor – exchange unit* transfer between two *exchange unit* ↔ *peripheral* transfers, etc.

6.3.2.1. *Concept*

The DMA controller is “inserted" between the processor and the peripheral controller (Figure 6.24).

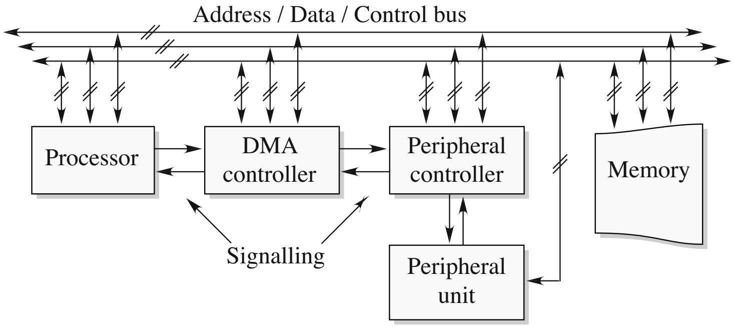


Figure 6.24. *DMA controller scheme*

It is equipped with a word transmission counter, which is incremented or decremented at each transfer, and with an address register and exchange control registers that are initialized from the processor before starting the exchange.

As we have already mentioned, there can be *conflict access* involving the address, data and control buses (Figure 6.25). When the DMA controller gains access to the memory, the processor must abstain from doing the same.

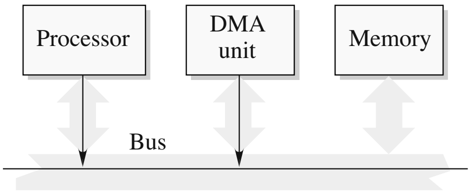


Figure 6.25. *Access conflict*

6.3.2.2. *Exchange protocol*

When the peripheral unit controller has data at its disposal (Figure 6.26):

1) it transmits a DREQ, or *DMA Request*, to the DMA controller;

2) the DMA controller forwards this request to the processor (HRQ, or *Hold Request*);

3) after *acknowledging* this request, the processor has its buffers set to their high impedance state;

4) the DMA controller can take over control of the buses. It sends a *DMA Grant* to the exchange unit, a grant that serves as a *chip select* signal.

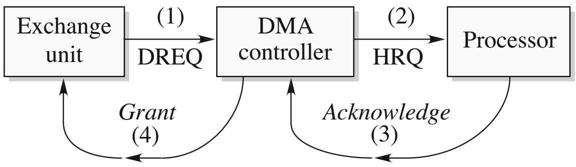


Figure 6.26. *Direct memory access protocol*

The controller is equipped with a READY input, used for inserting wait cycles if the memory access times are too long, just as a processor does.

EXAMPLE 6.2.– [“Peripheral to machine” transfer cycle] The following example is based on the operation of the I8237A DMA controller by Intel®. We will consider the case of two transfer cycles between an exchange unit and the memory, describing in detail the different phases involved.

1) Figure 6.27: initialization of the DMA controller (address and parameters of the transfer, initialization of the word counter, etc.) and execution of the transfer command on the exchange unit.

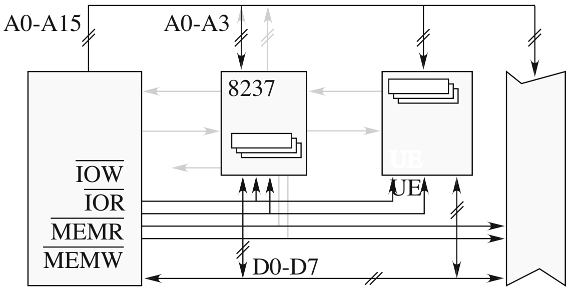


Figure 6.27. *DMA protocol: initializations*

2) Figure 6.28: the exchange unit emits a DREQ access request.

3) Figure 6.29: the DMA controller forwards this request to the processor (HRQ). This corresponds to a request to take over control of the buses.

4) Figure 6.30: once the processor has yielded control of the buses, it sends the DMA controller an HLDA (*Hold Acknowledge*).

5) Figure 6.31: the AEN line is used for controlling access to the bus by other exchange units that are part of the computer. This cycle is the first effective work cycle in DMA.

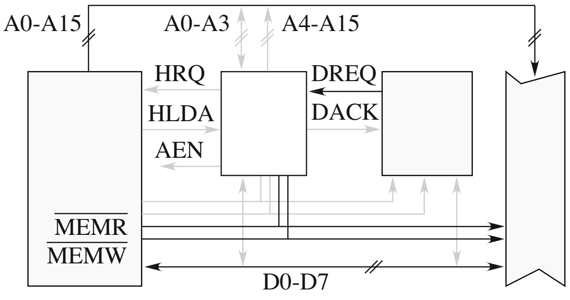


Figure 6.28. *DMA protocol: DREQ*

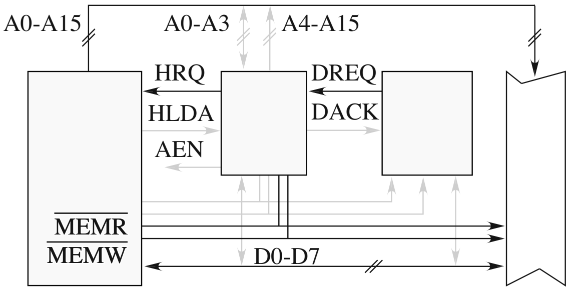


Figure 6.29. *DMA protocol: HRQ*

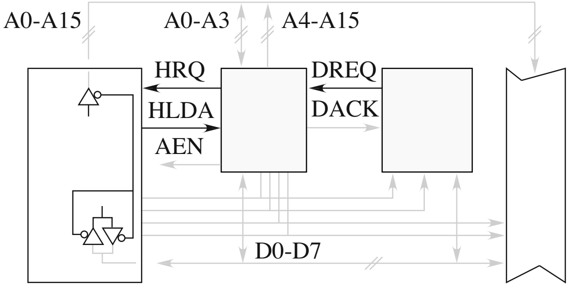


Figure 6.30. *DMA protocol: hold acknowledge*

6) Figure 6.32: the transfer of an element of data takes up three clock cycles denoted by S2, S3 and S4. The DMA controller emits an address and the required read/write signals (here we are reading from the peripheral control and writing into memory).

7) Figure 6.33: transfer of a second element of data (cycles S2, S3 and S4) and end of transfer (known as EOP, the acronym for *End Of Operation*).

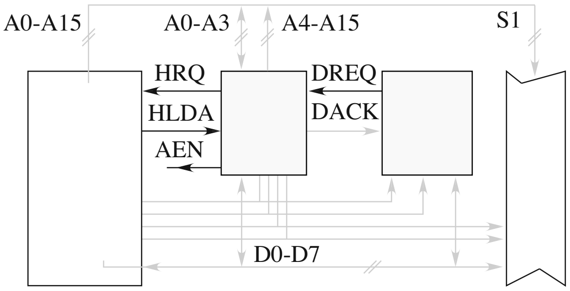


Figure 6.31. *AEN signaling that control of the bus has been taken over*

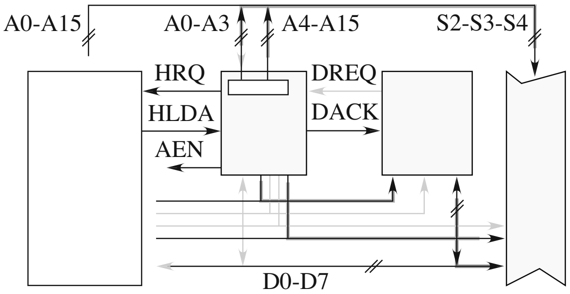


Figure 6.32. *First transfer, the processor no longer has access to the buses*

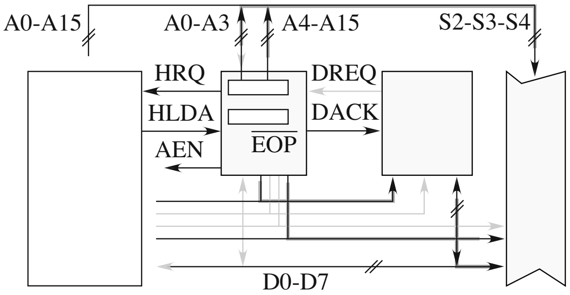


Figure 6.33. *Second access*

8) Figure 6.34: the EOP is processed by ending the emission of the HRQ.

9) Figure 6.35: control is returned to the processor.

The DMA control can follow several transfer modes: the three modes described below are based on the operation of Intel's I8237®.

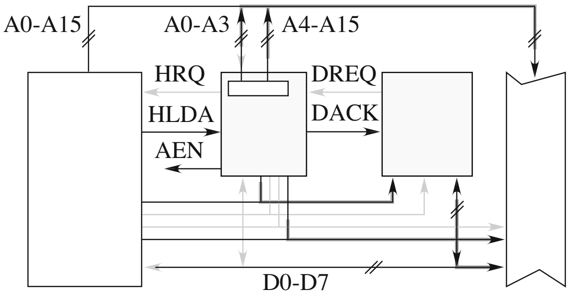


Figure 6.34. *DMA protocol: HRQ is "dropped"*

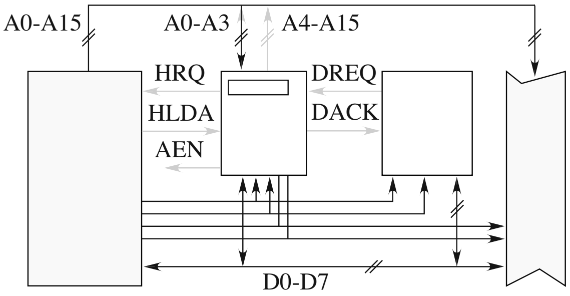


Figure 6.35. *DMA protocol: the processor takes back control*

6.3.2.3. *Halt burst mode*

This operating mode consists of keeping the DMA controller active throughout the entire transfer. DREQ is kept active so long as the acknowledgment from DACK has not been received. The DMA controller only yields control when the word transmission counter changes from 0 to FFFF_{16}. This mode allows high transfer rates at the cost of completely suspending processor activity. In practice, this is not quite true. The processor no longer has access to the buses, but can continue working with its cache memories (Chapter 8).

Certain controllers can automatically restart a transfer cycle without intervention from the processor (*auto-initialize mode*).

6.3.2.4. *Single transfer mode*

During the DMA, control is given back to the processor after each transfer. Even if the DREQ is kept active, HRQ switches off to give the processor at least one additional cycle (this is the case for the I8237 by Intel® in *single transfer mode*).

6.3.2.5. *Demand transfer mode*

In this DMA transfer mode, the transfer occurs either until the counter switches from 0 to FFFF$_{16}$, or until DREQ switches off (the peripheral controller has run out of data to transmit, this is the case for the I8237 by Intel® in *demand transfer mode*). In between transfers, the processor can take back control since the content of the address registers and of the counters is automatically saved. This transfer mode is well suited for slow transfers, or for when the bandwidth is highly unsteady.

EXAMPLE 6.3.– The PC–AT could use DMA for exchanges between peripherals. It was equipped with two DMA controllers set up in a cascade configuration as shown in Figure 6.36.

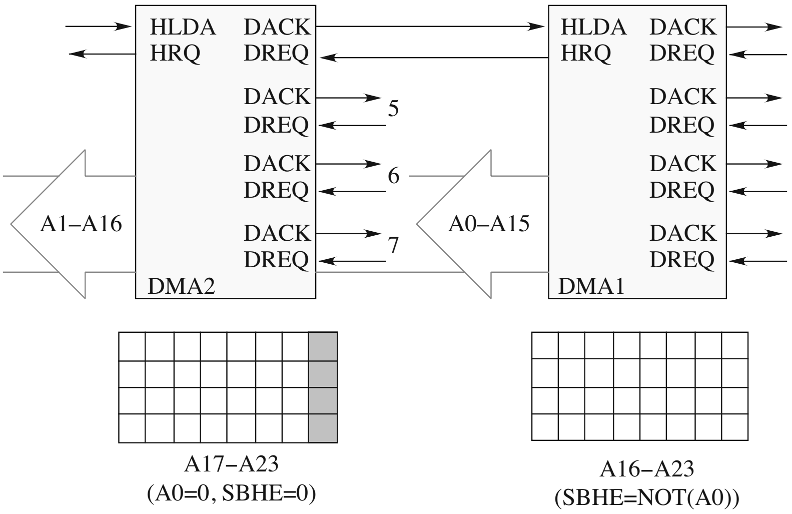


Figure 6.36. *DMA channels*

Each controller handles four exchange channels: this means that it is equipped with four "address register + counter" sets. In the PC–AT, the channels were numbered from 0 to 7:

– channel 0 for the memory refresh;

– channel 1 useable for extension cards;

– channel 2 reserved for the floppy disk controller;

– channel 3 reserved for the hard drive;

– channel 4 reserved for cascading the two controllers;

– channels 5–7 available.

The addresses generated by the controllers are 16 bits in length. This explains the presence of two sets of four registers as extra capacity when addressing 20 lines (PC) or 24 lines (AT): these are called *page registers*.

6.3.3. *Interrupts*

In the section on DMA, we mentioned that during a task in halt burst mode, the end of the transfer is identified by the fact that the DMA controller yields to the processor. In the two other modes on the other hand, the only way to know if the transfer has been completed is to test the status register of the controller to find out if the transmitted word counter has switched to zero.

Likewise, a machine dedicated to industrial process control (monitoring, control, etc.) must be able to suspend a process to quickly assess external events, whether planned or not. It is therefore necessary to have a way to signal that an input–output activity is completed, that an unplanned event has occurred, etc., without having to perform repetitive queries. The *interrupt* mechanism provides these abilities.

An interrupt is an external event which causes the current processor task to stop so that another task can be executed. Physically, this event takes the form of a voltage variation on an "input" line of the processor's control unit (Figure 6.37).

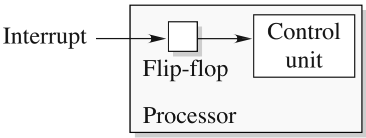


Figure 6.37. *Interrupt command*

For an interrupt to be acknowledged, the control unit must constantly test the status of the interrupt line. A flip-flop memorizes the voltage changes on the line so that it is possible to know if an interruption has occurred. This is achieved by doing the following: the cycle for each instruction begins by testing the flip-flop. Depending on the result, the processor executes the instruction or handles the interrupt (Figure 6.38).

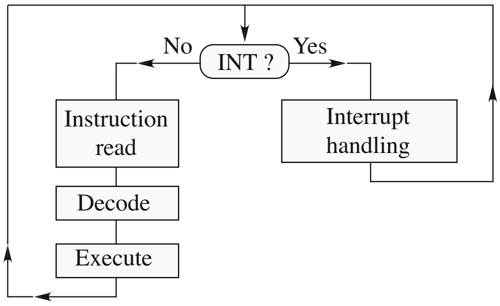


Figure 6.38. *Checking of an interrupt during an instruction processing*

6.4. Handling interrupts

6.4.1. *Operating principle*

Once the interrupt flip-flop has been set, the hardware can begin handling this interrupt. This occurs in the following phases:

6.4.1.1. *The first phase*

The first phase of the process can be decomposed as follows:

– The current instruction ends.

– The return address (the address of the following instruction), along with the content of the flag register, are saved in a stack (through a mechanism similar to the one implemented when calling a subroutine). Depending on the processor, or on the type of interrupt, the content of the registers, or of part of the registers, is saved in the stack. If this is not the case, it will fall upon the processing program to save what needs to be saved.

– The interrupt may be inhibited, in which case the process program will validate them again if necessary. On multimode machines, the processor switches to *system mode* (or *supervisor mode*).

– The processor may send an acknowledgment signal on a command line of the control bus. This acknowledgment is used by the logic circuit of the exchange unit that sent the interrupt.

6.4.1.2. *Calculating the handling address*

The machine begins handling the interrupt at an address, which can be determined through several methods:

– *Non-calculated direct mode*: This is the most common mode used during a hardware reinitialization. The program counter is loaded with a predetermined value.

EXAMPLE 6.4.– When booting the Intel® I8086, the code segment register CS is loaded with F000H and the program counter with FFF0H. This information can be checked by visualizing the content of the BIOS memory. A `jmp` instruction is located at that address.

– *Non calculated-indirect mode*: Each interrupt line is associated with one or several memory cells containing the address of the appropriate handling subroutine. This is how the Motorola® MC6800 operated.

– *Calculated-direct mode*: The content of the program counter is calculated based on information provided by the exchange unit that interrupted the processor.

EXAMPLE 6.5.– In the Intel® I8080 microprocessor, the program counter was loaded with the value $n \times 8$, where n is a value given by the interrupt controller on the data bus. The acknowledgment cycle is actually very similar to an instruction fetch cycle. The interrupt controller waits for the acknowledgment signal provided by the processor before taking control of the bus and using it to "display" an `RST n` (*restart*) instruction, where n is the interrupt number. The processor executes this instruction, which causes the program counter's content to be saved in the stack, followed by a branch at the address $8n$.

– *Calculated-indirect mode*: The exchange unit sets the data bus to an identification number read by the processor. This number, known as the *interrupt vector pointer*, is actually an entry in an address table. These addresses belong to the handling program, and can be modified at will. The table which holds them is called the *interrupt vector table*, and we use the phrase *vectored interrupts*. Figure 6.39 illustrates the calculated-indirect mode.

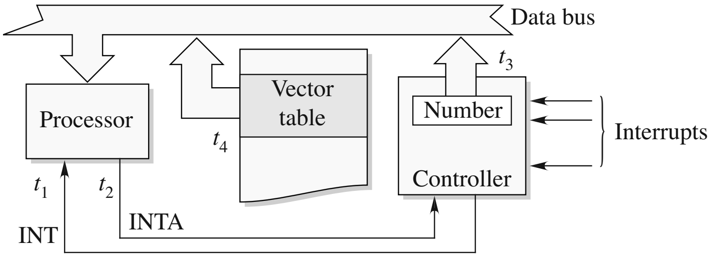


Figure 6.39. *Interrupt vector*

- at time t_1, the exchange unit sends an interrupt;

- at t_2 the processor replies with an acknowledgment;

- at t_3, the exchange unit sets the data bus to an interrupt number N read by the processor;

- at time t_4 the processor fetches, at the address that was calculated, the address of the interrupt handling program.

Note that the interrupt controller must wait for the acknowledgment signal to be received, since this is how it knows whether the bus is available when it presents the interrupt number.

6.4.1.3. *The third phase*

The interrupt handling program ends with a special return from interrupt instruction (`REI`, `RTI`, etc.) used for fetching the return address and, possibly, the content of other registers. If the stack mechanism is used, this instruction is similar to the return from subroutine (`RET`, `RTS`, etc.).

6.4.2. *Examples*

EXAMPLE 6.6.– A machine equipped with a stack mechanism has a 16-line address bus, an 8-line data bus and a 16-bit stack register. The address of the interrupt handler that is needed is located in FFF8_{16} (most significant) and FFF9_{16} (least significant). Consider the following initial situation (Figure 6.40):

– the program counter contains 1C00_{16};

– and the stack pointer A01C_{16}.

The stack pointer provides the address of the first available element in the stack.

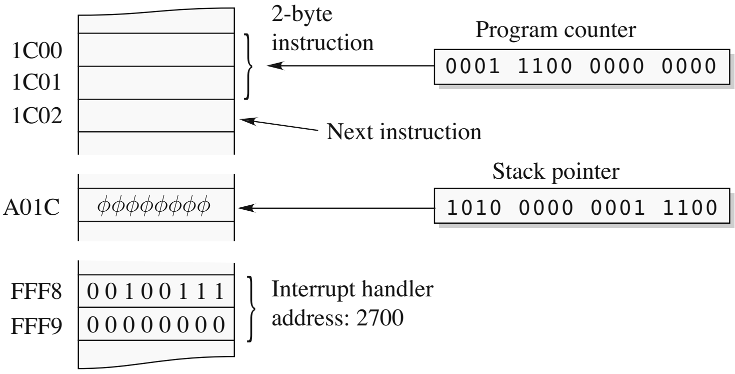


Figure 6.40. *Initial situation*

After the interrupt, the return address $1C02_{16}$ is stored in a stack (we assume that this is the only information stored in the stack). The stack pointer is updated with the value:

$$(\text{A01C}_{16} - 2) = \text{A01A}_{16}$$

The program counter is loaded with the address stored in FFF8_{16} and FFF9_{16}, i.e. 2700_{16} (Figure 6.41).

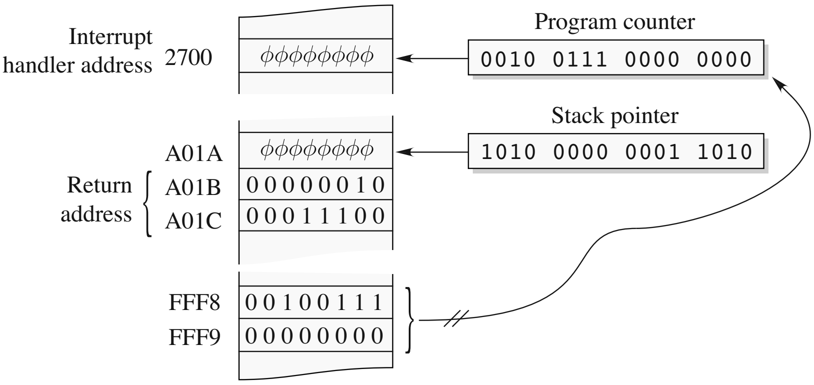


Figure 6.41. *After the interrupt*

At the end of the process, the program counter is reinitialized with the address that was saved in the stack. The stack pointer is updated (Figure 6.42).

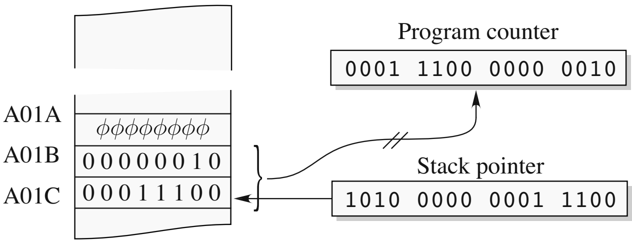


Figure 6.42. *Handling the return from interrupt*

EXAMPLE 6.7.– The Intel® I8086 microprocessor provides two acknowledgments (INTA, short for *INTerrupt Acknowledge*) during the handling of the interrupt.

The first is used to “freeze” the status of the interrupt controller (Figure 6.43). The second is used for transferring the interrupt number (1). This number is multiplied by four to produce an entry in the vector table (2). If the controller presents the number 12H, the processor will fetch from the address $4 \times$ 12H the 4-bit handling address (3). The content of the two memory words with addresses 48H and 49H is then loaded into the program counter, while the content at 4AH and 4BH is loaded into the *code segment* register (see Chapter 7 on memory). Then, the handling begins (4).

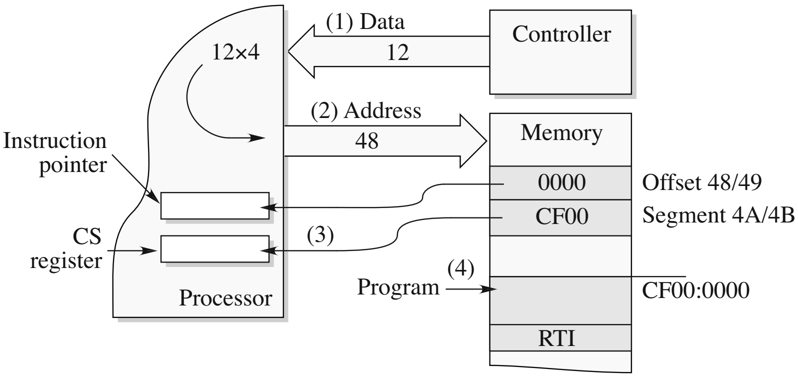


Figure 6.43. *Interrupt vectors on the Intel® I8086: the offset at address 48/49 is equal to 0 and the segment at the addresses 4A/4B is equal to CF00*

6.4.3. *Software interrupts*

Interrupt instructions or *software interrupts* rely on the same mechanisms as those involved in hardware interrupt handling. On the Intel® I8086, the `INT N` instruction, where N is an integer coded in eight bits, triggers an interruption that redirects to the program with the address contained in the entry N (address $4N$) of the interrupt vector table.

6.4.4. *Masking and unmasking interrupts*

Some interrupts can be of such importance (interrupts generated by alarm systems, for example) that it is not possible to either:

– *interrupt their handling*: It is then essential to prevent other interrupts from being handled. This is called *masking* these other interrupts.

– *or prevent their handling*: These are known as *non-maskable* interrupts.

Interrupt masking can be achieved by setting certain bits of the *interrupt register* using special masking and unmasking instructions. Upon reception of a maskable interrupt, its masking can be performed automatically. Its unmasking is then performed by the "return from interrupt" instruction.

EXAMPLE 6.8.– The external interrupt signal is masked by setting the bit associated with this interrupt to one in the interrupt register (Figure 6.44).

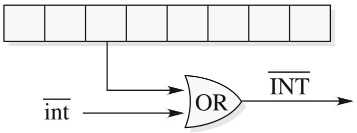


Figure 6.44. *Interrupt masking*

6.4.5. *Interrupt priorities or levels*

In a machine working with interrupts, each exchange unit may send interrupts to the processor. All of these interrupts can be combined using a simple logic gate (Figure 6.45).

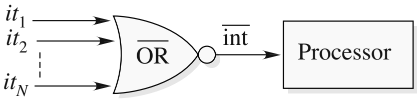


Figure 6.45. *Combining interrupts*

Interrupts that reach the processor have a first level of priority defined by the hardware (*vertical priority*). But, for a given priority, the handler must:

– send queries to the exchange units to find out which ones have sent an interrupt;

– determine which one must be handled first.

We can use an *interrupt controller* to combine the interrupt lines (Figure 6.46) and in some cases store into memory and handle priorities in between interrupts. The controller provides the number of the interrupt that was chosen.

6.4.6. *Similar mechanisms*

Other events implement a mechanism similar to the one used for interrupts. These are called *exceptions*. An exception is an event that occurs during the execution of an

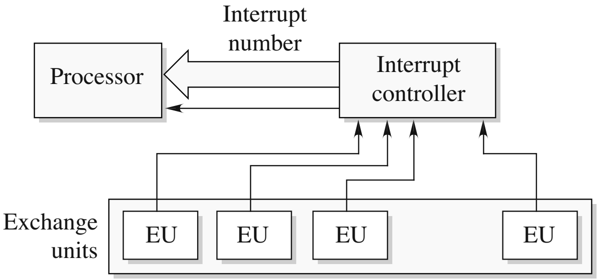


Figure 6.46. *Interrupt controller*

instruction (overflow, division by zero, etc.). Unlike the interrupt, the exception stops this execution (*synchronous* event). The exception handling program decides how to pursue the process. As interrupts can be masked, exceptions can be inhibited.

There are three categories of exceptions:

– *Trap*: exception that is handled after the execution of the problematic instruction. Traps include the handling of arithmetic errors (*overflow*, *underflow*, etc.), instruction code errors, software interrupts, etc. After the trap has been handled, the execution may resume at the following instruction, as with an interruption.

– *Fault*: the exception detected and handled during the execution of the problematic instruction. The instruction can be restarted after the fault has been handled (page fault, illegal memory access, invalid operation code, wait instruction, etc.);

– *Abort*: the exception that occurs during the execution of an instruction. It is impossible to restart (hardware errors, attempts to access a protected area of memory, etc.).

6.5. Exercises

Exercise 6.1 (Serial input) (Hints on page 331)

Consider a machine with the fetch timing diagram (based on the Intel® I8080) shown in Figure 6.47.

We wish to build an interface that allows fetch operations on a serial line according to the scheme in Figure 6.48.

Write a fetch function for characters coded in eight bits. You may use a language syntax similar to the one used for the Intel® 8086. The address assigned to the input line will be denoted by `addrIN`.

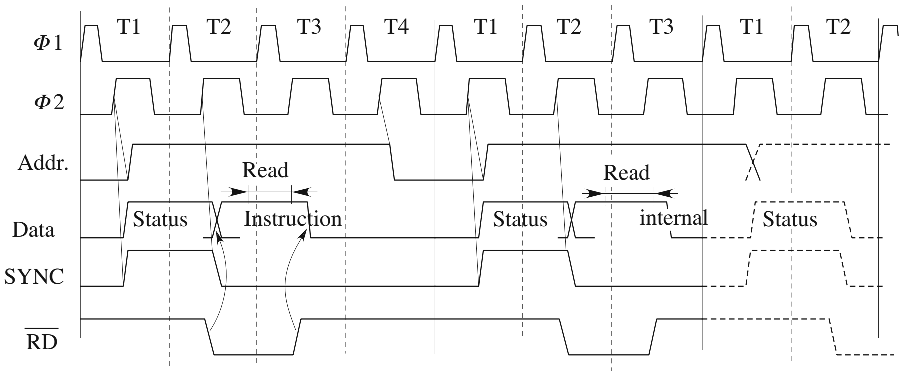


Figure 6.47. *Fetch timing diagram*

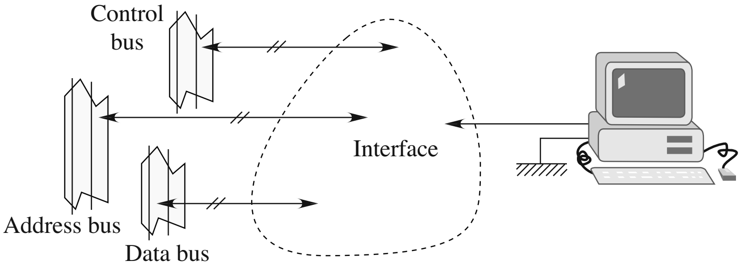


Figure 6.48. *Fetch scheme*

Exercise 6.2 (Serial output) (Hints on page 332)

The same machine is used, with the write timing diagram shown in Figure 6.49.

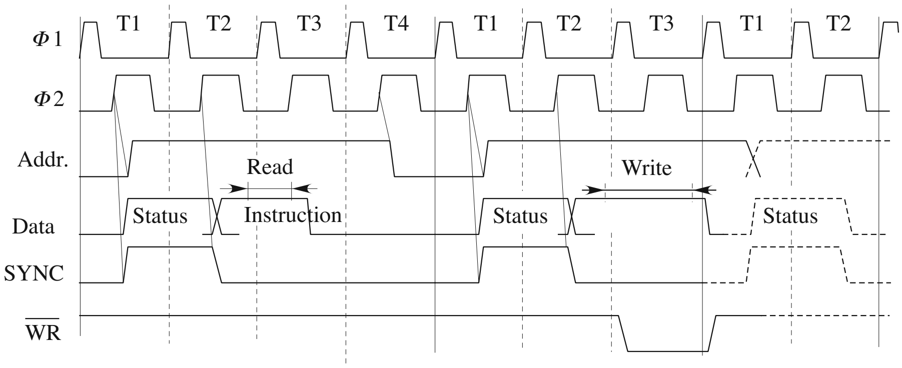


Figure 6.49. *Write timing diagram*

Build an interface that allows write access and write a write function. The address assigned to the output line will be denoted by `addrOUT`. Assume that the character are eight bits in length and use a single stop bit.

PART 3

Memory Hierarchy

Chapter 7

Memory

This chapter deals with how hardware manages memory space, an issue with considerable impact on machine performance. The information traveling between the memory and the processor is indeed the *main flow of information* in a computer. It is therefore no surprise that many efforts have been made to improve both communication between these two components and memory "speed". As a result, the size, access time and organization of memory are key to machine characteristics.

7.1. The memory resource

The processor fetches from the memory the *instructions* and the *data* that it uses to perform a given task. The location of each of these elementary pieces of information is "mapped" using an address, which is "presented" to the processor on the address bus. The transfer between the memory and the processor is then achieved via the data bus (Figure 7.1).

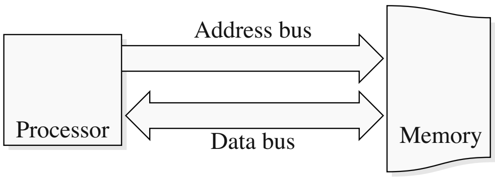


Figure 7.1. *Memory–processor interactions*

The main characteristics of memory are the following:

– The *size of the addressable space*, defined by the number of lines of the address bus.

– The *physical size*, meaning the amount of memory actually installed. This parameter strongly depends on hardware considerations, for example the number of available slots and the technology used (kits of 4, 8, 16 GB, etc.), and on cost considerations.

– The *access time*, defined as the interval between the moment memory is "addressed" and the moment it delivers the information. This parameter depends on the technology and the organization of memory.

– The *latency*, defined as the interval between the moment the processor performs the addressing and the selection and the moment the information becomes available ("stabilized") on the lines of the data bus. This parameter depends on the access time, the memory management devices, and the bus characteristics.

The memory cannot function unless it is combined with a device capable of signaling whether an element of data is ready to be fetched. This device is in the form of a command line (Figure 7.2), the status of which can trigger the insertion of *wait states* in the execution of the instruction (see example in section 5.2.1).

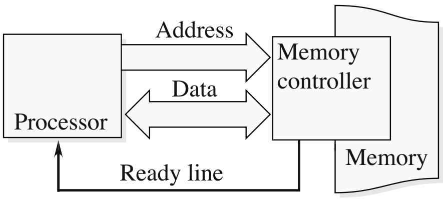


Figure 7.2. *Memory control*

7.2. Characteristics

The entity using the memory resources (user program, operating system, etc.) must be aware of its size, access times and protections that have been implemented:

– Access time is a factor bearing directly on the execution speed of programs. It is, however, difficult to separate it from other parameters that determine machine performance: processor speed, algorithm efficiency in managing memory, memory size, etc.

– Memory management, which is handled by hardware or software, must offer a level of security that guarantees the integrity of the information processed and prohibits the involuntary destruction of programs. This management must ensure the separation between the programs and/or the data of the users (Figure 7.3). If the machine serves only one user, the memory space is distributed between the operating system and user's programs and data. It is therefore essential to ensure that any programming errors on the part of the user do not lead to modifications of the area assigned to the operating system or to the user's program. If several users, or several tasks, make use of the machine's resources "simultaneously", and if, as a consequence, several programs reside in the memory at the same time, then they must be prevented from writing in, or fetching from, an area of memory that is not assigned to it.

– It is preferable for the memory size to be large enough to store programs and data without having to perform manipulations such as block exchanges between disk and memory when the memory size is too small.

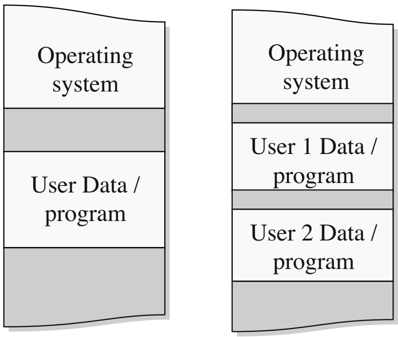


Figure 7.3. *Single- and multitasking*

7.3. Memory hierarchy

The ideal machine should have limitless memory capacity and an access time compatible with the processor speed. This solution is not economically feasible: the cost per "installed bit" of electronic memories is far greater than it is for magnetic technology memory (disks). Based on cost and performance criteria, a *hierarchy* can be established, a concept present in any architecture.

Implementing a memory hierarchy consists of using memory modules with access times that are "shorter" as they get "closer" to the processor. Programs and data can end up being in part duplicated onto several memory modules with very different physical characteristics. The access mode implemented for managing this hierarchy is as follows (Figure 7.4):

– When the processor wants to access information, it fetches it from the physically "closest" memory unit, in other words the fastest.

– If it cannot find it there, it accesses the next level and copies it, along with its *neighbors*, into the level with the shortest access time. This mechanism of copying in blocks increases the chances of success (see the principle of locality in section 7.3.1) in future attempts at the fastest level.

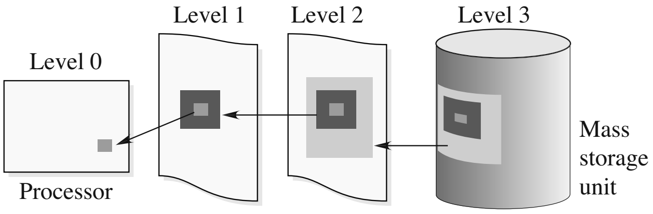


Figure 7.4. *Hierarchy and access method*

7.3.1. *Principle of locality*

Managing a memory hierarchy implies that accessing information observes the *principle of locality*. Memory accesses have two properties pertaining to locality:

– *temporal* locality, which expresses the fact that an address that was just referenced will likely be referenced again in the near future;

– *spatial* locality, which expresses the fact that if an address was just referenced, its *neighbors* will also be referenced in the near future.

This leads us to define the concept of the *working set* in the time window F at time t, denoted by $W(t, F)$, which is the set of addresses referenced between t and $t + F$.

EXAMPLE 7.1.– Consider the code in C of the following adding loop:

```
for (counter = 0; counter <= 99 ; counter++) sum = sum + 1;
```

and the corresponding code in machine language:

```
loop:
    incl %edx     ; counter
    incl %eax     ; sum
    cmpl $100,%edx
    jne loop
```

Let us assume that each instruction lasts T units of time. The principle of locality, which is divided into *temporal locality* and *spatial locality*, in this case is observed:

– for instructions:

- with respect to time, we will execute the loop 100 times and the same instruction will be referenced periodically;

- with respect to space, each time we access an instruction, we then reference its neighbor;

– for data:

- with respect to time, the variables `sum` and `counter` are referenced every $4T$;

- with respect to space, these variables are memorized in registers.

7.3.2. *Hierarchy organization and management*

The levels of a memory hierarchy are ordered from the fastest and smallest (*low* levels) to the slowest and largest (*high* levels). Figure 7.5 illustrates these different levels and the corresponding mechanism. Note the cache memories and the virtual memory management, which we will come back to in the later chapters.

In the case of caches, a block will be known as a *line*, and in the case of virtual memory as a *page*. In practice, lines have a size of several tens of bytes, and pages of several kilobytes.

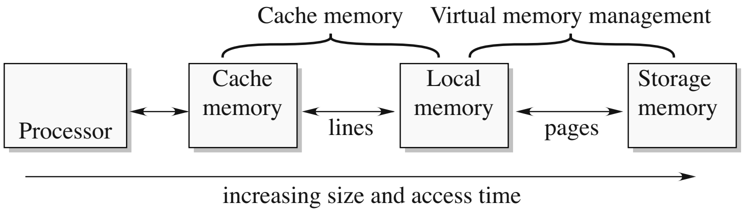


Figure 7.5. *Memory hierarchy*

There are two essential points to note on hierarchy management:

– the fast levels must manage the memory references by storing the most recently used addresses (temporal locality);

– transfers are performed in blocks (spatial locality).

7.3.3. *Definitions and performance*

Consider a two-level hierarchy, where M_1 is the fast level and M_2 is the slow level. We define the properties *size*, *access time* and *cost per bit* for these two levels:

	Size	Access time	Bit cost
M_1	S_1	T_1	C_1
M_2	S_2 $(S_1 < S_2)$	T_2 $(T_1 < T_2)$	C_2 $(C_1 > C_2)$

and the concept of *hit* or *miss*:

Hit	When a memory reference is satisfied in M_1
Miss	When a memory reference is not satisfied in M_1
Hit ratio (h)	Number of *hits*/total number of references
Miss ratio $(1 - h)$	Number of *misses*/total number of references

The values T_1 and T_2 are defined by:

T_1	Time needed to satisfy a *hit*
T_2	Time needed to satisfy a *miss*

The performances of this hierarchy with two levels M_1 and M_2 can then be evaluated with the help of:

– the mean memory access time:

$$T_m = h \times T_1 + (1 - h) \times T_2 \qquad [7.1]$$

– the speedup S due to the presence of M_1:

$$S = \frac{T_2}{T_m} = \frac{T_2}{h \times T_1 + (1-h) \times T_2} = \frac{1}{1 - h \times (1 - \frac{T_1}{T_2})} \qquad [7.2]$$

As you can see, if $T_2/T_1 = 10$, a decrease in 1% in a hit ratio of about one results in an increase of the mean access time by $\simeq$10%.

7.4. Memory size and protection

User expectations in terms of operating systems, the implementation of elaborate user interfaces, the use of software with ever-expanding features, image and sound processing, etc., require larger and larger amounts of memory space. Although there is no theoretical limit to how much we can expand the size of the address bus, this does lead to a certain number of hardware issues:

– Issues with *designing* the processor: the internal address bus gets “bigger” as the directly accessible memory space grows in size. Today, machines have passed 32-bit mark and reached 64 bits.

– Issues with *compatibility*: when a manufacturer decides to extend its range of hardware “forward”, it strives to maintain some compatibility with existing hardware. This forces them to come up with designs that provide for an increase in memory size with no modifications to the set of instructions.

It is also essential to have mechanisms available for protecting the memory (areas of memory assigned to a specific task or a given user, or that are read only, etc.) whenever using a multitask operating system. These devices, which are managed by software, are also partly under hardware control.

The solution consists of inserting a *translation* mechanism (Figure 7.6) between the address read in the instruction or generated by the execution of an instruction, known as a *virtual or logical address*, and the address that will actually be presented to the address bus (*real or physical address*).

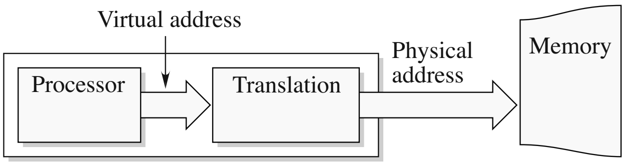


Figure 7.6. *Virtual and physical addresses*

These translation mechanisms can be implanted at the processor level or inserted between the processor and the memory in the form of a specialized controller. The following sections present the designs for these translation mechanisms.

7.5. Segmentation

Memory cells can be differentiated based on how they are used, by associating instructions and data with distinct areas of memory, called *segments*: *code segments*, *data segments*, *stack segments*, etc.

7.5.1. *Using segment registers: an example*

The solution chosen by Intel® for its I8086 series microprocessors is a simple illustration of how to manage memory using segmentation. We saw in section 4.1.2 how the design is based on four segment registers.

Each segment has a maximum size of 64 kB (2^{16} bytes). The total memory space is 1 MB (2^{20} bytes) and can be pictured as a partition of $65,536$ 16-byte "paragraphs", or $65,536$ segments "covering" $65,536$ bytes. The four segments used at a given time can overlap completely, partially, or not at all, depending on the value given to the corresponding registers (Figure 7.7).

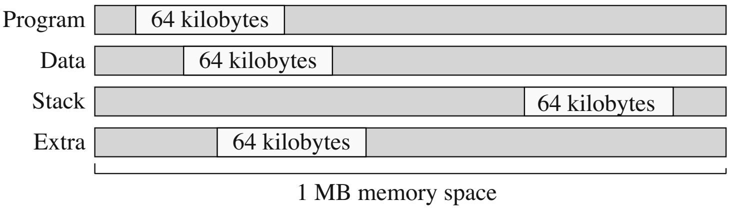


Figure 7.7. *Segment overlap*

Controlling access to different areas is ensured so long as the corresponding segments do not overlap. This is ensured by loading the adequate values into the segment registers. A memory write can only modify a word in the data segment associated with this instruction, since the set of instructions defines the range of application of each instruction, in particular the segments involved in the execution of each of these instructions. As an example, a branch instruction involves the content of the code segment register `cs`, and the actual physical branch address will be obtained from:

$$(16 \times [\texttt{cs}]) + \text{ operand}$$

On the other hand, a transfer instruction between memory and register preferentially uses the data segment register `ds` or possibly the extra segment `es`, depending on what the instruction specifies. The physical address is then:

$$(16 \times [\texttt{ds}]) + \text{ operand or } (16 \times [\texttt{es}]) + \text{ operand}$$

EXAMPLE 7.2.– The instruction `mov [bx],al` triggers:

– the fetching of the instruction from the address:

$$(16 \times [\texttt{cs}]) + \text{ program\_counter}$$

– the transfer of the content of the 8-bit register `al` into the memory word whose offset in the segment defined by `ds` is given by the content of the 16-bit register `bx`. The physical address is therefore:

$$(16 \times [\mathtt{ds}]) + \mathtt{bx}$$

since this `ds` register contains the default segment number associated with this instruction.

EXAMPLE 7.3.– The `push ax` instruction stores the 16-bit register `ax` at the top of the stack. The segment involved is then `ss`, the stack segment. The physical address, we get, is:

$$(16 \times [\mathtt{ss}]) + \text{ stack pointer}$$

A few things to note:

– The adder, which provides the physical address based on the content of the segment register and the offset, causes a propagation delay between the moment the logical address is provided and the moment the physical address becomes available on the address bus. This delay must of course be accounted for by the processor designer.

– Although the addressable memory space, which changes from 64 kB to 1 MB, has been significantly increased, the segment size remains 64 kB. This relatively small value makes it difficult to deal with program or data structures larger than 64 kB.

7.5.2. *Using segment descriptors*

With the Intel® I80286 microprocessor, a much more sophisticated segmentation mechanism was introduced, known as *direct mapping*. In particular, it features the ability to assign access rights to the segments and to specify their size.

The address sent, which is described as *virtual*, is 30 bits in length. The 13 most significant bits of the segment register constitute the *selector*, which serves as an entry in a segment descriptor table of $2^{13} = 8192$ entries. Like the Intel® 8086, this processor is equipped with four segment registers `cs`, `ds`, `es` and `ss`. The segment descriptor consists of 4 bytes. The 24-bit *base address* stored in this descriptor is added to the 16 least significant bits of the virtual address and provides the physical address in memory of the 64 kB segment (Figure 7.8). Bit 2 of the same segment register indicates in which table (*global* or *local* variables) the physical address of the segment can be read. This means we have 1 GB of virtual memory at our disposal, with a physical space of 16 MB:

$$2 \text{ (tables) } \times 2^{13} \text{ (descriptors)} \times 2^{16} \text{ (segment)} = 2^{30} \text{ (bytes)}$$

The descriptor contains the base address of the segment, its size, bits defining its access rights and *service bits* (Figure 7.8).

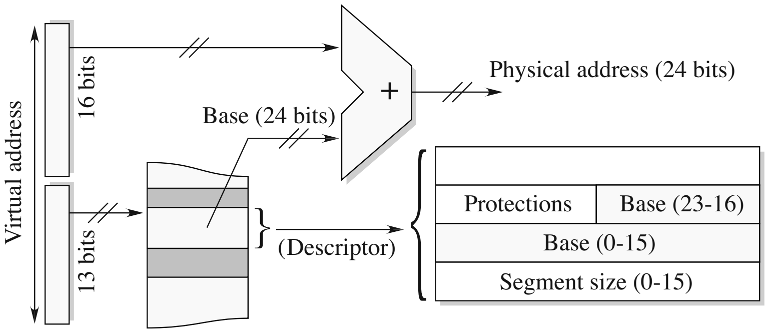


Figure 7.8. *Addressing in the Intel® I80286*

7.6. Paging

This solution consists of *cutting up* the central memory into blocks of equal size called *pages*. Secondary storage is used to provide the *physical address* of each of these pages in the central memory. By *cutting up*, we mean that we have an addressing mechanism which defines an address by:

– a *page number*;

– a *relative address* or *offset* within the page (Figure 7.9).

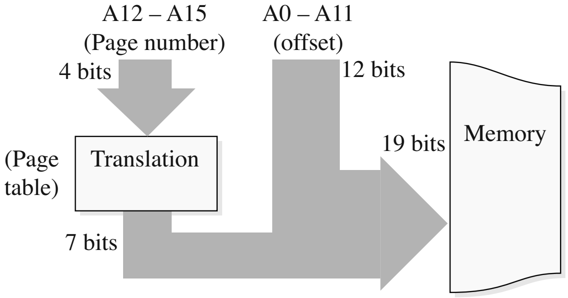


Figure 7.9. *Example of translating from a virtual address to a physical address*

EXAMPLE 7.4.– Consider a processor with an address bus limited to 16 bits. The system is set up to address 512 kilowords. The size of a page is assumed to be equal to 4 kilowords. The 16 address bits consist of 12 page address bits – lines A0 through A11 of the address bus – and four *page number* bits given by lines A12 through A15.

The total accessible memory comprises 128 pages of 4 kilowords. The paging system allows us to access 16 of those 128 pages (example in Figure 7.10).

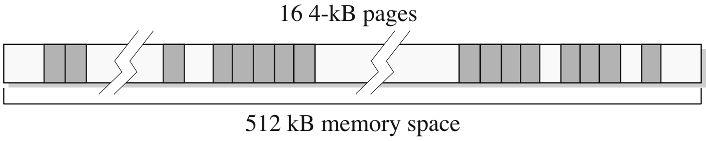


Figure 7.10. *Paged memory space*

The translation mechanism has four input lines that allow it to access sixteen 7-bit cells (or registers). These 7 bits, along with the 12 offset bits, make up the physical address in memory. An operating diagram is shown in Figure 7.11.

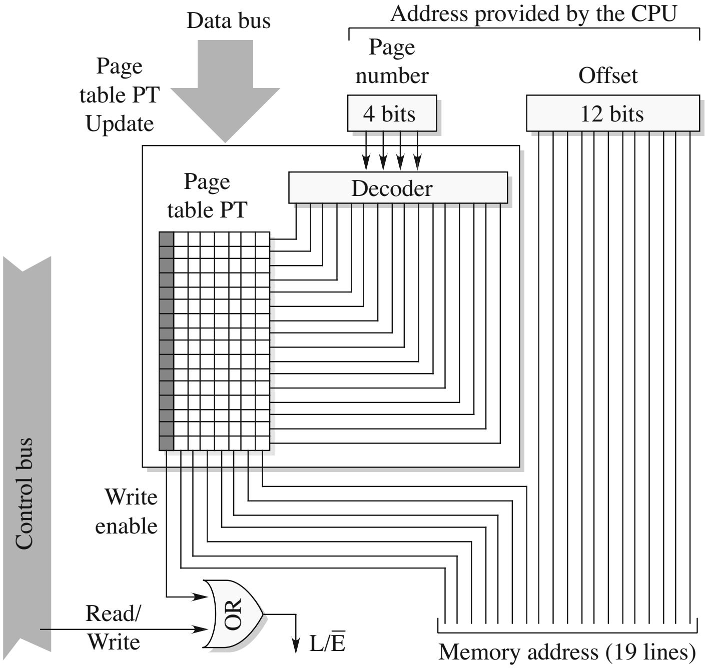


Figure 7.11. *The page table*

The processor “sees” only 64 kilowords of the real physical memory, distributed throughout the physical memory in 16 pages of 4 kilowords. Each entry in the page table, denoted by PT, can be completed with access authorization bits for the corresponding page. Here, we have added a write authorization bit that prevents, when set to one, from writing in the targeted 4 kiloword page.

In a multitask context, a set of pages is allocated to each task by the operating system. These pages can be distinct, to ensure their protection from one another, or shared (in order to share data or code). For each task, a Page Table Directory (PTD) of the pages that are assigned to that task is created in the memory (Figure 7.12).

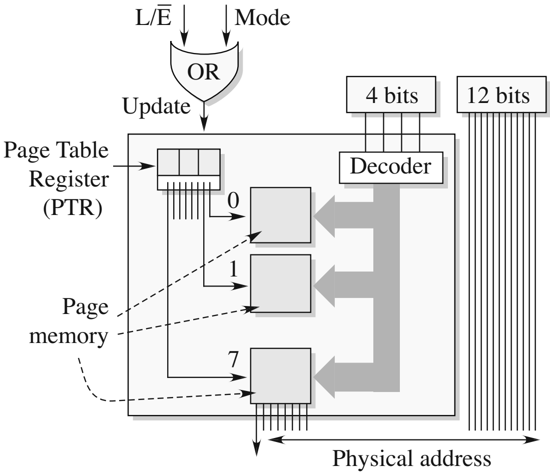


Figure 7.12. *The page table register and the protection mechanism*

Each time a task is executed, the operating system “puts it in charge”, the content of the page table PT is updated by copying it into the content of the PTD. These transfers, which generate a large number of memory accesses, can be avoided by duplicating the PT table. In the following example, eight page tables are enough to address the entire physical memory space. A 3-bit Page Table Register (PTR) provides the number of one of the eight tables. This register is only accessible in kernel mode (mode indicator bit set to one).

REMARKS.–

– Note that the translation mechanism leads to the systematic insertion of a *propagation delay*, as was the case with segmentation.

– The size of the directly accessible memory is still 64 kilowords, despite the size of the physical memory being 512 kilowords.

– The *kernel mode* used here for preventing unauthorized access to the PTR and the page tables is also called *supervisor mode* or *privileged mode*.

7.7. Memory interleaving and burst mode

Memory interleaving is a method for improving memory access time. The idea is to organize the memory in p banks such that the consecutive address words are not physically adjacent. We will describe two such organizations: *C-access* and *S-access*, short for *concurrent access* and *simultaneous access*, respectively.

7.7.1. *C-access*

This technique assigns a data register to each bank. The m most significant *bits* of the address allow us to simultaneously access p memory words. The n least significant *bits* control a multiplexer for accessing one of the words stored in a register (Figure 7.13).

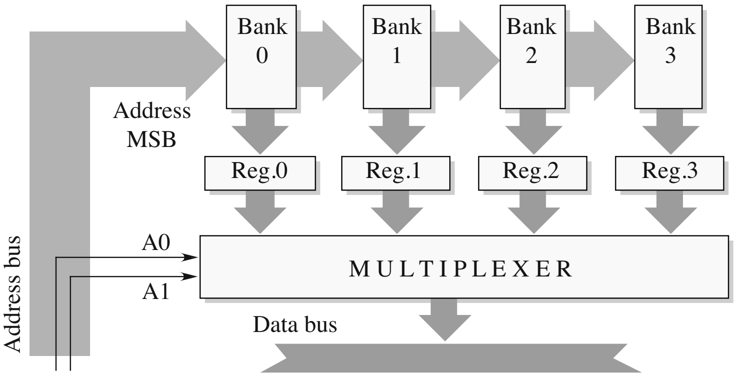


Figure 7.13. *Register multiplexing* $(n = 2)$

The bank addressing phases are performed in parallel. This organization is well suited for accessing the elements of a table. The timing diagram for C-access is shown in Figure 7.14.

7.7.2. *S-access*

This second method assigns an address register to each bank, so that the addresses can be “pipelined”. The m least significant *bits* of the address are decoded to access each of the banks. This type of organization works well for prefetch operations in a *pipeline* architecture (Figure 7.15).

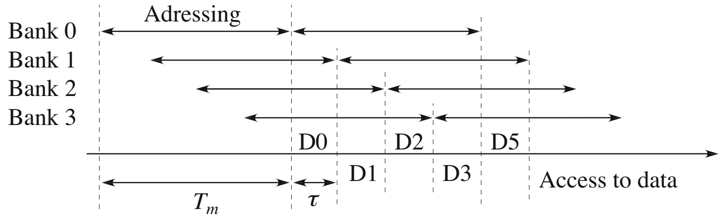


Figure 7.14. *Timing diagram (C-access)*

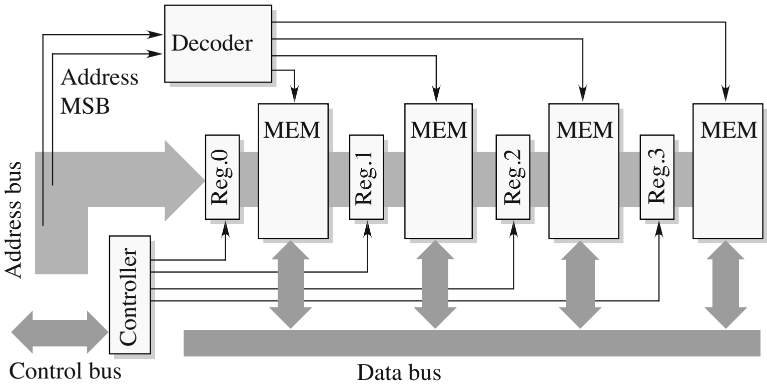


Figure 7.15. *Using address registers*

The timing diagram for S-access is shown in Figure 7.16.

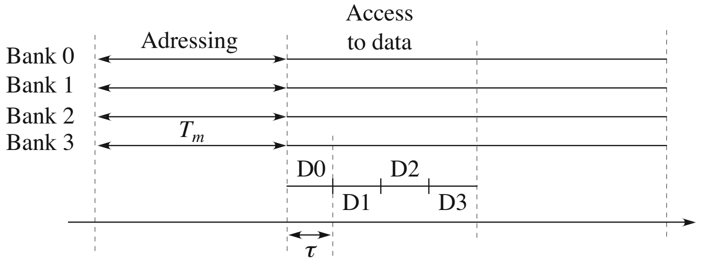


Figure 7.16. *Timing diagram (S-access)*

The time τ_a taken to access k consecutive elements of data ($k < p$) stored starting in bank i is:

$$\tau_a = \begin{cases} T + k\tau \text{ if } i + k \leq \text{ number } p \text{ of banks} \\ 2T + (i + k - p)\tau \text{ otherwise} \end{cases}$$

The two previous access modes can be combined thus (CS-access). The memory banks are then arranged in a table.

7.7.3. *Burst mode*

Interleaving memory banks is a way to artificially reduce the memory access time. *Burst mode* access relies on these access techniques. The address of the first element of data is transmitted to the bus (Figure 7.17). The memory management logic is then in charge of incrementing this address.

This mode is well suited for loading a cache (Chapter 8) or for supplying a machine's instruction buffer with a pipeline architecture (Chapter 10).

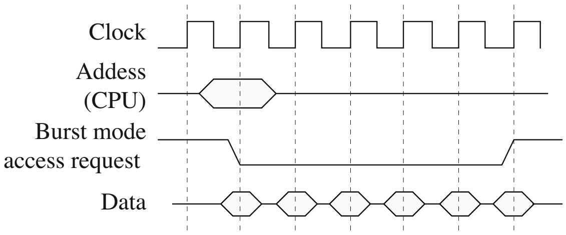


Figure 7.17. *Burst mode*

EXAMPLE 7.5.– Consider the timing diagram in Figure 7.18. When a memory bank is active (*addressed*), the other is in the process of refreshing or accessing the next element of data. In burst mode, only the first cycle requires a wait state. The memory access time after that is equivalent to half the physical access time. In practice, the penalty is only in the order of 3% compared to 0 *wait state* memory (figures provided for the AMD 29000-5 [ADV 87]).

7.7.4. *Prefetch buffers*

The instructions are fetched from the memory, the instruction cache or the branch cache (see section 10.3.3). To ensure that the execution flow is as fluid as possible, the instructions are "prefetched" to a buffer (*Instruction Prefetch Buffer* (IPB)) a few cycles before the processor needs them (Figure 7.19).

When an instruction reaches the decoding phase, its locations is made available for a new instruction.

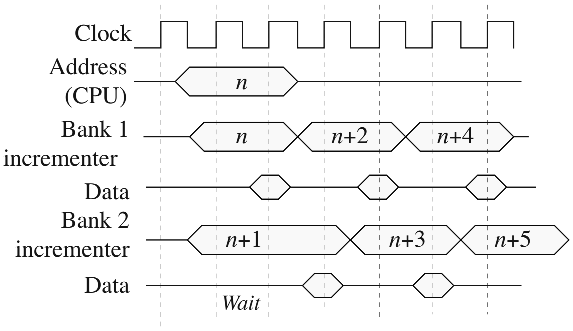


Figure 7.18. *Burst mode and interleaving*

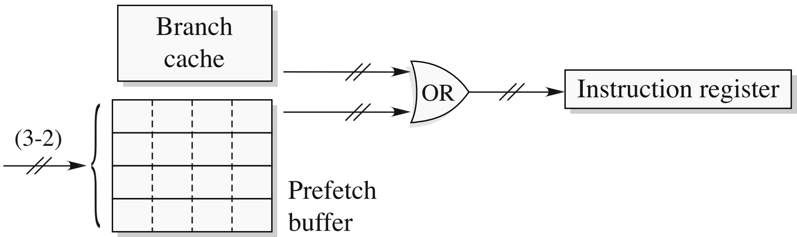


Figure 7.19. *Example a four* 32*-bit instruction buffer*

When a branch (branch, interrupt, trap, etc.) is performed, the buffer is reloaded, usually in "burst mode". This is achieved by having the memory management unit maintain a *fetch address register*, which memorizes the "target physical" address. This makes it possible to resume a preloading sequence that was interrupted. A counter is used for managing the addresses in the flow of prefetched addresses (Figure 7.20).

7.8. Protections, example of the I386

In the Intel® 386 architecture, a memory access performed in "protected mode" must go through the segment descriptor tables (Global Descriptor Table (GDT) and

Local Descriptor Table (LDT)). The descriptors stored in these tables contain the base address, size, type, access rights, etc., of code, data or stack segments, cs, ds, etc. Local tables must have a descriptor in the global table.

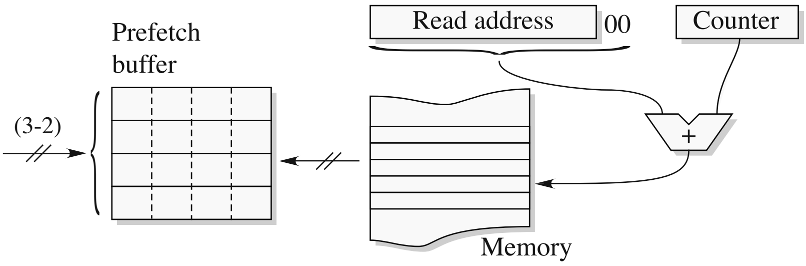


Figure 7.20. *Prefetched address management*

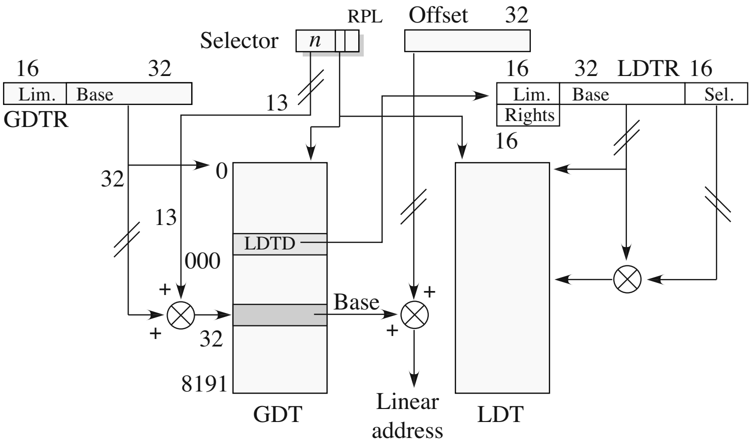


Figure 7.21. *Accessing segment descriptors*

Accessing a byte requires going through a segment selector and an offset. The selector (segment register) is used to access an 8-byte descriptor in the GDT, and thus a base address (Figure 7.21) for the corresponding segment.

In addition to code, data and stack segments, we also have at our disposal Task-State Segments (TSSs), which are used for manipulating the environment and the state of the tasks and local segment tables.

The operating system resources can be accessed through “gates” (call gate, trap gate, interrupt gate and task gate) identified by descriptors. The hardware verifies access rights when the system is solicited.

A table, the Interrupt Descriptor Table (IDT), similar to the global table, includes descriptors for these gates. Access to this table is achieved via indexing based on the interrupt number (hardware or software) or the exception number. The descriptor may correspond to a “task”, “interrupt”, or “exception” gate.

Each selector includes a Requestor Privilege Level (RPL) field for verifying access rights. In machines with an Intel® 386 architecture, there are four privilege levels, from 0, the highest privilege, to 3, which is reserved for applications. Protection levels are as follows:

– Data with a privilege level p cannot be handled by a program being executed at a level (the phrase used is Current Privilege Level (CPL)) $q \leq p$,

– A program being executed at a level q cannot be called by a task being executed at a level $r \geq q$.

When a level 3 program needs to access a level 1 function in the system (via a “gate”), its level is upgraded to 1 until the function returns. This is known as a privilege transfer.

Each segment descriptor has a Descriptor Privilege Level (DPL), which is interpreted differently based on the segment type. There are two indicators, RPL and DPL, so as to prevent a low priority program from using a higher priority function to access data at a higher level (smaller number). The RPL supersedes the CPL in case an access is attempted. Therefore, if RPL $>$ CPL, the processor considers that the task is performed at the RPL level.

Chapter 8

Caches

8.1. Cache memory

Introduced in the mid-1960s by M.V. Wilkes [WIL 65] under the name *slave memory*, a *cache memory*, or simply a *cache*, is a memory that acts as a buffer between the processor and the memory. Its access time is four to 20 times smaller than it is for the main memory. One of the first implementations of this concept was the IBM 360/85 in the 1960s. Its cache memory was 16 or 32 kB in size (sixteen or thirty-two 1-kB blocks of 64-byte lines).

8.1.1. *Operating principle and architectures*

To illustrate the operation of such a device, consider a fetch instruction. When the processor attempts to access this information, it begins by searching in the cache. If the information is not there, the processor copies into the cache a portion of the memory containing this information. This process is invisible to the user. The information stored in the cache is usually not accessible through programming.

It is perfectly conceivable to repeat this mechanism by introducing several layers of cache. Figure 8.1 shows the case of a machine with two levels of cache (today, general-use machines have three levels of cache). The first level (primary cache or level 1 cache) is integrated into the processor, the second is external (secondary or level 2 cache). In this case, the search for information is started in the primary and secondary caches, before moving on to the memory. In the following examples, and to simplify our description, we will most often only consider a single level of cache.

The operating principle of cache memory rests on the *principle of locality* (section 7.3.1): when we fetch a piece of information (data or instruction) from memory, there is a high probability that the following piece of information is located close to the first one. Therefore, if we load a portion of the memory into the cache, it is likely that the processor will only need to access the cache, for a certain time, before accessing the main memory. This time depends, of course, on the size of the cache, the *program structure* and the *data organization*.

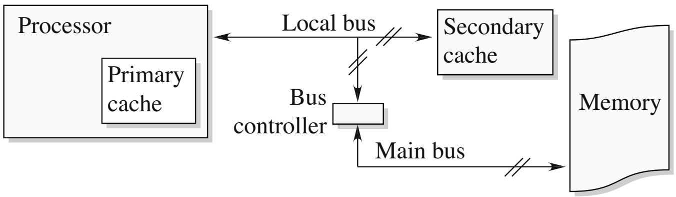


Figure 8.1. *Example of a two-level cache architecture*

8.1.2. *Cache memory operation*

8.1.2.1. *Reading*

In the case of reading an instruction or data:

– if the information is located in the cache, the processor can access it with an access time equal to that of the cache;

– otherwise, the processor loads into the cache a *set* of adjacent words that the desired information belongs to. This set is referred to as a *line*. Its size varies from four to 128 bytes depending on the processor. Once the information is in the cache, the processor can perform the fetch operation.

8.1.2.2. *Writing*

In the case of writing:

– If we are referring to information that is absent from the cache, we update the main memory. The line can be loaded into the cache (*write-allocate* or *fetch-on-write*), or not (*no-write-allocate* or *write-around*).

– If the desired information is in the cache, there are three methods:

- *Write-through*: this consists of writing the information into the memory while it is being written into the cache, or immediately after. It is possible to modify the system by inserting a first-in, first-out (FIFO) stack between the cache and the memory, so that the following operation can be undertaken before the writing into memory is performed.

- *Write-back* or *store-in* or *copy back*: the writing is done in the cache by setting an indicator bit called a *dirty bit*. The writing into memory is done when the position in the cache occupied by the line containing this information has to be made available. This method is, of course, much more powerful than the previous method. Indeed, if we increase the content of a counter in a program loop, it is more efficient to perform this operation only in the cache and not, at every iteration, in the cache *and* in the memory. This method presents a major drawback; however, when the memory is being accessed by several devices: this is the case with multiprocessor architectures and direct memory access. The same element of information could then have different values in one of the caches and in the memory (the data in memory could be "stale"). If we want to maintain coherence between caches and memory, a protocol must be in place to ensure it (see Chapter 12).

- *Posted write*: the write into memory is performed only when the cache controller has access to the bus. This method imposes less of a penalty for memory accesses than *write-through* and is safer than write-back.

REMARK 8.1.–

– Given the time necessary to update the cache from the memory, and vice versa, this operation can only be carried out by a *hardware device*.

– Because the processor is equipped with its *own cache access bus*, the external bus is often available during communication between the processor and the cache. As a result, a better *throughput* is possible if several processors, coprocessors and exchange units are connected to it. This is even more noticeable when the cache is managed using the *write-back* method, which is generally the case in a multiprocessor environment (Figure 8.2).

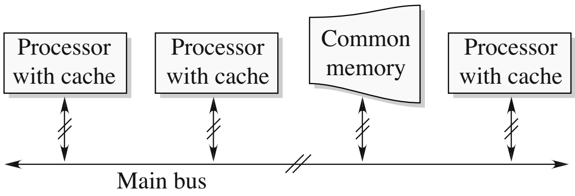


Figure 8.2. *Multiprocessor architecture and caches*

– It is possible to inhibit the operation of the cache memory. This is essential in a situation where the content of the memory has to be visualized to debug a program, or when updating an area of memory shared by several processors.

– Operation of the cache can be *frozen*, which results in a situation equivalent to having a large set of *read-only* registers. This feature can be used for processes that repeatedly use constant coefficients (fast Fourier transform, discrete cosine transform, etc.).

8.1.3. *Cache design*

8.1.3.1. *Fully associative caches*

The concept behind these caches is as follows: the address sent by the processor is divided into two fields - the tag on the most significant side and on the least significant side, the address of the information in the line (Figure 8.3). As a reminder, the "line" is the unit of transfer in exchanges between caches and the memory.

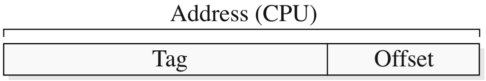


Figure 8.3. *Elements of the address*

During a fetch operation, the tag is compared to the address stored in the "content" (or *directory*) part of the associative memory. If there is a match, the processor directly accesses the information located in the cache. This fully associative design (Figure 8.4) is generally not implemented because of its cost: if the tag field is n bits long, producing such a cache would require an "n-bit" comparator for *each* of the lines in the cache (Figure 8.4).

EXAMPLE 8.1. –[The instruction cache of the TMS320C30]– The Texas Instruments® TMS320C30 microprocessor was specially designed for signal processing applications. It is equipped with a two-line, associative *instruction cache*, which is read-only. Each line is made up of 32 words of 32 bits each, in other words 32 *instructions*, and each word comes with a *presence bit P* (Figure 8.5).

The instruction addresses are 24 bits in length. The address memory therefore contains 19-bit words called *segment start address registers*. Cache management recognizes two kinds of *miss*:

– the *segment miss*, which corresponds to the case of the *miss* mentioned previously. All of the words are invalidated. In this particular case, the updates are performed *word-by-word* and not line-by-line;

– the *word miss*, when the line address is present in the cache, but with the presence bit set to 0. A single word is loaded into the cache, but the line is not invalidated.

8.1.3.2. *Direct-mapped caches*

Direct-mapped caches help achieve the goal of limiting the number of comparators. The tag is divided (Figure 8.6) into an *index* part and an *address-tag* part or *tag*. The index provides an entry for the directory. The content of this address is compared to the tag part of the address that was sent.

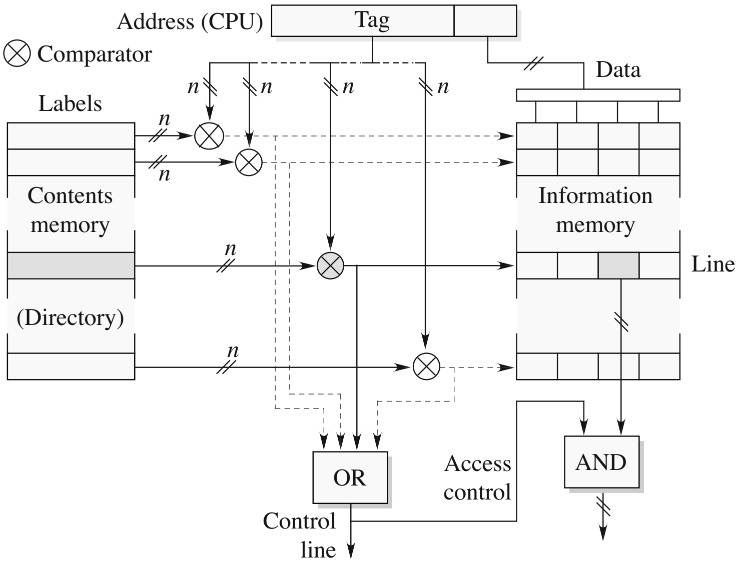


Figure 8.4. *Associative cache*

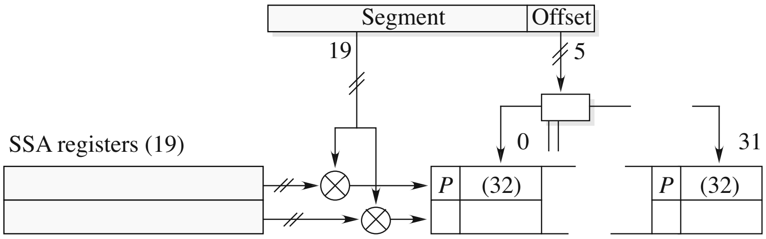


Figure 8.5. *Associative cache of the Taxas Instruments® TMS320C30*

This type of cache requires only one comparator. On the other hand, decoding the index creates an additional propagation delay. We will illustrate this concept with the example of the Motorola® 68020.

EXAMPLE 8.2.– [Cache in the 68020] The Motorola® M68020, a 32-bit microprocessor with 32-bit address buses, was created in 1984 and was equipped with a 256-byte instruction cache. A 6-bit index was used for addressing the memory, which contained $2^6 = 64$ entries. The *information* memory was used for storing 64 32-bit words (Figure 8.7).

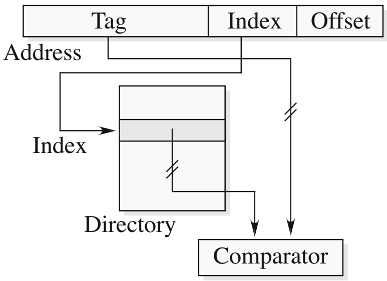


Figure 8.6. *Operating principle of direct-mapped cache*

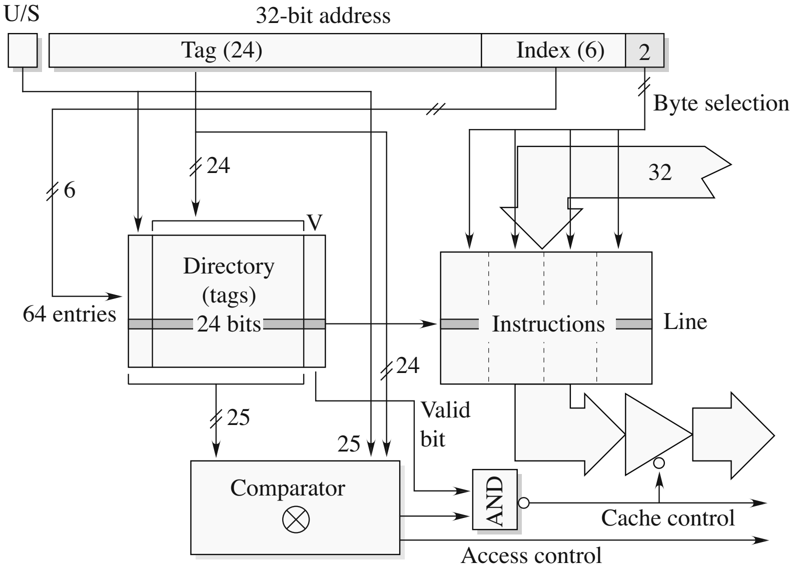


Figure 8.7. *Direct-mapped cache in the Motorola® M68020*

If, during a read operation, the instruction is not found in the cache, a 4-byte line is transferred from the memory to the cache. Note that in Figure 8.7, the comparison involves 25 bits, not 24. This is because certain accesses that are authorized in *supervisor* mode no longer are in *user* mode. The processor mode indicator (U/S bit in Figure 8.7) is taken into account by the comparator when triggering a possible exception consecutive to an unauthorized access.

8.1.3.3. *Set-associative caches*

The design of this *set-associative cache* combines the two above-described methods (Figure 8.8).

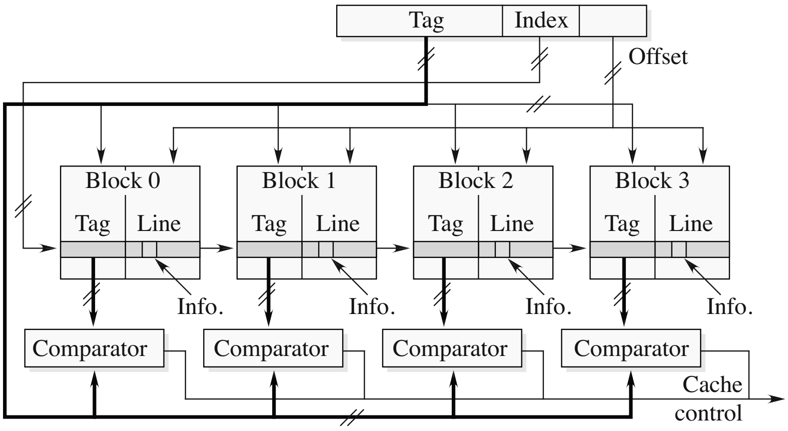


Figure 8.8. *Set-associative cache*

This is equivalent to using several direct-mapped caches set up in parallel. It is associative in the sense that the information selected in all of the lines are compared simultaneously to the tag part of the address. The set of lines with the same index is referred to as a *row* or a *superline*. The number of blocks is called the *associativity*.

Each *row* of a set-associative cache constitutes an associative cache.

EXAMPLE 8.3.– [The Intel® I80486 and Core^{TM}2 Duo] The internal cache in a version of the Intel® I80486 microprocessor consists of four sets (associativity of four) of 128 lines of 16 bytes (Figure 8.9). The Intel® E8500 has an up to 6-megabyte L2 cache with an associativity of 24- and 64-byte lines.

8.1.3.4. *Subset caches*

This design is very similar to the previous design. The difference lies in the fact that there is only one tag memory for a same set of lines.

The control logic uses both the tags and the valid bits associated with each line.

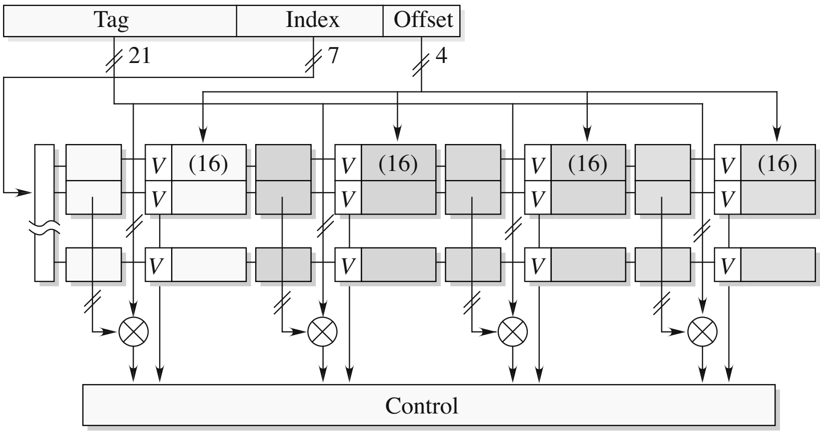


Figure 8.9. *Internal cache of the Intel® I80486 microprocessor*

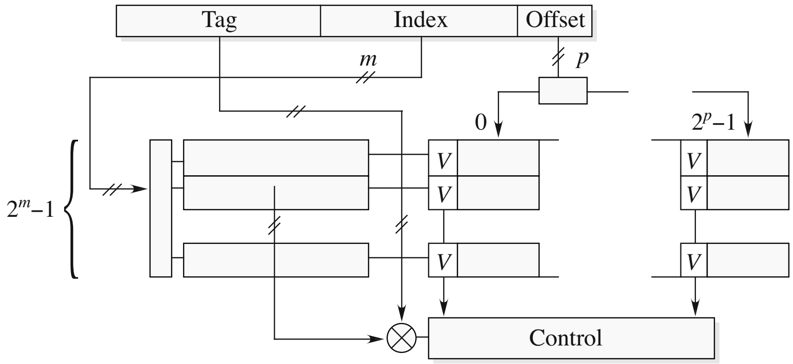


Figure 8.10. *Subset caches*

REMARK 8.2.–

– In the previous figures, an additional bit is located in the content memory, the *valid bit* or *presence bit*. Its role is as follows: a match between the address sent by the processor and the address stored in the content memory must only be taken into account if we are certain that the information sought is the right information. However, this may not be true in one of the following cases:

- during a write operation performed for an input–output device in direct memory access mode, where only the main memory is modified, there is a discrepancy between the contents of the cache and the memory,

- when the machine initializes, the cache content is random and should not be accessed,

- when designing programs and if the cache is "frozen", all of the entries are invalidated,

- when implementing *write-back* or *posted write* management policies, for which we saw that the information and their copy could be different in the cache and in the memory.

– Processors equipped with a *Harvard*-type internal architecture, now a classical architecture, have either two caches, one receiving instructions and the other receiving data, or a single instruction cache. Indeed, because the efficiency of a cache rests on the principle of its locality, it is reasonable to conceive that the concept of neighbors applies differently to instructions and data.

– The most common designs have caches that vary in size from 64 words (specialized microprocessors with integrated cache) to several kilowords, and even megawords on three levels. In the most recent microprocessors, the three levels of cache are integrated in the device.

8.2. Replacement algorithms

There are three main policies regarding the management of replacements in cache memory. We can:

– randomly pick the line to replace (*random policy*);

– pick the line that was loaded first (and is therefore the oldest) among all of the lines (*FIFO policy*);

– overwrite the "oldest" line, in the sense of when it was last accessed (*least-recently used* (LRU) policy). This is generally the policy used. We describe it in detail below.

8.2.1. *The LRU method*

Managing line replacements in associative or set-associative caches (for which we consider rows) requires the use of a stack, referred to as the LRU stack. It contains the line numbers arranged in order of relative age, by which we mean the time since their last use. The *most recently used* (MRU) line is located at the top of the stack, and the LRU line at the bottom. When a line is fetched, its number is moved to the top of the stack (Figure 8.11).

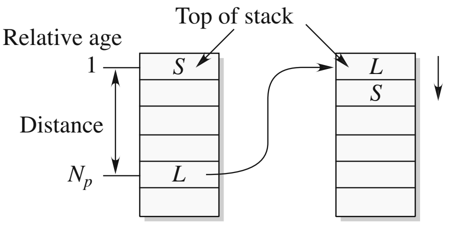


Figure 8.11. *The stack associated with the cache and its update mechanism when the line L is accessed*

This is known as the LRU method. The LRU stack constitutes a *cache behavior model* that assumes nothing about the effective design of the cache management logic. It can be summed up as follows:

– In case of a *miss*:

- the content of the line that was least recently referenced is replaced with the content read in memory,

- the number of this line is placed at the top of the stack, and all the other numbers are shifted.

– In case of a *hit*, the number for the line that was referenced is moved to the top of the stack.

EXAMPLE 8.4.– Consider the case of an associative cache consisting of four lines, 1, 2, 3, 4, where a, b, c, etc. denote the tags of the referenced addresses.

In Figure 8.12, the age management stack is represented vertically and the four lines of cache horizontally. We are interested in the behavior of this cache after it has received the initial sequence a, b, c, d. We assume that the following references make up the sequence b, a, e. The first reference b (tag b) triggers a *hit*. The number corresponding to the line, that is two, is moved to the top of the stack (Figure 8.12).

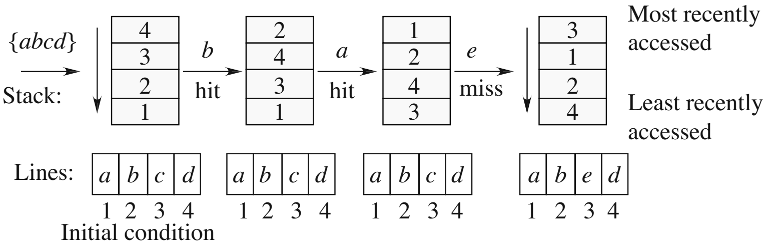


Figure 8.12. *LRU replacement*

This method has a drawback, though. Consider the access sequence with the following tags: a, b, c, d, e, a, b, c, f. Note that for the second sequence of references a, b, c, a memory access is necessary for every reference (Figure 8.13).

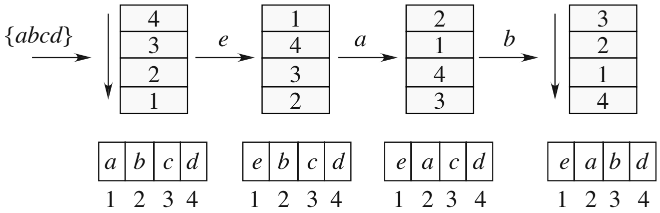


Figure 8.13. *LRU replacement, poor use of the cache*

An optimal replacement policy would require knowing the future. In the previous case, for example, it should not be necessary to replace a, b and c, but the LRU policy imposes a replacement for each new reference.

8.2.1.1. *Introducing a shadow directory*

A possible improvement[1] consists of introducing another directory, known as the *shadow directory*, which contains the tags that have been recently used but are no longer in the directory (Figure 8.14).

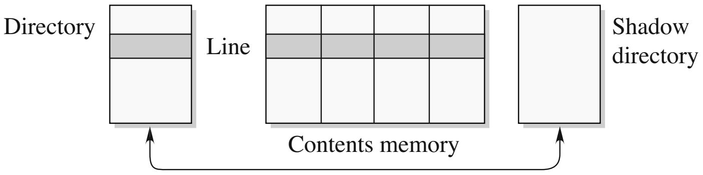


Figure 8.14. *Shadow directory*

When a new line is updated in the main cache (in case of a miss), the tag of the replaced line is introduced into the shadow directory. Misses can be divided into two categories:

– *transient misses*, in which the line is not in the cache and the tag is not in the shadow directory;

[1] Proposed by J. POMERENE, T.R. PUZAK, R. RECHTSCHAFFEN, and F. SPARACIO, “Prefetching Mechanism for a High-Speed Buffer Store”, cited in [STO 90].

– *shadow misses*, in which the line is not in the cache, but its tag is in the shadow directory. We can assume that there is a risk of this occurring again sooner or later. We therefore have an incentive to favor shadow references over those that lead to transient misses.

A bit indicates the type of *miss*. Several management policies can be implemented. If we consider once again the previous example and choose to overwrite lines corresponding to transient *misses*, we get the sequence illustrated in Figure 8.15.

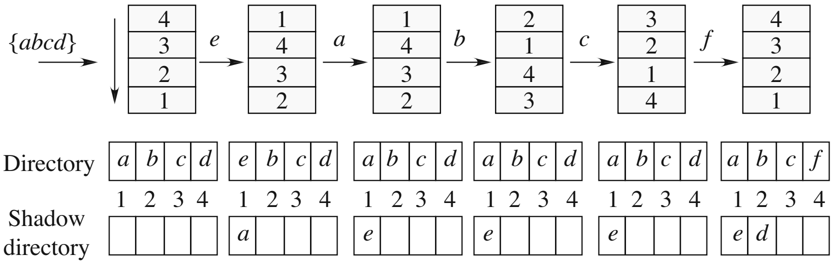


Figure 8.15. *Using the shadow cache*

8.2.1.2. *Using a "victim cache"*

When a line is going to be overwritten following a replacement, it is first copied into a small-size associative cache (Figure 8.16). This cache is called the *victim cache* [JOU 98].

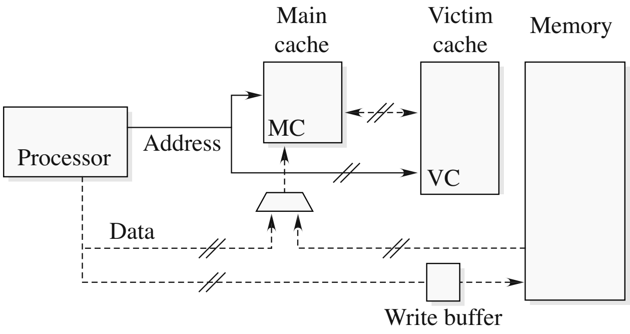


Figure 8.16. *Victim cache*

After a certain number of cycles, the victim cache (VC) contains mostly references to information that should not have been excluded from the main cache

(MC). These are called conflict *misses* [HIL 89], defined as *misses* that are avoidable either by modifying the associativity (*mapping misses*) or by modifying the replacement policy (*replacement misses*).

When the processor sends out an address, the corresponding tag is compared to the tags stored in MC and VC. In case of a hit in VC, the lines are switched (the LRU line in MC and the line found in VC).

8.2.1.3. *Hardware design of the LRU replacement policy*

There are essentially three implementation techniques:

– Using counters: each counter, referred to as an *age counter*, is associated with a line of the cache (Figure 8.17).

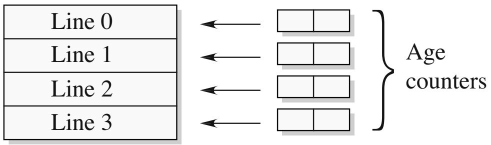


Figure 8.17. *Age counters*

If the cache consists of 2^n lines, there are 2^n of these counters, and they are equipped with n binary elements. If a reference is satisfied in the cache (hit), the associated counter, the content of which is assumed equal to p, is set to zero, while all the other counters with content less than p get their content increased. Figure 8.18 shows how the counter management logic works.

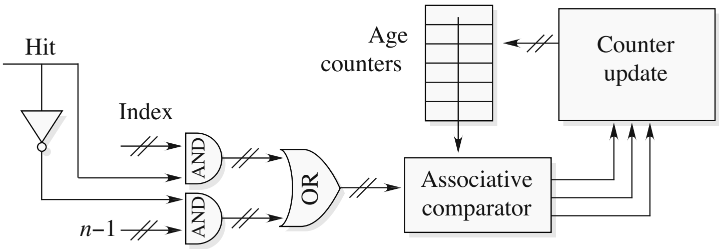


Figure 8.18. *Updating age counters*

The line with the counter containing the value $2^n - 1$ is the oldest line. In order to find within a sufficiently short response time, we can perform an associative search. In the case where the cache is not full, the new line is loaded and its counter initialized at zero, while all other counters are incremented.

– Using a hardware stack: this stack contains the line numbers arranged in order from the most recent reference at the top of the stack (on the left in Figure 8.19), to the oldest at the bottom of the stack (on the right in Figure 8.19). If there are 2^n lines, there must be 2^n n*-bit* registers. The line number p is set at the top of the stack and the appropriate shift is performed. This requires the presence of comparators for generating the shift clocks. In this way, all the registers from 0 to $p - 1$ are shifted in the stack. Since the oldest element ends up at the bottom of the stack, decoding it leads to the immediate identification of the line that needs to be used.

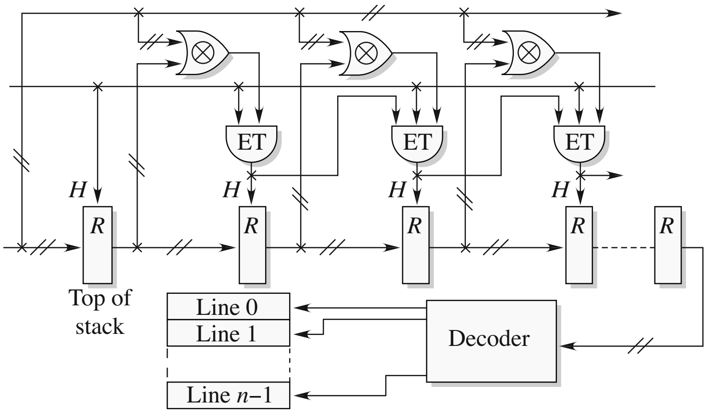


Figure 8.19. *The hardware stack*

In case of a miss, all the elements are shifted and the line number p is written at the top of the stack. In case of a hit, there is a partial shift, as indicated in the figure, and the line number p is written at the top of the stack.

– Using a state matrix associated with the cache: this matrix contains Boolean elements denoted by 0 and 1. When a line p is referenced, line p of this matrix is set to 1 and the column p to 0.

EXAMPLE 8.5.– [Using a state matrix] Consider a four-line cache initialized by a sequence $\{a, b, c, d\}$, which fills up its lines in the order 0, 1, 2, and 3. *After these four accesses*, the line k of the state matrix A contains k 1's. Let us assume a new access to the element b (hit). The matrix is modified as follows:

$$A = \begin{array}{|c|c|c|c|c|c|}\hline \text{Line} & & & & & \\ \hline 0 & 0 & 0 & 0 & 0 & a \\ \hline 1 & 1 & 0 & 0 & 0 & b \\ \hline 2 & 1 & 1 & 0 & 0 & c \\ \hline 3 & 1 & 1 & 1 & 0 & d \\ \hline \end{array} \xrightarrow{hit} \begin{array}{|c|c|c|c|c|c|}\hline \text{Line} & & & & & \\ \hline 0 & 0 & 0 & 0 & 0 & a \\ \hline 1 & 1 & 0 & 1 & 1 & b \\ \hline 2 & 1 & 0 & 0 & 0 & c \\ \hline 3 & 1 & 0 & 1 & 0 & d \\ \hline \end{array}$$

Starting with the same initial state, let us now assume a miss (we access the element e). Since the LRU line is the line 0, the state we obtain is:

Line					
0	0	0	0	0	a
1	1	0	0	0	b
2	1	1	0	0	c
3	1	1	1	0	d

$\xrightarrow{miss}$

Line					
0	0	1	1	1	e
1	0	0	0	0	b
2	0	1	0	0	c
3	0	1	1	0	d

This way, the number of the most recent line is given by the matrix line that contains the highest number of 1's, while the oldest is given by the line containing nothing but 0's.

This method is better suited for small caches. Large caches are divided into subsets that do not overlap, and the replacement policy is implemented on two levels: the first level manages subsets and the second level manages the content of the selected subset.

8.2.2. *The case of several levels of cache*

When several levels of cache are used, it is possible to define behaviors for each level.

8.2.2.1. *Inclusive caches*

An *inclusive* cache contains all of the information in lower level caches. The Intel® i7 microprocessor, for example, has three levels of cache. The L3 cache contains a copy of the data in the lower level caches L1 and L2. L3 is described as *inclusive*. Note that updating a line only requires testing the L3 cache. On the other hand, if we need to replace a line in L1 or L2, we must also replace it in L3.

An inclusive cache operates exactly as a simple memory hierarchy. In the case, for example, of a cache with two levels L1 and L2, if there is a miss during a read in L1 or L2, then L2 is updated from the memory, and L1 from L2. If there is a miss in L1 and a hit in L2, then L1 alone is updated from L2. This mode of operation is helpful in managing information coherence in the involved caches.

8.2.2.2. *Exclusive caches*

Another mode of operation consists of imposing as a rule that an element of information can only be held by a single cache. These are *exclusive* caches. This mode is used, for example, in the AMD Athlon$^{\text{TM}}$ Thunderbird microprocessor.

Figure 8.20 illustrates the architecture of a two-level exclusive cache system.

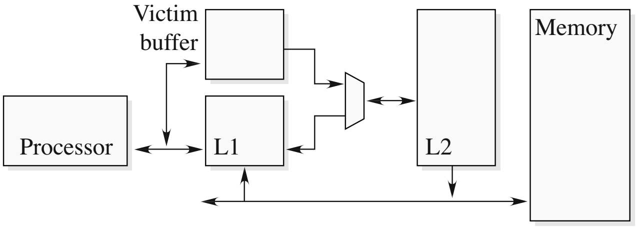


Figure 8.20. *Exclusive cache with two levels L1 and L2*

A buffer known as a *victim buffer* (VB) makes it easier to handle exchanges between L1 and L2 for the purpose of maintaining their exclusive nature. Let us assume that the line size is the same for L1 and L2. Unlike the victim cache, the victim buffer only holds to information for the smallest amount of time possible. Transfers to L2 are performed as soon as possible.

In the two-level case, a hit in the L2 cache and a miss in L1 leads to an *exchange* of lines between the two caches rather than a simple transfer from L2 to L1: the victim line is transferred to L1 via VB, the addressed line is transferred from L2 to L1, then the exchange is completed by an update a L2 from VB.

If there is a miss in the two caches L1 and L2, the corresponding line is brought directly from the memory to L1 with a transfer, if necessary, from L1 to VB.

8.2.3. *Performance and simulation*

The elements that need to be taken into account when evaluating the performances of a cache are the associativity, the line size, the number of bits of the index, the type of write management, the specific characteristics (write protection when "caching" non-volatile memory, coherence management, exclusivity, etc.) and the access time.

Consider a set-associative cache (Figure 8.21). The address can be divided as follows:

[Tag, Index, "number of words in the line"]

Here, the number of blocks is denoted by B, the number of lines per block by L and the number of words per line by M. The total number of words is $B \times L \times M$. The performance of the cache depends on these three parameters. The rule of thumb states that doubling the cache size leads to a 30% drop in the number of misses.

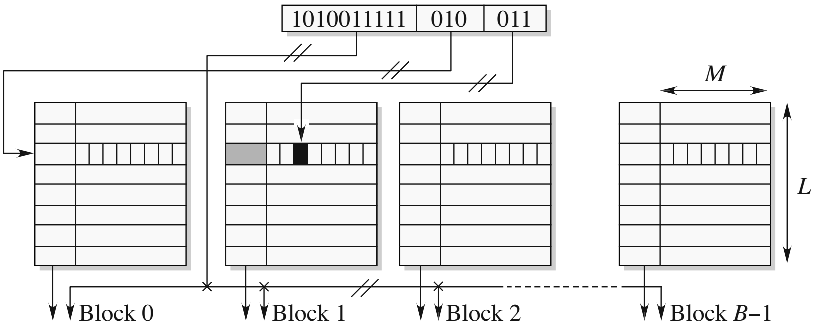


Figure 8.21. *Set-associative cache*

The choice of a cache structure must be done as follows: for a given cache size, we first set the number of blocks B. We then change L and M and count the number of *misses*.

The simulator input is typically a set of programs. Each time an instruction is executed, the sequence of the different memory references that have occurred is written into a file (in some cases, if the machine allows a step-by-step operation, the operating system can perform this task).

The Intel® Corp. released some statistical data (Table 8.1) based on figures relative to – old - mainframes. The size of these tags is not disclosed, but the results shown provide interesting indications. These caches are direct-mapped (associativity of 1) or set-associative.

Size (kB)	Associativity	Line (bytes)	Hit rate (%)
1	1	4	41
8	1	4	73
16	1	4	81
32	1	4	86
	2	4	87
	1	8	91
64	1	4	88
	2	4	89
	1	8	92
128	1	4	89
	2	4	89
	1	8	93

Table 8.1. *Some figures on caches (Source: Intel® Corp.)*

There are two problems with these simulations:

– We can never be sure that the workload imposed is representative of what is actually occurring. Note that test programs can consist of millions of instructions. Since the simulation is roughly 1,000 to 100,000 slower than the execution of the program we are studying, this poses major feasibility issues.

– The second problem lies in the fact that the simulation must be long enough that it is not affected by the transient part corresponding to the beginning of the program, when the number of *misses* is particularly important.

REMARK 8.3.– A particular form of cache use is *trace cache* [ROT 96, PAT 99], which stores *sequences* of instructions that have been executed. This type of cache, which was first introduced in the VAX machines by Digital Equipment (*fill unit*) [MEL 88] to lighten the load of the decode stage, was used by Intel® in its Pentium® IV. The trace cache is built in parallel with the instruction cache, but provides the result of the decoding of instructions, rather than the instructions themselves. A similar mechanism is used in the Intel® i7 processor, which has a cache for storing microinstructions (*μop-cache*).

Chapter 9

Virtual Memory

Virtual memory is a management technique that relies on the combined use of the main memory and a mass storage unit. This unit, which usually consists of a magnetic disk, is used as a complement to the main memory. Even though the main memory can be quite large in practice (consumer microcomputers are theoretically capable of addressing up to 4 GB with their 32-bit address buses, and the upgrade to 64 bits is becoming widespread), economic considerations make it necessary to use a part of the disk space to “simulate” the presence of electronic memory. Besides, this management concept fits naturally into the *hierarchy* (section 7.3) of a memory management system where mass storage, main memory, cache and registers form the building blocks of the memorization system.

The first attempt to implement such a device was for the ATLAS computer [KIL 62, SUM 62] at the University of Manchester in the early 1960s. This approach was referred to as a *one-level store*. The memorization system comprised a 16 kilowords random access memory and a magnetic drum storage space with a capacity of 96 kilowords. This approach was equivalent to offering the user a total memory of 96 kilowords divided into blocks or *pages* of 512 words. The ratio between the access times for the drum and for the main memory was in the order of 500. A *learning program* ensured a completely seamless user experience and guaranteed that the *most commonly used* pages were kept in the physical memory to optimize their accessibility.

All modern general use microprocessors offer this feature. The variety of solutions is comparable to the variety of architectures they are implemented on.

9.1. General concept

9.1.1. *Operation*

In a virtual memory management system, the user perceives the set consisting of the *main memory mass storage* as a single, large size space (Figure 9.1).

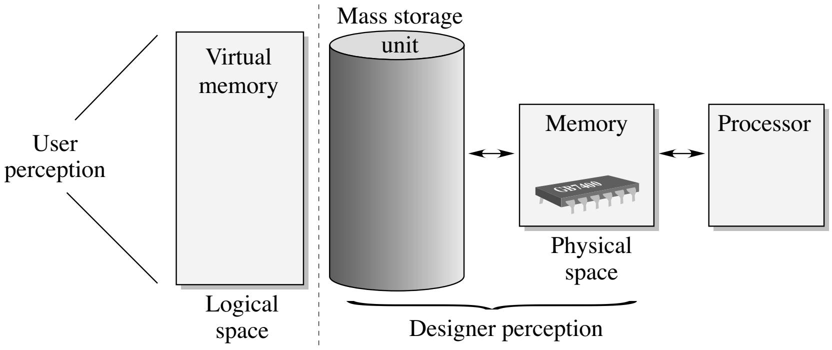


Figure 9.1. *How memory is perceived*

A mechanism for transferring blocks of information, known as *pages*, between the mass storage unit and the physical memory, *implemented by the operating system*, ensures that the user does not have to worry about the actual location of the information they are handling. The data are stored in rows and the programs are written as if the available main memory had a size equal to the set (or a subset) of the storage units. The only inconvenience for the user is caused by the delay suffered by some of the responses to requests, delays caused by the transfers between the mass storage unit and the main memory.

9.1.2. *Accessing information*

Figure 9.2 summarizes the sequence of operations performed to carry out a fetch operation in the context of virtual memory management. The steps are the following ones:

– The processor sends out an address. The translation mechanism indicates the presence or absence of the desired information in the main memory of the page.

– If the page is present, the information (a) is immediately accessed.

– Otherwise, the page containing this information must be transferred from the mass storage unit to the main memory. If necessary, some space has to be cleared.

However, this space may be taken up by a page that has been modified since its transfer to the main memory:

- if that is the case, this page is copied into the mass memory unit ($c1$). Then, the desired page is transferred to the main memory ($c2$) and access is achieved ($c3$);

- otherwise, the desired page is transferred into memory ($b1$) without saving any information prior to access ($b2$).

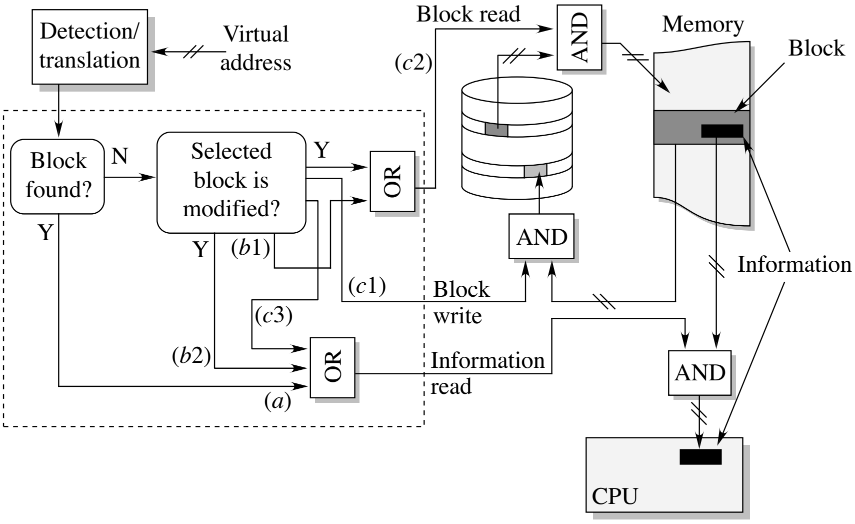


Figure 9.2. *Virtual memory management*

9.1.3. *Address translation*

Since the generated addresses are no longer *physical addresses* in the main memory, they are referred to as *virtual* or *logical addresses*. The management system ensures the *translation* of these virtual addresses into physical addresses, in other words switching from the logical space experienced by the user to the physical space. This translation relies on detecting the presence of the desired information in memory. If this information is not found in the memory, we have a *fault*. The system must search the mass storage unit for the page containing the information, as described in the previous example.

The virtual and physical spaces are divided into blocks of the same size, known as *pages* (*virtual pages* and *page frames*), which act much like *lines* do in a cache. The address of the information then consists of a page number and of the *offset* of the word in the page (Figure 9.3).

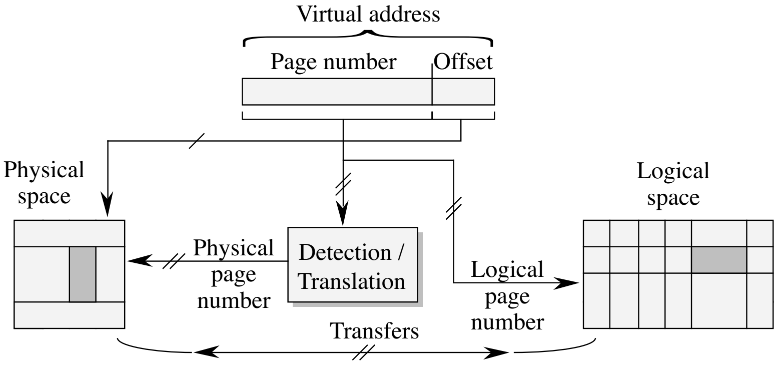


Figure 9.3. *Virtual and physical spaces*

A virtual memory management system ensures the translation of a virtual page number (VPN) into a physical page number (PPN). An VPN can only have one corresponding PPN. On the other hand, a PPN can correspond to several VPNs.

The translation device is purely hardware. In practice, translation tables are kept in memory with a cache system such that access times are compatible with the desired performance. The detection/translation device relies on a cache called the *translation lookaside buffer* (TLB), the *translation buffer* (TB), or the *address translation cache* (ATC).

Despite sharing the same major concepts, there are important differences to note between translation cache and conventional memory cache:

– the information collected in the translation cache is not the information itself, but the number of the physical page that contains it and that will be used by the cache memory (see example, section 9.3);

– the page search within the storage unit is managed by the operating system, not by the hardware.

9.2. Rules of the access method

9.2.1. *Page fault*

During the execution of a program, the processor generates virtual addresses that are sent to the memory management module. The memory management module (the translation cache and the page table) indicates whether the generated address corresponds to an address in the main memory:

– if that is the case, the translation from virtual address to physical address is performed immediately, and the desired information is fetched;

– otherwise, the memory management module sends an *abort* signal to stop the execution of the instruction. This is called a *page fault*.

A page fault is fundamentally different from an interrupt: the instruction is supposed to *revert* to its initial state instead of stopping. The registers that were altered by the execution must be restored so that, once the page fault is handled, it is possible to restart the instruction from the beginning. The address saved in the stack is not the address of the instruction, but of the instruction that triggered the fault.

The mechanism for obtaining the address of the page fault handling program is similar to the one used for interrupts: after saving the address of the instruction responsible for the fault, the processor retrieves the address of the handling program in the interrupt vector table. This mechanism is inhibited when the aborted cycle corresponds to a prefetch instruction (this is the case in pipeline architectures, see Chapter 10), because it is not known whether the instruction in question will be executed, and it is therefore useless to start the exception handling process. Only the prefetch will be canceled.

9.2.2. *Multi-level paging*

The page table can take up considerable memory space. Consider a machine with a 32-bit address space: if the page size is 1 kiloword, and if we have $d = 10$ offset bits, the page number is coded in 22 bits. The page table must therefore have access to 2^{22} words, or 4 megawords, to store the physical addresses of the pages. If each PPN comes with service bits and takes up 32 bits, the size of the table will be 16 MB. This explains why processor designers had to introduce an additional level of tables, where the first level is stored in memory, and the second level table (or tables) can be partially stored on a disk. The first level table is called the *page directory*, *root table page*, or *hyperpage table* in operating system terminology. The virtual address is divided into three fields (Figure 9.4).

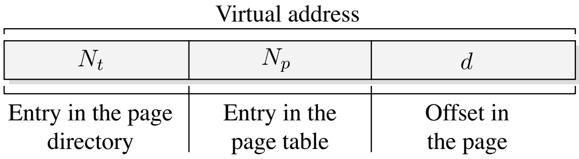


Figure 9.4. *A virtual address in two-level paging*

The N_t most significant bits provide an entry in the *page directory*, which in turn provides the base address of a page table. The address of the directory permanently stored in memory is usually provided by the content of a register called the *directory register*. The next N_p bits, combined with the address provided by the content of the directory, give an entry in the page table (Figure 9.5). For performance reasons, the directory is kept in memory.

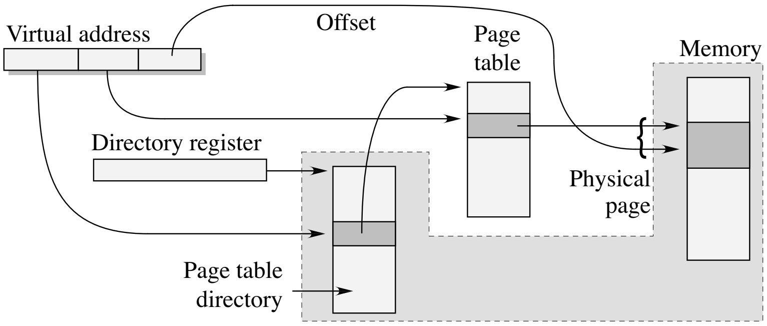


Figure 9.5. *Two-level paging*

Virtual memory management thus applies to page tables as well as pages. Only the first level table (the page directory) is kept in memory (Figure 9.6).

9.2.3. *Service information, protection and access rights*

Service bits are associated with each page. These bits are used for protecting data and programs, and for the management of memory by the operating system:

– The *modify bit* M, or *dirty bit*, indicates if the page has been modified. It is used by the operating system to know if the page needs to be copied into the mass storage unit before being replaced. It is, of course, useless to save a portion of memory that has not been modified.

– The *valid bit* V is used in the same way as in cache memory. It helps avoid accessing pages that are referenced in a table but have not been transferred to memory yet. This can occur:

- upon initialization of the machine, when the content of the memory is uncertain;

- when a page in memory is copied onto a disk to save space. Its page number is replaced by the number of the disk block it was loaded into. The corresponding *V bit* is then set to the invalid position.

– The *reference bit* R is used by the operating system to estimate how frequently the page is used. Each access to the page sets this bit to 1. It is also periodically set to 0. The pages with their R bit set to 0 are referred to as LRU. This makes it possible to implement an algorithm such as this one: if a page is rarely used, or has not been used in a "long" time, it will be more likely than others to be replaced if the system needs space in the main memory. Not all processors are equipped with this service bit. Operating systems that require them must then simulate them using the M bit or the V bit.

– each entry in the page table and in the TLB includes bits for managing page access rights (for reading, writing, and executing), and, often, bits giving the number of the process owning the page.

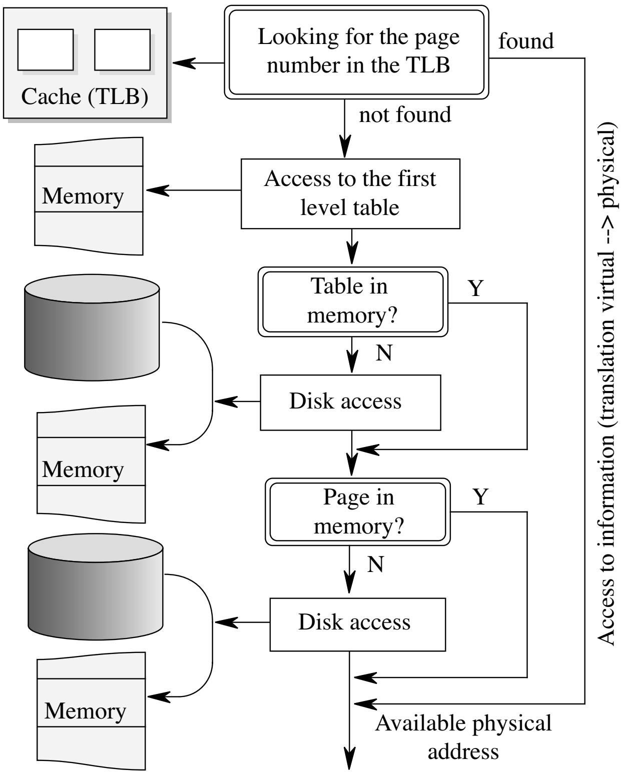

Cache (TLB)
Looking for the page number in the TLB
found
not found
Memory
Access to the first level table
Table in memory?
Y
N
Disk access
Memory
Page in memory?
Y
N
Disk access
Memory
Available physical address
Access to information (translation virtual --> physical)

Figure 9.6. *Accessing information in a two-level paging system*

9.2.4. *Page size*

Page size is a major factor for performance in a virtual memory management system. It is chosen based on conflicting criteria:

– The smaller the page size, the larger the description tables have to be, hence the need to have a somewhat large page size.

– Because the size of programs or data structures is not always an exact multiple of the page size, there will always be wasted space (a phenomenon known as *internal fragmentation*). This makes a small page size advantageous.

– The smaller the page size, the shorter the transfer time between the mass storage unit and the main memory. We should not, however, conclude that the page size must be as small as possible, since the size of the transferred block is not proportional to the transfer time: the time it takes for the read–write heads of the disk to position themselves is often longer than the transfer itself. It can sometimes be quicker to transfer one 2-kB block than two 1-kB block.

– The locality principle favors a small page size: this is because at a given time, we do not need the entire program in the main memory.

– If the pages are too small, there will be many transfers from the mass storage unit to the main memory, in other words considerable time lost due to the saving of contexts and to the execution of the page replacement algorithm.

– Another factor in our choice is the relationship between the size of the physical sector on the mass storage unit (the physical unit for the transfer) and the page size (the logic unit for the transfer).

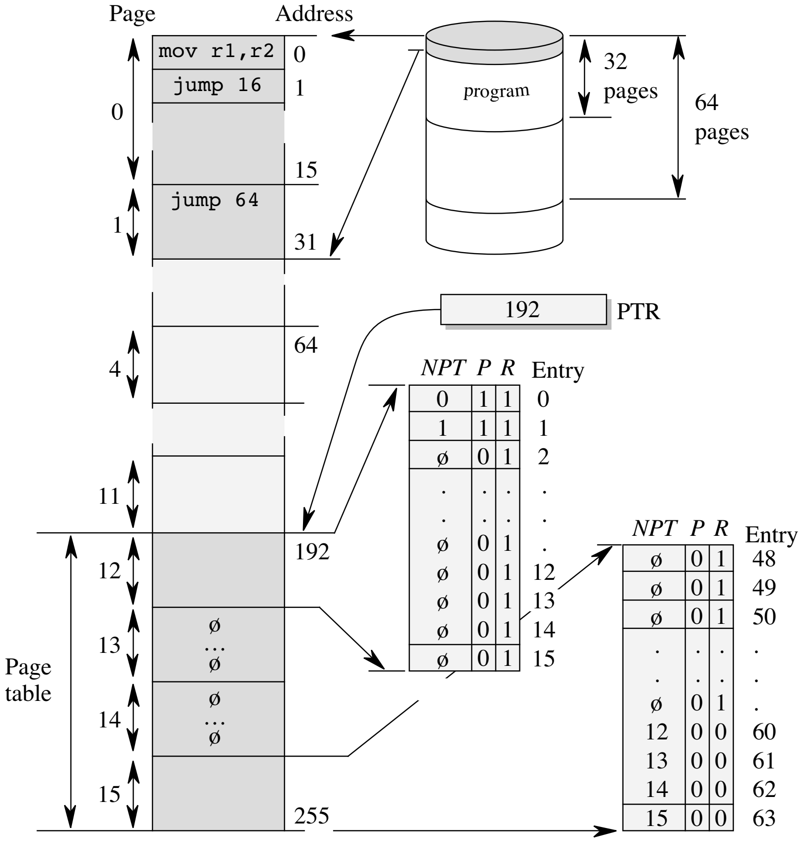
In practice, the page size is chosen an equal to one to eight times the size of the disk sectors, or equal to the disk allocation unit (the granularity or *cluster* from the point of view of the operating system). The page size in Windows or Mac OS X is usually 4 kB. It is related to the granularity parameter in the processor's memory management system.

9.3. Example of the execution of a program

For the purpose of making it easier to understand the mechanisms, we are dealing with, we will discuss the example of a machine equipped with 256 words of physical memory organized in 16 word pages, or 16 pages of 16 words and a mass storage unit with 1,024 words, or 64 pages of 16 words (Figure 9.7), set aside for the user's needs.

The virtual addresses sent by the processor are coded in 10 bits, making it possible to address 1,024 words. The instructions are all coded in one word.

Consider a program that takes up 512 words of memory (32 pages in the mass storage unit). The first instruction `mov r1,r2` is an instruction to move from one register to another. The next `jump 16` is a branch to the word with the address 16 and, at address 16, we find another branch `jump 64`. To manage the translations, the operating systems store in memory a table with 64 entries corresponding to the 64 virtual pages. Each entry contains a PPN and a presence bit P indicating whether the page is in memory ($P = 1$) or not ($P = 0$). The page table address is stored in the 8-bit PTR register (page table register).



NPT	*P*	*R*	Entry
0	1	1	0
1	1	1	1
ø	0	1	2
.	.	.	.
.	.	.	.
ø	0	1	.
ø	0	1	12
ø	0	1	13
ø	0	1	14
ø	0	1	15

NPT	*P*	*R*	Entry
ø	0	1	48
ø	0	1	49
ø	0	1	50
.	.	.	.
.	.	.	.
ø	0	1	.
12	0	0	60
13	0	0	61
14	0	0	62
15	0	0	63

Figure 9.7. *Example: the first two pages of the program are loaded into memory (physical pages 0 and 1)*

Initially, the first two pages of the program are loaded into pages 0–1 of the physical memory. The other entries have their presence bit set to 0. The physical pages numbered 12 through 15, which contain the page table we wish to permanently store in memory, must not be used for other purposes. They are referenced in the virtual space with a page number from 60 to 63. The corresponding $R = 0$ bit indicates that these pages are read-only. We could add information (an operation mode bit for example) but, for reasons of simplicity, we will limit ourselves to just the R bit.

9.3.1. *Introducing the translation cache*

Let us assume that the translation system is a *fully associative cache* with four entries. Its state is shown in Figure 9.8.

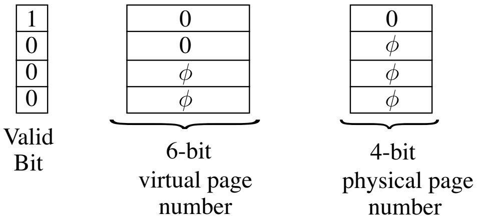


Figure 9.8. *Initial state of the translation cache*

The machine is equipped with a fully associative cache memory with 4-word lines. Initially, all of its entries are marked as *invalid*. The organization of the memory management is illustrated in Figure 9.9.

We will adopt the following notations:

– for each TLB entry, V_T is the valid bit;

– for each cache entry, V_c is the valid bit;

– *NL* is the number of the logic page (virtual page);

– *NP* is the number of the physical page in the TLB;

– *np* is the number of the physical page in the cache;

– for each entry in the page table, P is the presence bit;

– *NPT* is the number of the physical page in the *page table*.

9.3.2. *Execution*

After loading (Table 9.1) two pages into memory, the page table is updated accordingly and we assume that a previous access validated the first entry in the TLB.

1) When the memory is first accessed, the program counter is set to 0. The translation module TLB that contains the corresponding page number (page 0) with a valid bit V_T equal to 1 returns a hit.

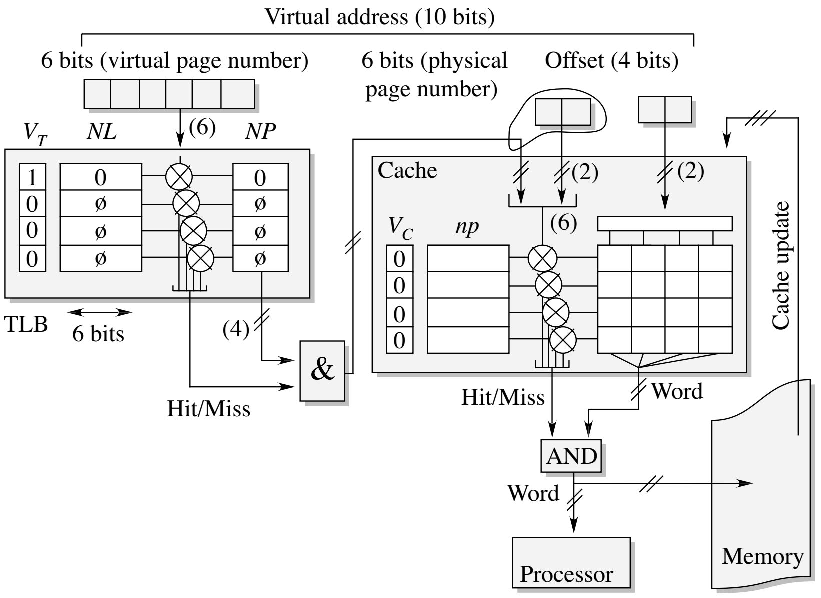


Figure 9.9. *Organization of the memory management (example)*

Reading and execution of `mov`											*NPT*	*P*	
											0	1	0
V_T	*NL*	*NP*		V_c	*np*	lines					1	1	1
1	0	0		0	0	ϕ	ϕ	ϕ	ϕ		ϕ	0	
0	ϕ	ϕ		0	ϕ	ϕ						ϕ	
0	ϕ	ϕ		0	ϕ	ϕ							
0	ϕ	ϕ		0	ϕ	ϕ							63
TLB				*Cache memory*							*Memory*		

Table 9.1. *Initial conditions*

Reading and execution of `mov`											0	1	0
Cache update											1	1	1
1	0	0		1	0	`mov`	`jump 16`	`xxx`	`xxx`			ϕ	0
0	ϕ	ϕ		0	ϕ	ϕ						ϕ	
0	ϕ	ϕ		0	ϕ	ϕ							
0	ϕ	ϕ		0	ϕ	ϕ							63

Table 9.2. *Cache memory update*

The PPN that is delivered is 0. On the other hand, the cache memory will return a miss because of the value of its valid bit V_C. The words with addresses 0 through to 3 are transferred to the cache and the line is set to *valid*. The instruction being read can be transferred to the instruction registered to be handled (Table 9.2).

2) At the second access, the program counter takes on the value 1. The page number is still 0. The translation system returns a *hit* and the physical page number is 0. Since the cache that contains the information we need, the instruction `jump 16`, the instruction fetch is "instantaneous" (Table 9.3).

Reading of the branch instruction									0	1	0
TLB hit			CM hit and reading of `jump 16`						1	1	1
1	0	0	1	0	`mov`	`jump 16`	`xxx`	`xxx`	ϕ	0	
0	ϕ	ϕ	0	ϕ	ϕ					ϕ	
0	ϕ	ϕ	0	ϕ	ϕ						
0	ϕ	ϕ	0	ϕ	ϕ						63

Table 9.3. *Referenced information*

3) The attempt to read the content of the address 16 (located on page 1) results in a miss in the TLB. On the other hand, this page is, in fact, referenced in the page table (Table 9.4). The processor fetches it without triggering a fault because the presence bit is set to 1. The processor copies the page number *NPT* into the translation cache (through an LRU algorithm).

Execution of the branch									0	1	0
TLB update			CM miss and reading of `jump 64`						1	1	1
1	0	0	1	0	`mov`	`jump 16`	`xxx`	`xxx`	ϕ	0	
1	1	1	0	ϕ	ϕ						
0	ϕ	ϕ	0	ϕ	ϕ						
0	ϕ	ϕ	0	ϕ	ϕ						63

Table 9.4. *Case of a TLB miss with correct page table*

There is then a miss at the cache level, which is now updated too (Table 9.5).

									0	1	0
TLB update			CM update and reading of `jump 64`						1	1	1
1	0	0	1	0	`mov`	`jump 16`	`xxx`	`xxx`	ϕ	0	
1	1	1	1	1	`jump 64`	`xxx`	`xxx`	`xxx`		ϕ	
0	ϕ	ϕ	0	ϕ	ϕ						
0	ϕ	ϕ	0	ϕ	ϕ						63

Table 9.5. *Handling the first two misses*

4) During the execution of the `jump 64` instruction, the page number that is sent is equal to four. The TLB shows a miss and, since the presence bit in the page table is

equal to one, there is a *page fault*. The corresponding address has to be saved so we can come back to it after handling the page fault. This save is often done in a special register and not in the stack. The operating system is then put in charge, and tasked with fetching from the disk the virtual page 4. If we assume that the physical page that is chosen is the third one, and that this page was not modified in the meantime, there is no reason for saving it first (Table 9.6).

TLB update											0	1	0
											–	–	–
1	0	0		1	0	`mov`	`jump 16`	`xxx`	`xxx`		3	1	4
1	1	1		1	1	`jump 64`	`xxx`	`xxx`	`xxx`			ϕ	
0	ϕ	ϕ		0	ϕ	ϕ							
0	ϕ	ϕ		0	ϕ	ϕ							63

Table 9.6. *Handling the exception*

Since the address responsible for the fault was saved, the restart process updates the TLB. This is the case because when the operating system yields control, the execution of `jump 64` causes a *TLB miss* which is then resolved the same way as previously, since the presence bit is now set to one (Table 9.7).

TLB update											0	1	0
											–	–	–
1	4	3		1	0	`mov`	`jump 16`	`xxx`	`xxx`		3	1	4
0	ϕ	ϕ		1	1	`jump 64`	`xxx`	`xxx`	`xxx`			ϕ	
0	ϕ	ϕ		0	ϕ	ϕ							
0	ϕ	ϕ		0	ϕ	ϕ							63

Table 9.7. *Updating the TLB*

9.3.3. *Remarks*

1) In the case where we rely on a paged organization, only one block of a fixed size – the *page* – is loaded into memory. Since the pages all have the same size, the management of the memory space is considerably simpler compared with segmentation. The *replacement* of pages does not imply any reorganization of the memory.

The criteria that determine page replacement vary from system to system: least commonly used pages, oldest in memory, etc. As a general rule, the operating system tries to keep in memory the most used pages (the phrase *working set* is sometimes used to refer to the set of pages of a process that have this property).

Virtual memory management relies on this *paging–transfer* association.

2) Updating the translation cache poses a more difficult problem than in the case of instruction/data caches. This is because in a multiprogramming context, two virtual addresses can be identical in the address memory of the TLB [CLA 85, STO 90]. We could add a *task identifier* (TID) to the TLB (see exercise 9.1).

- A first solution consists of invalidating the entire content of the TLB after a page fault, or after the operating system resets the table page *bit P* to zero. This is the case for example for the early versions of the Intel® 386 microprocessor [INT 85].

- A second solution consists of invalidating only part of the entries involved. This second solution requires having access instructions to the lines of the TLB. This is the case of the AMD29000 microprocessor [ADV 87]. After a *TLB miss*, an exception called a *TLB trap* is generated. The memory management module then provides in the *recommendation register* the TLB entry number corresponding to the LRU line. The address responsible for the trap is also saved in a register so that the operating system can update the TLB using the `mttlv` (*Move to TLB*) instruction and page table stored in memory.

9.4. Example of two-level paging

To illustrate the mechanisms used in virtual memory management systems, we will discuss an example of the practical implementation of the *memory management modules*. This example is based on the National Semiconductor Corporation's NS32000®, in which the virtual memory and the physical memory have the same size.

9.4.1. *Management*

9.4.1.1. *Concepts*

This first example focuses on the design of the memory management module in the 16- and 32-bit National Semiconductor [NAT 84] microprocessors, which date back to the 1980s (Figure 9.10).

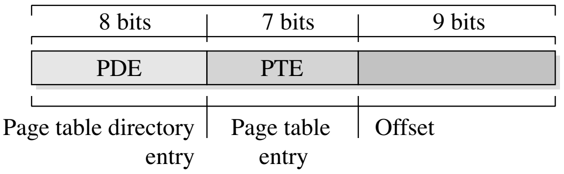


Figure 9.10. *Virtual address structure*

The processor sends a 24-bit address. The size of a page is 512 bytes (nine offset bits) and the page number is 15 bits long. As we saw previously, if we opt for one-level paging, the virtual page table will need to hold 2^{15} entries. The solution actually implemented is *two-level paging*: the page size is still 512 bytes and the VPN is divided into two fields of eight and seven bits.

The field made up of the eight most significant bits, known as the *page directory entry* (PDE), provides one of 256 entries in a table called the *page directory* (PD), which holds the addresses of the page tables. The address of this table, which is permanently stored in memory, is built based on the content of a register called the *page table base* (PTB). The following 7-bit field, known as the *page table entry* (PTE), combined with the 15 bits provided by the page table directory, is one of 128 entries in the page table. The nine least significant bits indicate the offset in the page specified in the page table "up to" the desired bit. Each PTE provides a 15-bit PPN as well as a series of service bits. Since the memory is organized in 32-bit words, then without the page directory, the page table would permanently take up 2^{15} words, or 128 kilobytes (Figure 9.11).

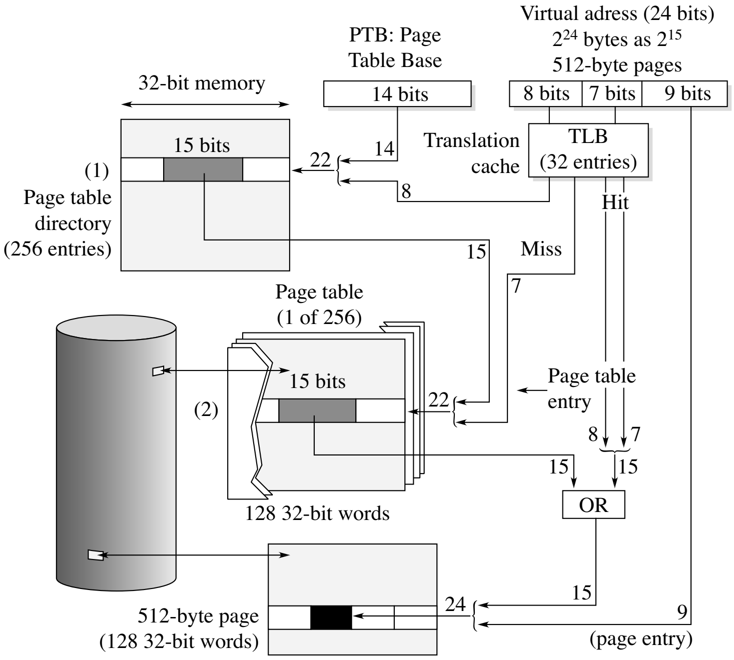


Figure 9.11. *Two-level paging*

9.4.2. *Handling service bits*

The entries of the page directory (word (1) in Figure 9.11) and of the page table (word (2)) are written in the format shown in Figure 9.12.

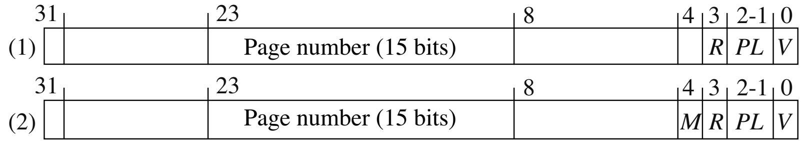


Figure 9.12. *Entry format in the page directory and in the page table*

V is the *valid bit*, M the *dirty bit* or *modify bit*, R the *referenced bit*, and *PL* the 2-bit privilege level.

The current privilege level is determined based on the read/write signals, the operating mode (user/supervisor), and the *access override bit*. Depending on the value of this bit, the (MMU) will or will not verify the access rights when the processor is in user mode.

The M (*modified*), R (*referenced*) and V (*valid*) bits are used as shown in the flowchart in Figure 9.13.

Access right verification is performed by comparing the current privilege level and the *PL* field located in the page table. The *PL* field is equal to the most restrictive privilege level of the *PL* fields read in the directory and in the page table.

The MMU is equipped with a 32-entry associative TB (Figure 9.14) and a register used for memorizing the address where the exception occurred. Replacement is performed through an LRU-type algorithm.

9.4.3. *Steps in the access to information*

We now examine three cases of access to an element of information. TR refers to the directory and TP$_i$ to a page table:

– Scenario 1: the TLB provides the physical address during the T_{mmu} clock cycle. This address is instantly available (Figures 9.15 and 9.16). This is the most favorable case in terms of access time.

– Scenario 2: the TLB does not contain the page number. The page is referenced as *valid* in the page table and the page table is referenced as *valid* in the directory (Figures 9.17 and 9.18).

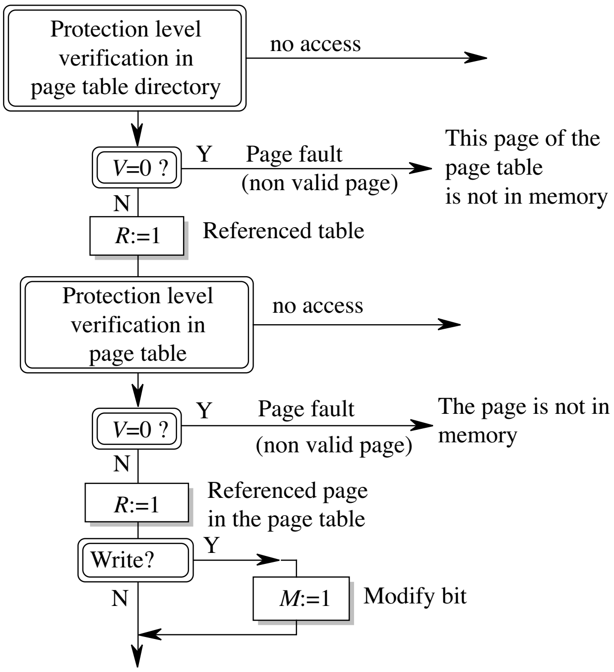

Protection level verification in page table directory
no access
V=0 ?
Y
Page fault (non valid page)
This page of the page table is not in memory
N
R:=1
Referenced table
Protection level verification in page table
no access
V=0 ?
Y
Page fault (non valid page)
The page is not in memory
N
R:=1
Referenced page in the page table
Write?
Y
N
M:=1
Modify bit

Figure 9.13. *Rules for access with two levels of paging*

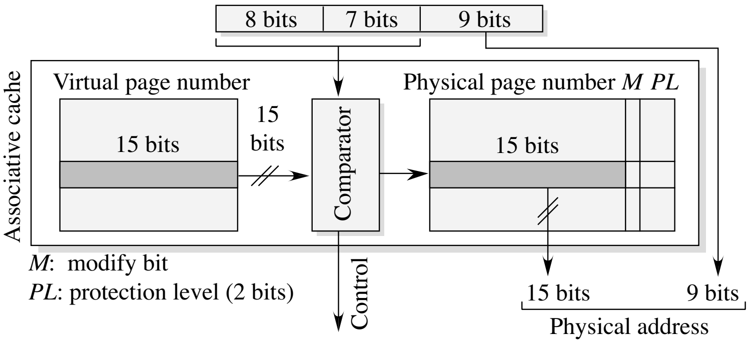


Figure 9.14. *The translation buffer*

Figure 9.17(b) illustrates the access to the directory. The directory provides the page table number n_t, which is stored in the *page frame number* (PFN) register. Access to the page table provides the PPN n_p, which is also stored in the PFN register (Figure 9.18(a)). This register, combined with the offset, allows access to the information in the page (Figure 9.18(b));

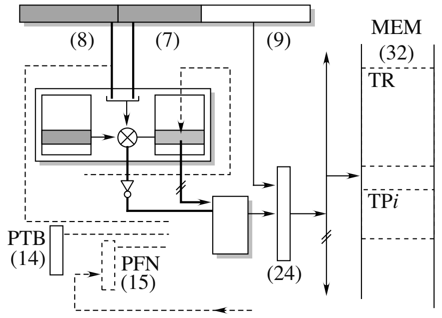


Figure 9.15. *Immediate access to the information: TLB hit (scenario 1)*

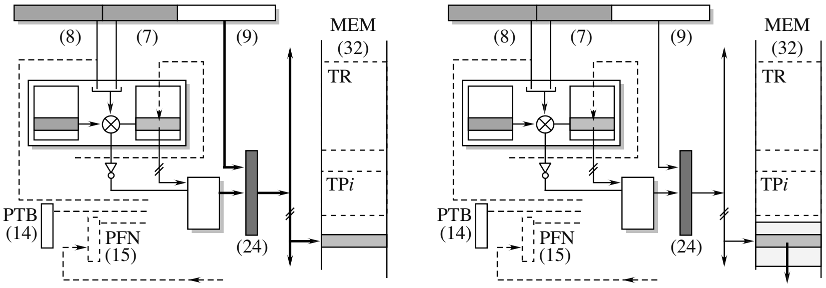


Figure 9.16. *Immediate access to the information: the address is built using the content of the TLB and the offset of the virtual address. The information is then accessed*

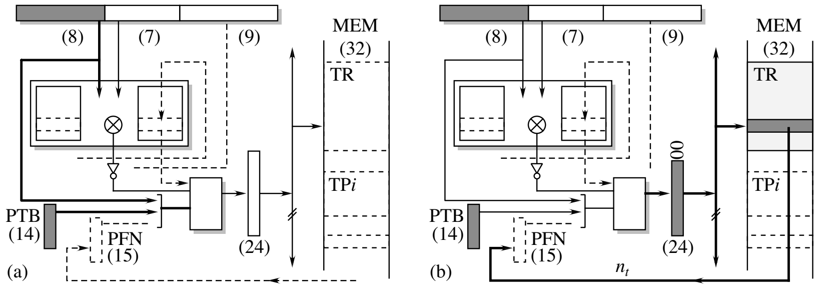


Figure 9.17. *Accessing the directory (scenario 2): miss in the TLB a), access to the page table number in the directory b)*

– Scenario 3: the directory entry, provided by the PDE field (the eight most significant bits), indicates a page table that is invalid because it is absent (Figure 9.19).

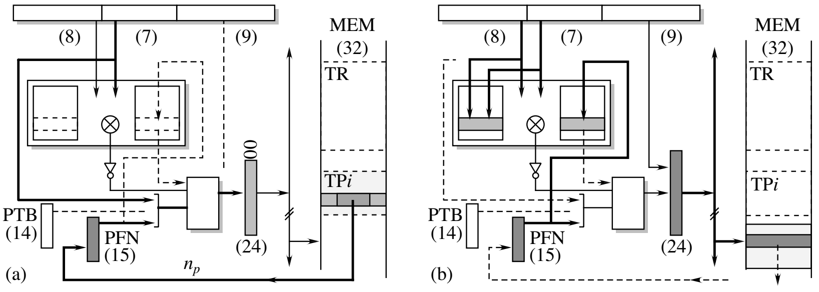


Figure 9.18. *Accessing the information a) and updating the TLB b)*

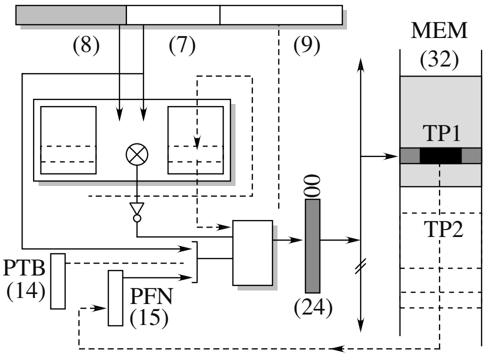


Figure 9.19. *Invalid directory entry: PT1 must be loaded into PFN*

Control is yielded to the operating system. It fetches from the mass storage unit the corresponding page table. The directory entry given by the PDE is updated. Access is performed once again. The PFN is fetched from the directory (Figure 9.20(a)). With the PTE field, we access an entry in the page table (Figure 9.20(b)).

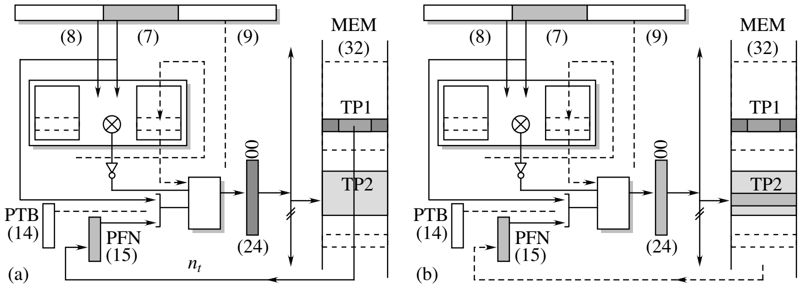


Figure 9.20. *Building the address in the page table: access in the directory a), access in the page table b)*

This, however, produces an invalid address. As it did before, the operating system fetches the required page from the mass storage unit and updates the PTE entry in the page table (Figure 9.21).

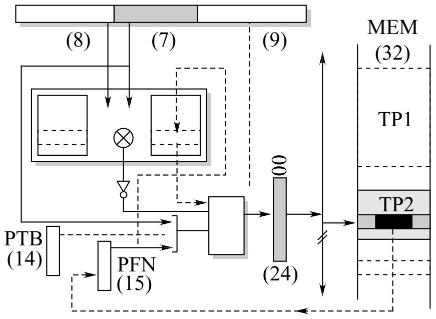


Figure 9.21. *Reading the page table entry*

The information access scenario is the same as in the previous scenarios. After the handling process, the address is constructed using the content of the PTE entry in the page table (transferred into PFN, Figure 9.18(a)) and the nine offset bits. The TLB is updated (Figure 9.18(b)).

9.5. Paged segmentation

The example described here illustrates how the Intel® 80386 microprocessor operates when it functions in what is known as *protected mode*.

As with the Intel® I8086, where the address was built using the content of a segment of a register and an offset, the addresses here are provided in the form of a *segment selector* (the segment register) and an *offset*. The first phase in building the address therefore relies on segmentation.

The selector is a 16-bit register. Its 13 most significant bits indicate an offset in memory for finding an 8-byte segment. We therefore have 8,192 8-byte descriptors. Bit 2 of the selector indicates whether we are dealing with a table of local or global descriptors, and the two least significant bits indicate the privilege level of the task associated with this segment. Each segment descriptor includes a 32-bit base address, the length of the segment in 20 bits (2^{20} blocks of 4 kB at most), and service bits (access rights and service information). The virtual memory addressable space therefore consists of $2 \times 8{,}192$ times (two tables of 8 kilodescriptors) 4 GB, or 64 TB, whereas the physical space is limited to 4 GB.

The segment length is defined by the 20-bit *limit* field. This field indicates how many *clusters* are occupied by the segment. A so-called *granularity* bit indicates the cluster size: 1 byte or 4 KB depending on the value of this bit. Thus, the maximum segment size can be 1 MB or 4 GB.

The virtual memory management system ensures two-level paging of the segments.

Two caches are added to improve the performance of the system. The first (the *descriptor cache*) receives the 8 bytes of the descriptor associated with the selector. This is actually not a cache but a series of registers, not accessible by programming, which store a copy of the descriptor. The second (the TLB) is a cache with 32 entries. It is an set-associative with an associativity of 4. With a page size of 4 kB, this results in 128 kB of addressable space. According to Intel®, the reference hit rate is on the order of 98% for common multi task systems.

The paging system can be invalidated to work only in segmented mode.

The service bits included in each entry of the directory or the page table (Figure 9.22) are used as follows:

– The *A* (*accessed*) bit is set to one after a read or write access to the page referenced in this entry.

– The *D* (*dirty*) bit is set to one when a write operation is performed in the page indicated in this entry.

– The *U/S* (*user/supervisor*) and *R/W* (*read/write*) bits are used in the page protection mechanism.

– The *P* (*present*) bit indicates that this entry can be used for translating address and therefore be loaded into the TLB.

– Three bits are available to the operating system for implementing a page replacement algorithm.

If an entry is not found in the TLB (miss), the presence bit, denoted by P_r, of the directory indicates if the corresponding page table is present in memory or not. If $P_r = 1$, the corresponding entry in the page table is read and the A_r bit is set to one. If the P_t bit in the page table is equal to one, the processor updates the A_t and D_t and accesses the data. The TLB is updated with the 20 bits of the linear address fetched from the page table.

If P_r or P_t is equal to 0, then an exception 14 (*page fault*) is triggered. The address (linear address) of the instruction responsible for the page fault is stored in the control register CR2. A 16-bit word of *error code* (the R/W, U/S and P bits) is placed in the stack where it is handled by the exception handling program. When the page table is updated, the operating system invalidates the TLB entries. This is achieved by writing in the CR3 register.

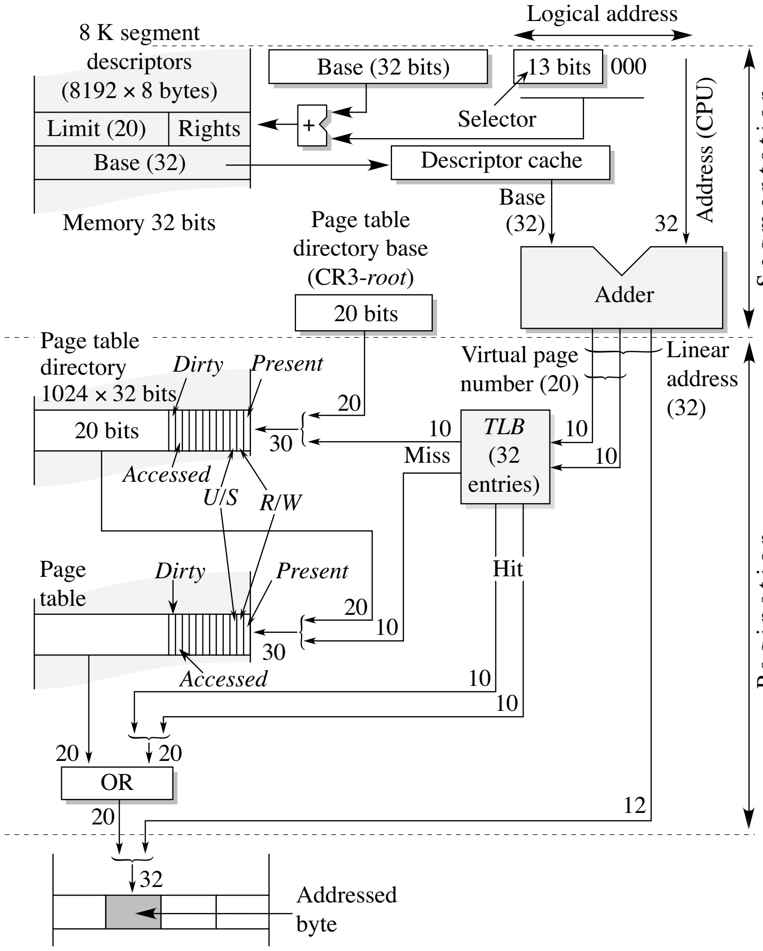


Figure 9.22. *Paged segmentation in the Intel® I80386*

9.5.1. *36-bit extensions*

Four GB of space quickly turned out to be insufficient for machines that had to manage large databases. Intel® introduced the *page size extension* (PSE) with its Pentium® II. The page size is 4 MB (22 offset bits) and only one level of paging is set up by the page table directory, which produces 14-bit page numbers. This mode is only used above 4 GB, while below that limit, 4-kB or 4-MB pages can be used.

Intel® introduced another 36-bit addressing mode, known as the *physical address extension* (PAE), on the Pentium Pro® processors. From a physical perspective, this transformation, which is intended to still allow working with 32 bits, consists of adding a level in the paging mechanism (Figure 9.23). In the paging levels, the directory entries and the page tables take up 8 bytes instead of four.

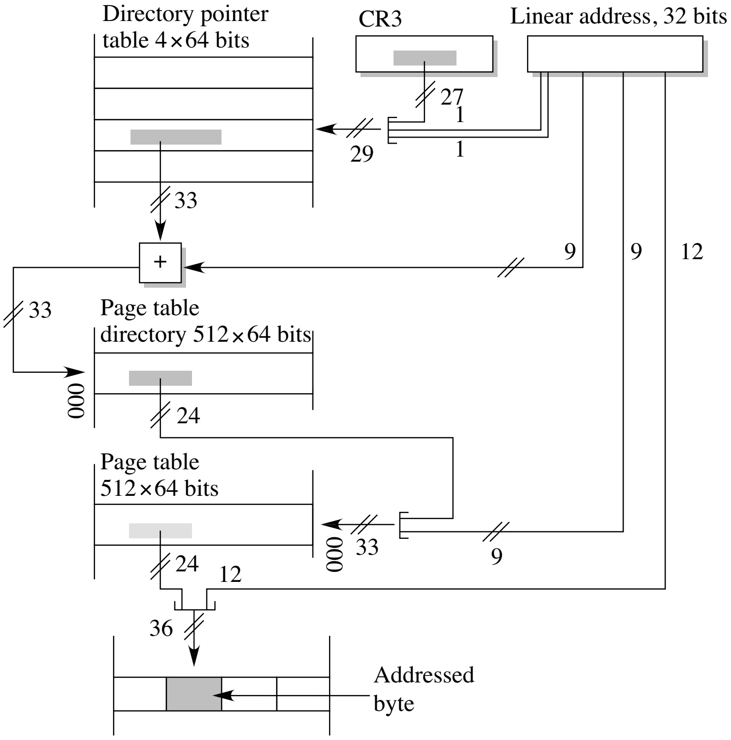


Figure 9.23. *PAE addressing extension for a 4-kB page size*

It is possible to define pages that are of 2 MB in size. In that case, as for the PSE mode, paging is performed on a single level. The page number given in the page directory is then 15 bits in length.

9.6. Exercise

Exercise 9.1 (Virtual memory in the AM 29000) (Hints on page 333)

The AM 29000 [ADV 87] is one of the first standard 32-bit microprocessors. Dating back to the early 1980s, it is equipped with a 25-MHz clock, a pipeline with four levels (*fetch*, *decode*, *execute*, *write-back*), an MMU, a branch predictor, etc. It was widely used in various controllers, in particular video cards used in Apple computers.

Using the documentation excerpts given below, answer the following questions:

1) General questions:

i) Draw an annotated diagram laying out the mechanism for switching from a virtual address to a physical address.

ii) Explain the role of the TLB.

iii) Why do we use the term *trap* rather than interrupt?

2) MMU parameters (we will assume here that the pages are 1 kB in size):

i) In section 9.7.2.1, what is the meaning of the phrase *supervisor mode*? Which is the only mode that allows access to the TLB?

ii) In section 9.7.2.2.1, what is the purpose of the tag *field*? What is the purpose of the *valid bit*?

iii) In section 9.7.2.2.2, what is the maximum number of addressable physical pages?

iv) What category of caches does this TLB belong to?

v) What is the purpose of the U bit?

vi) Still assuming 1-kB pages, give the number of bits in the tag and index fields. What is the purpose of the TID field?

3) Design diagram: draw a diagram of the TLB that clearly outlines the comparators and the fields involved.

9.7. Documentation excerpts

It is an excerpt from a documentation from AMD. It is a detrimental change from sans serif to serif and from two colums to one column.

9.7.1. *Introduction to the MMU*

9.7.1.1. *Memory management*

The AM 29000 incorporates an MMU which accepts a 32-bit virtual byte-address and translates it to a 32-bit physical byte-address in a single cycle. The MMU is not dedicated to any particular address-translation architecture.

l_1: Address translation in the MMU is performed by a 64-entry TLB, an associative table that contains the most recently used address translations for the processor. If the translation for a given address cannot be performed by the TLB, a TLB miss occurs, and causes a trap that allows the required translation to be placed into the TLB.

Processor hardware maintains information for each TLB line indicating which entry was least recently used. When a TLB miss occurs, this information is used to indicate the TLB entry to be replaced. Software is responsible for searching system page tables and modifying the indicated TLB entry as appropriate. This allows the page tables to be defined according to the system environment.

TLB entries are modified directly by processor instructions. A TLB entry consists of 64 bits, and appears as two-word-length TLB registers which may be inspected and modified by instructions.

TLB entries are tagged with a task identifier (TID) field, which allows the operating system to create a unique 32-bit virtual address space for each of 256 processes. In addition, TLB entries provide support for memory protection and user-defined control information.

9.7.1.2. *Memory management unit*

l_2: The MMU performs address translation and memory protection functions for all branches, loads and stores. The MMU operates during the execute stage of the pipeline, so the physical address that it generates is available at the beginning of the write-back stage.

All addresses for external accesses are physical addresses. MMU operation is pipelined with external accesses, so that an address translation can occur while a previous access completes.

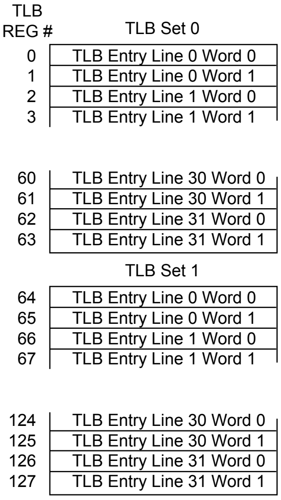
l_3: Address translation is not performed for the addresses associated with instruction prefetching. Instead, these addresses are generated by an instruction prefetch pointer, which is incremented by the processor. Address translation is performed only at the beginning of the prefetch sequence (as the result of a branch instruction), and when the prefetch pointer crosses a potential virtual-page boundary.

9.7.2. *Description of the TLB*

9.7.2.1. *TLB registers*

The AM 29000 contains 128 TLB registers (Figure 9.24).

The TLB registers comprise the TLB entries, and are provided so that programs may inspect and alter TLB entries. This allows the loading, invalidation, saving and restoring of TLB entries.



TLB REG #	TLB Set 0
0	TLB Entry Line 0 Word 0
1	TLB Entry Line 0 Word 1
2	TLB Entry Line 1 Word 0
3	TLB Entry Line 1 Word 1
60	TLB Entry Line 30 Word 0
61	TLB Entry Line 30 Word 1
62	TLB Entry Line 31 Word 0
63	TLB Entry Line 31 Word 1
	TLB Set 1
64	TLB Entry Line 0 Word 0
65	TLB Entry Line 0 Word 1
66	TLB Entry Line 1 Word 0
67	TLB Entry Line 1 Word 1
124	TLB Entry Line 30 Word 0
125	TLB Entry Line 30 Word 1
126	TLB Entry Line 31 Word 0
127	TLB Entry Line 31 Word 1

Figure 9.24. *TLB registers*

TLB registers have fields that are reserved for future processor implementations. When a TLB register is read, a bit in a reserved field is read as a 0. An attempt to write a reserved bit with a 1 has no effect; however, this should be avoided, because of upward compatibility considerations.

The TLB registers are accessed only by explicit data movement by Supervisor-mode programs. Instructions that move data to and from a TLB register specify a general-purpose register containing a TLB register number. The TLB register is given by the contents of bits 6 – 0 of the general-purpose register. TLB register numbers may only be specified indirectly by general-purpose registers.

TLB entries are accessed by registers numbered 0 – 127. Since two words are required to completely specify a TLB entry, two registers are required for each TLB entry. The words corresponding to an entry are paired as two sequentially numbered registers starting on an even-numbered register. The word with the even register number is called Word 0 and the word with the odd register number is called Word 1. The entries for TLB Set 0 are in registers numbered 0 – 63, and the entries for TLB Set 1 are in registers numbered 64 – 127.

9.7.2.2. *Entries*

9.7.2.2.1. Word 0

Bit 31-15: Virtual Tag (VTAG) When the TLB is searched for an address translation, the VTAG field of the TLB entry must match the most significant 17, 16, 15 or 14 bits the address being translated – for page sizes of 1, 2, 4 and 8 kB respectively – for the search to be valid.

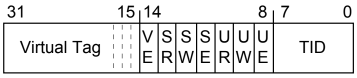


Figure 9.25. *Word 0*

When software loads a TLB entry with the address translation, the most-significant 14 bits of the VTAG are set with the most-significant 14 bits of the virtual address whose translation is being loaded into the TLB.

Bit 14: Valid entry (VE) if this bit is set, the associated TLB entry is valid; if it is 0, the entry is invalid.

Bit 13: Supervisor read (SR) if this bit is set to 1, Supervisor-mode load operations to the virtual page are allowed; if it is set to 0, Supervisor-mode loads are not allowed.

Bit 12: Supervisor write (SW) if this bit is set to 1, Supervisor-mode store operations to the virtual page are allowed; if it is set to 0, Supervisor-mode stores are not allowed.

Bit 11: Supervisor execute (SE) if this bit is set to 1, Supervisor-mode instruction accesses to the virtual page are allowed; if it is 0, Supervisor-mode instruction accesses are not allowed.

Bit 10: User read (SR) if this bit is set to 1, User-mode load operations to the virtual page are allowed; if it is set to 0, User-mode loads are not allowed.

Bit 9: User write (SW) if this bit is set to 1, User-mode store operations to the virtual page are allowed; if it is set to 0, User-mode stores are not allowed.

Bit 8: User execute (UE) if this bit is set to 1, User-mode instruction accesses to the virtual page are allowed; if this bit is set to 0, User-mode instruction accesses are not allowed.

Bit 7-0: Task identifier (TID) when the TLB is searched for an address translation, the TID must match the process identifier (PID) in the MMU Configuration Register for the translation to be valid. This field allows the TLB entry to be associated with a particular process.

9.7.2.2.2. Word 1

Bit 31-10: real page number (RPN) The RPN field gives the most-significant 22, 21, 20 or 19 bits of the physical address of the page – for page sizes of 1, 2, 4 and 8 kB page, respectively – to form the physical address for the access.

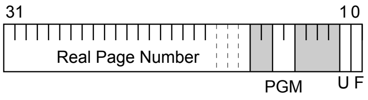


Figure 9.26. *Word 1*

When software loads a TLB entry with an address translation, the most-significant 19 bits of the RPN are set with the most-significant 19 bits of the physical address associated with the translation. The remaining 3 bits of the RPN must be set either to the corresponding bits of the physical address, or to zeros, depending on the page size.

Bit 7-6: User programmable (PGM). These bits are placed on the MPGM0-MPGM1 outputs when the address is transmitted for an access. They have no predefined effect on the access; any effect is defined by logic external to the processor.

Bit 1: Usage (U). This bit indicates which entry in a given TLB line was recently used to perform an address translation. If this bit is 0, then the entry in Set 0 in the line is least recently used; if it is 1, then the entry in Set 1 is least recently used. This bit has an equal value for both entries in a line. Whenever a TLB entry is used to translate an address, the Usage bit of both entries in the line used for translation are set according to the TLB set containing the translation. The bit is set whenever the translation is valid, regardless of the outcome of memory protection checking.

Bit 0: Flag (F). The Flag bit has no effect on address translation, and is affected only by the MTTLB instruction. This bit is provided for software management of TLB entries.

9.7.3. *TLB features*

9.7.3.1. *Memory management*

The AM 29000 incorporates an MMU for performing virtual-to-physical address translation and memory access protection. Address translation can be performed only for instruction/data accesses. No address translation is performed for instruction ROM, input/output, coprocessor or interrupt/trap vector access.

9.7.3.2. *Translation look-aside buffer*

The MMU stores the most recently performed address translation in a special cache, the TLB. All virtual addresses generated by the processor are translated by the TLB. Given a virtual address, the TLB determines the corresponding physical address.

The TLB reflects information in the processor system page tables, except that it specifies the translation for many fewer pages; this restriction allows the TLB to be incorporated on the processor chip where the performance of address translation is maximized.

Each TLB entry is 64 bits long, and contains mapping and protection information for a single virtual page. TLB entries may be inspected and modified by processor instructions executed in the Supervisor mode.

The TLB stores information about the ownership of the TLB entries with an 8-bit Task Identifier (TID) field in each entry. This makes it possible for the TLB to be shared by several independent processes without the need for invalidation of the entire TLB as processes are activated. It also increases system performance by permitting processes to warm-start (i.e. to start execution on the processor with a certain number of TLB entries remaining in the TLB from a previous execution).

Each TLB entry contains two bits to assist management of the TLB entries. These are the Usage and Flag bits. The Usage bit indicates which set of the entry within a given line was least recently used to perform an address translation. Usage bits for two entries in the same line are equivalent. The Flag bit has no effect on address translation, and is not affected by the processor except by explicit writes to the TLB. This bit is provided only for use by the software; for example, it may indicate that a TLB entry may not be replaced.

9.7.3.3. *Address translation*

For the purpose of address translation, the virtual instruction/data space for a process is partitioned into regions of fixed size, called pages, which are mapped by the address-translation process into equivalently sized regions of physical memory, called page frames. All accesses to instructions or data contained within a given page use the same virtual-to-physical address translation.

Pages may be of size 1, 2, 4 or 8 kB, as specified by the MMU Configuration Register. Virtual addresses are partitioned into three fields as shown in Figure 9.27.

9.7.3.3.1. Address translation controls

The processor attempts to perform address translation for the following external accesses:

1) Instruction accesses, if the Physical Addressing/Instructions (PI) and ROM Enable (RE) bits of the Current Processor Status is 0.

2) User-mode accesses to instruction/data memory if the Physical Addressing/Data (PD) bit of the current Processor Status is 0.

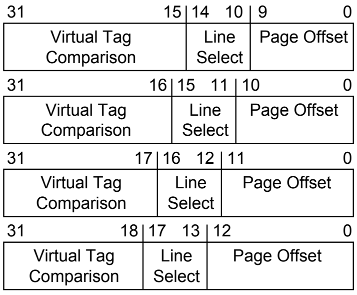


Figure 9.27. *Virtual address.*

3) Supervisor-mode accesses to instruction/data memory if the Physical Address (PA) bit of the load or store instruction performing the access is 0, and the PD bit of the current Processor Status is 0.

Address translation is also controlled by the MMU Configuration Register. This register specifies the virtual page size, and contains an 8-bit PID field. The PID field specifies the process number associated with the currently running program. This value is compared with TID fields of the TLB entries during the address translation. The TID field of a TLB entry must match the PID field for the translation to be valid.

9.7.3.3.2. Address translation process

Address translation is performed by the following fields in the TLB entry: the VTAG, the TID, the VE bit and the RPN. To perform an address translation, the processor accesses the TLB line whose number is given by certain bits in the virtual address (see Figure 9.27).

The accessed line contains two TLB entries, which in turn contain two VTAG fields. The VTAG fields are both compared to bits in the virtual address. This comparison depends on the page size as follows (note that the VTAG bit numbers are relative to the VTAG field, not the TLB entry):

Page Size	Virtual Add. Bits	VTAG Bits
1 Kbyte	31-15	16-0
2 Kbyte	31-16	16-1
4 Kbyte	31-17	16-2
8 Kbyte	31-18	16-3

Note that certain bits of the VTAG field do not participate in the comparison for page sizes larger than 1 kB. These bits of the VTAG field are required to be zero bits.

PART 4

Parallelism and Performance Enhancement

Chapter 10

Pipeline Architectures

The concept of a machine with a *pipeline architecture* is quite old. The IBM System/360 Model 91 had some of the characteristics of such an architecture as early as 1966. At the time, it was in competition with the CDC 6600 for the title of the world's most powerful machine. The introduction of this architecture changed performance by an order of magnitude. Many of the ideas implemented in the 360/91 were introduced starting in 1954 during the development of the IBM 7030 (*Project Stretch*) [SIM 88]. The intention was for this machine to be 100 times more powerful than its predecessor, the IBM 704, even though the technology would only provide a 10-fold enhancement. The answer to this challenge turned out to be the introduction of parallelized instructions.

10.1. Motivations and ideas

The introduction of parallelism in machines happened at a time when machine design was extremely complex. This led designers to initially focus on the instruction set, rather than on the architecture around it. The advent of reduced instruction set computer (RISC) architectures was a result of this approach.

10.1.1. *RISC machines*

From the 1950 to the 1980, the complexity of the instruction set never stopped increasing, earning these machines the name *complex instruction set computers* (CISC). In 1948, the Mark I had seven instructions. In 1978, the Vax, by Digital Equipment Corp.®, the archetype for the CISC, had more than 300. The Intel® 8080 microprocessor and the Motorola® 6800, which date back to the mid-1970s, were equipped with 100 instructions, while the Motorola® 68020 offered roughly 200. As for the Intel® I80286, it came with 150 instructions, not including the addressing modes and the coprocessor instructions. If we assume that SSE extension instructions and new specialized instructions should be included, the Intel® Core$^{\text{TM}}$ i7 type processor can be said to offer about 300 basic instructions, not including the addressing modes.

As the instruction set grew in size, it also became more complex, as is clear from:

– the increase in the number of addressing modes: 9 for the Vax, 11 for the Motorola® 68020, 12 on Intel® processors, etc.;

– the increase in the number of operands (three or more);

– a wide variety of instructions formats and sizes.

This evolution is easily justified. Consider:

– a significant part of development used to be done in machine language;

– processor designers thought they could simplify the design of compilers by offering machine language that was close to high-level language;

– memories were expensive and small. This made the density of the code a determining factor.

Furthermore, the need for a high level of compatibility from one model to the next forced manufacturers to hold on to "fossil" instructions. As a result, the instruction set quickly became quite portly.

A corollary to large instruction sets is increasing processor complexity, particularly for decoding and executing instructions. This complexity reached a point where designers started questioning the wisdom of the approach. They made the following observations:

– Only a few of the available instructions are used: on the IBM®-370, compilers used about 30% of the instruction set.

– High-level languages have rendered assembly language programming obsolete. Assembly language poses significant issues of readability, maintenance and portability.

– Compilers often generate a code that is better optimized when written by a programmer, even by a programmer with experience.

These observations led several teams to challenge the approach used until then and to design a new generation of processors known as RISC, as opposed to the previous architectures.

Douglas Jensen from Carnegie Mellon [COL 85] proposed the following definition for RISC architectures:

1) The instruction set consists of only a few instructions and addressing modes, unlike complex instruction set computers.

2) The instruction format is fixed.

3) Instructions are executed in a single machine cycle.

4) The sequencer is hardware-implemented.

These conditions facilitate a pipeline operation. All of the processors made today are built with such an architecture, although the instruction sets are no longer "reduced".

10.1.2. *Principle of operation*

Each instruction unfolds in several *phases*, with one phase per clock cycle. A phase is handled by one of the pipeline *stages*. This is similar to how an automobile assembly line operates: there is no waiting for the end of the instruction before the next one begins (Figure 10.1).

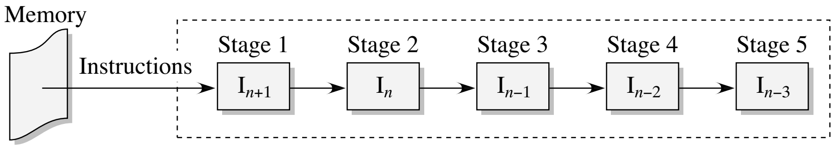


Figure 10.1. *A five-stage pipeline architecture: instruction I_n is in phase 2, while instruction I_{n+1} is in phase 1, etc.*

Loading instructions at each clock cycle requires the ability to simultaneously access the instructions and the data (Figure 10.2).

This requirement imposes the need for a specific memory architecture, the Harvard memory structure, at least for the cache memory.

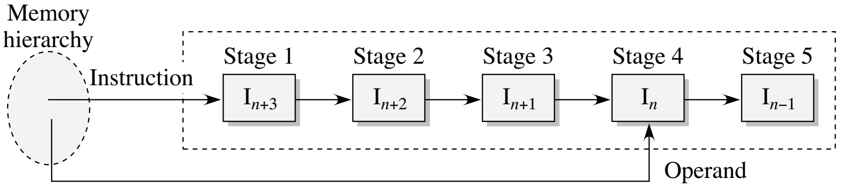


Figure 10.2. *At time n, I_{n+3} is in phase 1 (its own fetch operation), whereas instruction I_n is in phase 4 (fetching an operand)*

EXAMPLE 10.1.– [The Texas Instruments TMS320C30 microprocessor] The TMS320C30 microprocessor by Texas Instruments, which became available in the late 1980s, is a DSP (*digital signal processor*). It is built with a four-stage pipeline architecture with phases denoted by: *F* (*fetch*), *D* (*decode*), *R* (*read*) and *E* (*execute*).

The four phases (Figure 10.3) perform the following functions:

– F: instruction fetch and program counter update;

– D: decoding and computation of the operand addresses;

– R: reading of the operands into memory (if applicable);

– E: execution, two possibilities:

- execution of the operation and writing of the result into a register,
- writing of the result of an execution into memory.

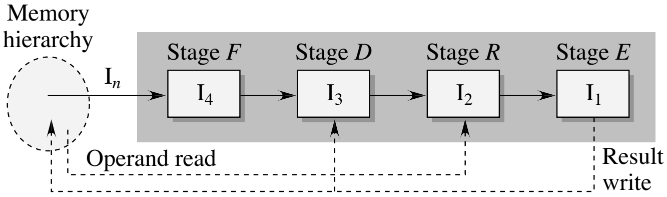


Figure 10.3. *The four-stage pipeline architecture of the Texas Instrument's TMS320C30: the registers are "in" stage D*

Consider the sequence of four instructions I_1, I_2, I_3 and and I_4, the execution of which is represented in Table 10.1.

	$\leftarrow$ Major cycle C_M (τ_m) $\rightarrow$					
	C_m					
	T	$2T$	$3T$	$4T$	$5T$	
Instants	$n-3$	$n-2$	$n-1$	n	$n+1$	
I_1	*F*	*D*	*R*	*E*		
I_2		*F*	*D*	*R*	*E*	
I_3			*F*	*D*	*R*	...
I_4				*F*	*D*	...

Table 10.1. *Evolution of the pipeline state: at time n, I_1, I_2, I_3 and I_4 occupy all four stages*

The major cycle C_M corresponds to the time it required for executing an instruction while the minor cycle C_m is the base clock cycle. When I_1 produces a result, the three instructions that follow are already being executed in the pipeline: I_2 is in phase R, I_3 in phase D, etc.

Phase $3T$ is a good illustration of the previous comment. It contains a memory access for *F* from I_3 and possibly another one for *R* from I_1. The comment is valid for all of the phases.

The pace at which the results are displayed is set by minor cycles, even though the execution time of an instruction is always the duration of the major cycle. This leads us to the following, important property:

> A machine with an N-stage pipeline architecture has a throughput N times greater than the equivalent "non-pipelined" machine.

As we will see, there are many obstacles to overcome so that we can achieve this throughput.

10.1.3. *Cost of the pipeline architecture*

The increase in processor throughput has a cost:

– The control unit is more complex than in a non-pipelined processor with similar specifications.

– To ensure parallel computing, *buffer* elements need to be inserted between the stages to memorize the intermediate results and transfer useful information to the next stages.

– This duplication increases the duration of the major cycle duration, and therefore the *latency* between the reading of the instruction and its execution.

10.2. Pipeline management problems

In section 5.2, we discussed the traditional and very simple architecture of Intel® I8080. In this machine, as in many other machines at the time, the instructions varied in length. This type of instruction set made it impossible to execute one instruction per cycle.

Although it is simpler, the pipeline structure still causes a certain number of issues in managing the flow instructions. These are called *hazards* or *conflicts*, and fall into three categories:

– *Structural hazards* caused by conflicts in accessing a *hardware* resource (memory, register, ALU, etc.): we will discuss below the example of the DLX architecture in which two stages may happen to modify the content of the registers in the same minor cycle.

– *Data hazards*, or *dependency conflicts*: because we are using a pipeline architecture, several instructions going through the pipeline can manipulate the same information. It must be ensured that they do so in the order indicated by the program.

– *Control hazards*, caused by the presence of branch instructions: the instructions that follow have already been loaded into the pipeline when they need to be executed. However, if we are dealing with a "simple" branch, the following instruction should definitely not be executed.

10.2.1. *Structural hazards*

There are two situations that lead to *structure conflicts*:

– Accessing a read – write register: the solution lies in the ability to write and read in a register during the same cycle: the first half-cycle is used for writing, the second for reading.

– Double memory access: when the processors fetch an operand from memory, and if this memory happens to be of the type that can be both program and data, conflict is unavoidable, since there is also access to an instruction at every cycle. This warrants the presence of at least one instruction cache.

EXAMPLE 10.2.– [Structural hazards in registers] Consider a four-stage pipeline machine for which instructions are coded in 32 bits and operations performed in 32 bits. We have the instruction I_n:

```
add r1,r3,#10 ; r1 := r3 + 10
```

which is handled in four phases, as shown below:

– The instruction I_n is copied into the instruction register, and the program counter is incremented. We have the *number* (3) of the register involved in the calculation (`r3`), as well as the immediate value (`10`) of the instruction in the code.

– In the second phase, `r3` and the value 10 (“extended” to 32 bits) are copied into buffer registers. We will assume that there are no *data hazards*, in other words that the value of `r3` is the same as what would have been provided by the sequence of instructions executed on a non-pipelined machine. The content of the instruction register (or of part of the instruction) is saved in a buffer, to keep a record of the number of the target register `r1`. Simultaneously, the instruction I_{n+1} is copied into the instruction register.

– In phase 3, the result of the calculation, which is performed in 32 bits, is stored in buffer T3. The number (1) of the target register `r1` is propagated, so it can be used in the following phase. The instruction I_{n+2} is copied into the instruction register and I_{n+1} advances to the second step.

– Finally, the result is written in register `r1`, the number of which has been propagated. Note that if I_{n+2} requires access to `r1`, there is a conflict with the write operation on the same register (see Figure 10.4).

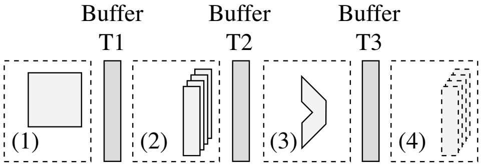


Figure 10.4. *The pipeline has four stages: the registers are represented twice, in stages (2) and (4), even though they are really only present in stage (2): this expresses the possibility of writing and reading in a register during the same cycle*

From a temporal perspective, the instruction unfolds as illustrated in Figure 10.5.

Buffers can be understood as two registers side-by-side. One appears at the end of a stage and the other at the beginning of the following stage. In stage k, the “downstream register” V_k is loaded with the values fetched or calculated in the corresponding phase. These pieces of information have not yet shown up as the buffer outputs. In the next phase, what had been loaded into V_k is transmitted to the outputs of these same buffers, the “upstream buffers” M_{k+1}, which serve as the inputs for stage $k+1$.

The final phase triggers a write operation in the register file. The register file is shown twice, in stages 2 and 4, for reasons of coherence and phase representation.

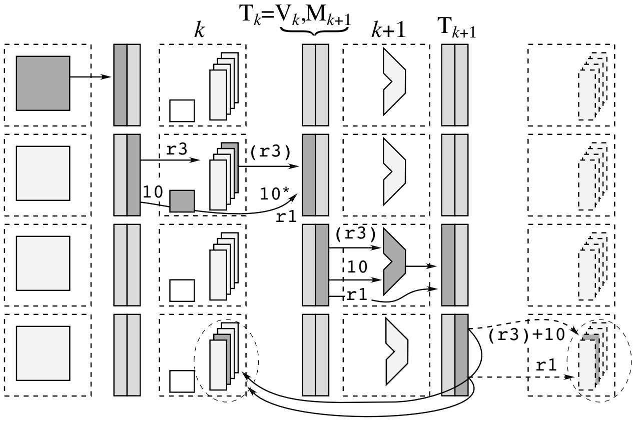


Figure 10.5. *The role of buffers in execution: 10* indicates the extension of this constant to 32 bits in phase 2*

Figure 10.6 gives some idea of how the instruction is carried out.

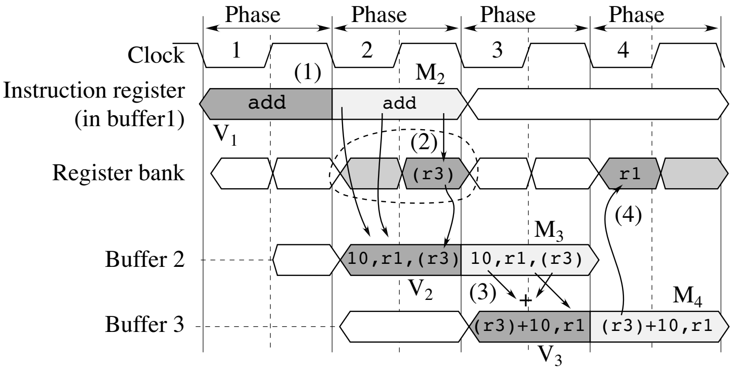


Figure 10.6. *Timing diagram: the dashed line indicates a structural hazard solved by separating phase 2 into half-cycles: writing of the result in the first, and transfer to a buffer in the second. This prevents the waste of a cycle in carrying out the instruction*

The numbers in the figure have the following meaning:

1) The instruction is loaded into the instruction register of buffer T1 (V_1).

2) The instruction is still in the same buffer T1 (M_2). The only numbers available are the numbers of registers `r1` and `r3` and the immediate value `10`. The content of `r3` (`(r3)`), the number “1” and the value `10` (“extended” to 32 bits) are copied into buffer T2 (V_2).

3) The result of the sum (r3)+10 is copied into buffer T3 (V_3), along with the number "1".

4) The writing of the result occurs in phase 4: the result and the register number are in T3 (M_4).

In the following examples, we will not describe buffer operations in such detail. We will instead simply represent their loading.

EXAMPLE 10.3.– [Structural hazard in memory] Consider the following portion of a program. We assume that data and instructions are fetched from *a single* memory. The processor (DLX), we chose as an example, consists of five stages: *fetch* (F), *decode* (D), *execute* (E), *memory* (M), *write-back* (W).

```
lw  r1,0(r2)  ; r1:=(r2)
and r3,r3,r5  ; r3:=r3+r5
sub r7,r6,r4  ; r7:=r6-r4
or  r8,r1,r5  ; r8:=r1 or r5
```

The sequencing is represented in Figure 10.7.

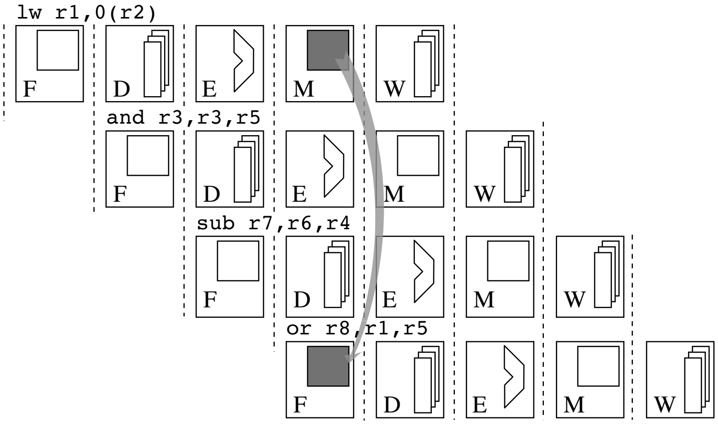


Figure 10.7. *Structural conflict in a memory access*

Phase F of the instruction or r8,r1,r5 is in conflict with the read access to lw (Figure 10.7). Obviously, this type of conflict can be solved by simply adding an instruction cache separate from the data cache.

10.2.2. *Dependency conflicts*

A *dependency conflict* occurs when several instructions being handled in the pipeline manipulate the same registers. The results obtained must be identical to those produced by a non-pipelined machine!

Dependencies are usually divided into three types, RAW, WAR and WAW, with the following behavioral rules:

– RAW (*Read After Write*): an element of information should only be fetched after it has been modified by a preceding instruction.

– WAR (*Write After Read*): a register should only be modified after it has been read by a preceding instruction.

– WAW (*Write After Write*): updates must be performed in the right order. This is typically the case for superscalar architectures. Indeed, if two instructions use two different pipelines and modify the same register, there is a risk of an error if the second write operation is performed before the first one.

EXAMPLE 10.4.– [RAW conflict] Consider the following program, in which the register r1 initialized in the first instruction is used in the following three instructions. We now assume that there are two distinct caches, one for "instructions" and one for "data".

As Figure 10.8 shows, the two instructions and and sub do not have the value of r1 in the decoding step. The or instruction obtains the correct value nonetheless because of the possibility to read and write during the same clock cycle.

```
add r1,r2,#2  ; r1:=r2+2
and r3,r3,r1  ; r3:=r3+r1
sub r7,r1,r4  ; r7:=r1-r4
or  r8,r1,r5  ; r8:=r1 or r5
```

The sequence of instructions is illustrated in Figure 10.8.

WAR-type hazards cannot occur with this architecture. The writing of an information by I_{n+1} is performed in phase W, well after it has been fetched by I_n in phase D. The same is true for WAW conflicts. We will later discuss the case of superscalar architectures, which present this type of conflict.

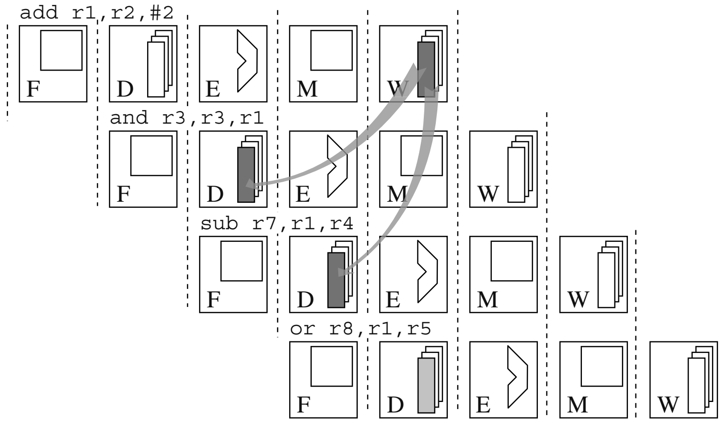


Figure 10.8. *RAW dependency conflicts*

10.2.3. *Branches*

Branches are divided into two categories:

– *unconditional* branches, which always modify the program counter, for example in programs that call procedures or functions;

– *conditional* branches, for which the choice of the following instruction depends on the result of the previous instructions, for example:

- `if ... then ... else` structures for which branches always go "forward";

- loops (`for`, `do`, `while`), which branch either "backward" or "forward" at the last iteration.

The problem inherent to dealing with branches can be expressed as follows: how do we handle a redirect to the target address when the instructions that follow the branch are already in the pipeline? We can prevent the incorrect loading of these instructions by waiting for the computation of the target address to be completed. This is known as "introducing a branch penalty".

10.2.3.1. *Introduction of penalties*

The following example uses the same five-stage architecture as above.

EXAMPLE 10.5.– [Example of an unconditional branch] Consider the instruction `j target`. The address calculation is performed by adding the offset specified by `target` and the address of the instruction that follows the branch. The target address will therefore only be known in the execution phase. We assume here that decoding the branch suspends the execution of the two following instructions. This is known as a *single branch*. We assume that the program counter is updated in phase M of the branch instruction.

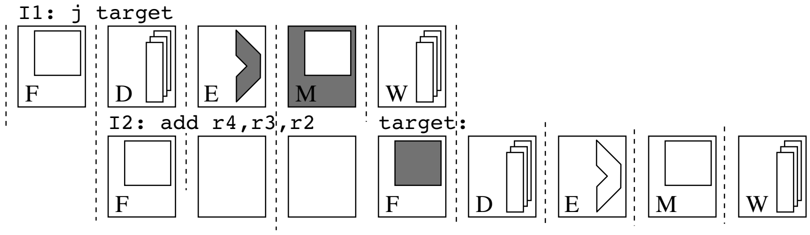


Figure 10.9. *Single branch: branch I1 is followed by I2, I3, etc.*

The instruction is fetched from memory in phase W (diagram in Figure 10.9).

Three cycles are lost during the execution. We say that the penalty is equal to three; given the high number of branches in programs, this is too high a cost.

10.3. Handling branches

The introduction of penalties is an obstacle to achieving maximum pipeline performances. Many solutions involve the hardware and the software to solve this problem. The simplest solution from an architecture standpoint is the use of *delayed branches*, which we discuss in the following section.

10.3.1. *Delayed branches and software handling*

Many RISC machines from the mid-1980s (for example SPARC processors, short for *Scalable Processor ARChitecture*) adopted the solution of *delayed branches*: one or several of the instructions that follow the program are systematically executed. If I_b is a delayed branch instruction with a delay of k, then the k instructions that follow I_b are always executed. A simplistic use of this type of instruction consists of writing k nop instructions after the branch. In the following examples, $k = 1$. The optimal use of this type of branch is left to the compiler, or to the programmer. The solution is to shift k program instructions behind the branch, without modifying the logic.

EXAMPLE 10.6. (Downstream shift – if)– In the case of a if ... then .. else ... structure, the instruction that precedes the branch is shifted so that it instead immediately follows it.

```
          Before                              |           After
----------------------------------------------+------------------------------
          Ij                                  |           ...
          ...                                 |           ...
          beqzd r2,target   ; if              |           beqzd r2,target
          nop                             ->  |           Ij
          ...               ; else            |           ...
          ...                                 |           ...
target:   ...               ; then            |  target:  ...
```

EXAMPLE 10.7.– [Downstream shift – `do ... while`] In the case of a `do ... while` loop, an instruction from the loop replaces the `nop`.

```
        Before                 |       After
-------------------------------+------------------------
loop:   I1                     | loop:  I1
        ...                    |        ...
        add r1,r1,#4           |        ...
        ...                ->  |        In
        In                     |        beqzd r2, loop
        beqzd r2, loop         |        add r1,r1,#4
        nop                    |        ...
```

EXAMPLE 10.8.– [Upstream shift] If the downstream shift is not feasible, one of the instructions that follow can be shifted instead. Thus, in the case of `if ... then ... else ...`

```
          Before                              |           After
----------------------------------------------+--------------------------
          add ...                             |           add ...
          beqzd r1,target   ; if              |           beqzd r1,target
          nop                                 |           In
          ...               ; else            |           ...
          bra endIF                       ->  |           bra endIF
          nop                                 |           nop
target:   ...               ; then            |  target:  ...
          ...                                 |           ...
endIF:    ...                                 |  endIF:   ...
          ...                                 |           ...
          In                                  |           ...
```

EXAMPLE 10.9.– [Shift using a tagged instruction] It is also possible to duplicate the target instruction (we could simply shift it, but nothing tells us it is not the target of another instruction):

	Before			After
target:	sub r5,r6,r17		target:	sub r5,r6,r17
	I1		target2:	I1
	...			...
	add r1,r2,r3			add r1,r2,r3
	beqzd r1,target	→		beqzd r1,target2
	nop			sub r5,r6,r17
	...			...

The additional execution of sub must not trigger a malfunction when the loop comes to an end. When the branch is executed, there is an improvement. The order of instructions is not changed here, which is not the case in example 10.7.

EXAMPLE 10.10.– [Using a delayed branch] The following is a program written in C and the (simplified) assembly code generated by the compiler on a machine equipped with a SPARC processor, a RISC processor with a pipeline that handles branches according to the delayed branch principle, with a delay equal to one:

```
#include <stdlib.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
  static int i, k;
  i = atoi (argv[1]);
  if ( i == 0 ) k = 3;
     else k = 4;
  printf ("k = %d\n", k);
  return 0;
}
```

The parameters are provided to the procedures in registers Reg0, Reg1, etc. The simplified assembly code is as follows:

```
        .common  i, 4
        .common  k, 4
Format:
.asciz "k = %d\n"
main:
call    atoi
load    i, Reg0      /* load the content of i into Reg0 */
cmp     Reg0,0       /* compare Reg0 to 0 */
bne     Label5       /* if Reg0 <> 0 goto Label5 */
store   Reg0, i      /* store Reg0 into i */
```

```
b       Label7        /* goto Label7 */
mov     3, Reg 1      /* set Reg1 to 3 */
Label5:
mov     4, Reg1       /* set Reg1 to 4*/
Label7:
store   Reg1, k       /* store Reg1 into k */
call    printf
load    Format, Reg0 /* store Format address into Reg0 */
ret
```

We can make the following observations:

– Regarding `call`: register `Reg0` is loaded with an instruction located after the function call in the code, but executed at the same as that call in the pipeline.

– Regarding `bne`: if `Reg0 = 0`, the instructions that follow `bne` are executed: `store`, `b` and `move 3,Reg1`. `Reg1` is therefore moved in 3 before being stored in `k`. If `Reg0 <> 0`, we go to `Label5`, execute the instruction that follows `bne`, which has no bearing on the result, and `Reg1` is set to 4.

10.3.2. *Branch predictions*

10.3.2.1. *Static predictions and the annul bit*

The first prediction techniques were described as *static*. Some processors offered instructions for finely tuning the branching behavior of machines. The use of the *annul bit* is an implementation of this type of prediction. It consists of indicating which structures must be executed depending on the branch result.

There are two different rules for improving how programs behave depending on whether the branch goes forward or backward. A bit known as the *annul bit* is used for choosing the branch operating mode. The following examples are again based on the five-stage architecture from example 10.5, an architecture equipped with a system (section 11.3.2) that limits the penalty to one.

– First mode: if we are moving toward the target instruction, the instruction $\mathtt{I}_{n+1}$ that follows the branch is cancelled, and we move on to the target instruction. If the branch is skipped, $\mathtt{I}_{n+1}$ is executed. In the following example, `beqzd` is replaced with `beqz,a`, which operates according to the specified mode.

```
               Before          |             After
-----------------------------------------------------------
loop1:    beqzd target   →    | loop1:    beqz,a target
          add...               |           add...
          ...                  |           ...
          bra loop1            |           bra loop1
target:   ...                  | target:   ...
```

Before the modification, the add instruction that follows the branch is performed an extra time upon exiting the loop.

– Second mode: if the branch is skipped, the instruction $\mathtt{I}_{n+1}$ that follows the branch is canceled and we move on to $\mathtt{I}_{n+2}$. If we are headed toward the target instruction, $\mathtt{I}_{n+1}$ is executed. Consider again example 10.9 with beqzd replaced by beqz,a, which operates like we just described:

```
                Before                 |              After
---------------------------------------+------------------------------
 target:  sub r5,r6,r17                | target:   sub r5,r6,r17
          mul ...                      | target2:  mul ...
          ...                          |           ...
          add r1,r2,r3                 |           add r1,r2,r3
          beqzd r1,target     →        |           beqz,a r1,target2
          nop                          |           sub r5,r6,r17
          ...                          |           ...
```

The sub instruction is executed the same number of times as before the program was modified, because it is not executed upon exiting the loop. The SPARC architecture used this type of static prediction.

EXAMPLE 10.11.– [The PowerPC$^{\mathrm{TM}}$ PPC601] The PowerPC$^{\mathrm{TM}}$ PPC601 offers the ability to choose the operation mode for conditional branches. The instruction code contains a 5-bit field in which the last bit (the “y” bit) indicates the mode that is being operated as explained above [IBM 93].

The optimal use of such branches by a compiler may require the prior execution of the program to perform statistics on the branching results. This is not always possible. It then becomes necessary to use any prior knowledge available about the operation of the program. For example, a branch that moves to a target address with a value less than the current address most often corresponds to the end of a loop. It is therefore logical to predict that the branch will be taken, and in fact, *benchmark* test shows that 56% of branches are “taken” versus 44% which are not.

10.3.2.2. *Runtime prediction*

The ideal way to optimize a program’s performance is to know ahead of time the sequence of instructions, and in particular, the results of the branches. One approach consists of *predicting* the program’s behavior. This prediction can be done *before* the execution. It is referred to as a static prediction, as we saw in the previous section. It can also be undertaken during the execution program, an approach known as *runtime branch prediction*.

– *One-bit prediction*: a first solution consists of assuming that a branch will behave as it did the last time it occurred. A bit called the *prediction bit* indicates the result of this previous execution.

To implement this prediction, the branch instructions are stored in a direct access cache called the *branch history table* (BHT). Each line is associated with a prediction bit (Figure 10.10).

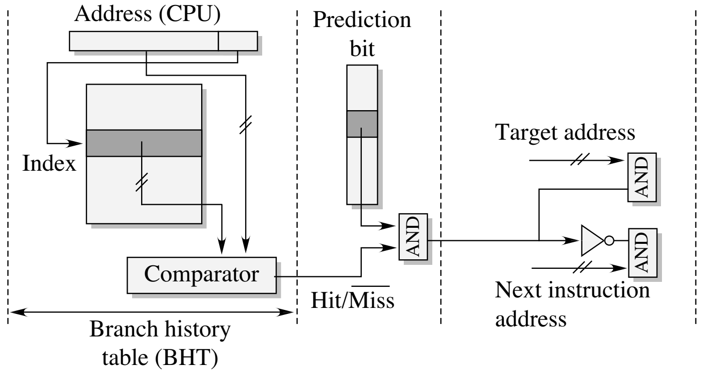


Figure 10.10. *One-bit predictor (branch history table)*

In the instruction reading phase, the processor accesses the BHT. In case of a hit, the prediction bit indicates the address of the following instruction. The validity of this prediction will be confirmed or denied during branch handling.

EXAMPLE 10.12.– Consider the following program with the initial value `r1` $= 5$ and no dependency conflict between `subi` and `bnez`.

```
loop1:  ...
        subi  r1,r1,#1
        ...
        bnez  r1,loop1
```

In the first iteration, there can be no prediction. The following three branches are correctly predicted and the last one incorrectly predicted: we therefore have a 75% successful prediction rate. Performance depends on the penalty cost that stems from incorrect predictions.

– *Two-bit branch prediction*: to improve prediction, we can use predictors equipped with a higher number of bits, for example two. J.E. Smith was the first to propose this runtime prediction model [SMI 81]. The predictor operates as shown in Figure 10.11. We will refer to this type of graph as a *prediction graph*.

- $\bar{b}$ denotes a branch "not taken" (move to the next instruction), b a branch "taken" (move to the target instruction);

- A_1 and A_2 indicate a prediction of $\bar{b}$ with one and two correct predictions, respectively, and B_1 and B_2 indicate a prediction of b.

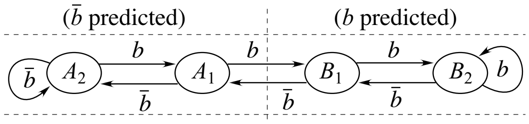


Figure 10.11. *Example of a 2-bit predictor*

In this case, two erroneous predictions in a row are needed for a decision change. This mechanism can be interpreted as a "2-bit" counter. When the counter is equal to 0 or 1, we predict that the branch will not be taken and, if it is equal to 2 or 3, we predict the opposite. Another solution is to introduce "hysteresis", as shown in Figure 10.12.

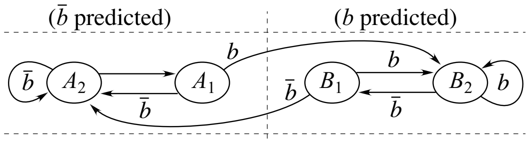


Figure 10.12. *Example of a 2-bit predictor "with hysteresis"*

States A_2 and B_2 are described as "strongly not taken" and "strongly taken", and states A_1 and B_1 as "weakly not taken" and "weakly taken". Note that these two algorithms show hysteresis, since the states predicted as "not taken" and "taken" do not switch in the same way. As with 1-bit prediction, a BHT memorizes n branch instruction addresses. For each entry, a state is updated after the branch is executed according to the prediction (the previous state) and the decision made (the transition).

EXAMPLE 10.13.– Consider again the previous example with the loop nested in another loop:

```
extloop: ...
    ...
intloop: ...
    ...
    subi  r1,r1,#1
    bnez  r1,intloop ; Ii
    ...
    bnez r4,extloop  ; Ie
```

At least two iterations of the internal loop bring the state to B_2. The move to the external loop switches the state to B_1. Returning to the internal loop (`Ii` instruction) immediately provides a correct prediction (switch from B_1 to B_2). This mechanism is well suited for nested loops. If we had only one prediction bit, executing the last instruction `Ii` of the internal loop would switch this prediction bit, and returning to this instruction would produce an erroneous prediction.

NOTE.– It is possible to use n prediction bits where $n > 2$. The resulting gains do justify such a choice. The determining factor is the number of cache entries. In practice, 4,096 entries seem to be a good compromise.

10.3.3. *Branch target buffer*

An additional improvement consists of storing along with the branch instruction the target address of its last execution [LEE 84]. This is done using a *branch target buffer* (BTB) (Figure 10.13) with N entries associated with the stage for the instruction fetch.

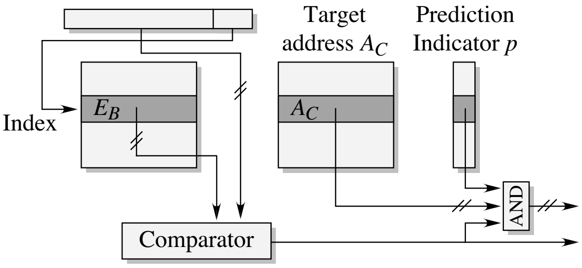


Figure 10.13. *Branch target buffer*

Each entry contains three fields: the address tag E_B of a branch instruction I_B, a target address A_c and in some cases a prediction indicator p (see example 10.20 of a branch target buffer without a predictor). The set of prediction bits is sometimes called the *pattern history table*.

There are several options for managing the BTB, involving the three elements of information E_B, A_c and p:

– Initialization of the E_B field (line loading):

- the branch instructions are systematically loaded into the BTB, whether the branch is taken or not or,

- a branch instruction is only stored in the BTB the first time the branch is taken. If it is never taken, it will never be included in the BTB;

– Initialization of the “target” field:

- only the target address is kept in A_c or,

- the address resulting from the branch (target address or address of the following instruction) is stored in A_c;

– Initialization of the prediction bits: “taken” or “not taken” depending on the line loading option.

10.3.3.1. *Return stack buffer*

Return functions are a special type of branch. They consist of an unconditional branch with an indirect addressing mode. However, the return address depends on the location of the `call` instruction and there is no point in using the BTB.

EXAMPLE 10.14.– [Using the BTB for a `ret` instruction] Consider the following sequence of instructions:

```
       ...
       call func1 ; instruction I1
       ...
       call func1 ; instruction I2
       ...
func1: ...
       ...
       ret
```

After the first call, the tag for `I1` is stored in the BTB along with the target address `func1`. When `ret` is executed, the tag for this instruction is stored in the BTB along with the return address (the address of `I1` “+1”) and the branch is taken. The second return produces a hit in the BTB, but the target address is incorrect. This situation is quite common.

To avoid this mistake, we use a specific stack – the return stack buffer (RSB) (example of the RSB in the Intel® Pentium® P6) – for memorizing the return addresses using a “push/pop” mechanism (8 entries in the P6, 12 in the Pentium® 4, 16 in the *return address stack* of the AMD-K6, etc.). In the case of multithreaded multiprocessors, an RSB is attached to each thread.

10.3.4. *Global prediction*

Until now, branches were dealt with individually, particularly in regard to prediction. We have not used the correlations that may exist between "neighboring" branches. In fact, in a loop, the result of a test, and therefore the branching decision, can depend on a previous test. This can generate a periodic sequence of decisions that could be taken into account by the predictor [YEH 91].

To memorize the sequence of decisions, in other words their history, we use an m-bit shift register (a history register, HR). Generally, m ranges from 2 to 4. If $m = 2$ and RH $= 10$, the last branch was not taken (bit set to 0) and the first-to-last was taken (bit set to 1). This is known as the *global history*, since it involves several distinct branches. Contrary to what we have seen so far, we keep here a record, for each of the branches, of the global history, instead of predicting its behavior in an isolated manner. The content of each HR, and therefore each "historical" configuration, can be used to choose one of 2^m predictors. Thus, each branch is managed by several predictors indexed by the global history (HR) (see Figure 10.14).

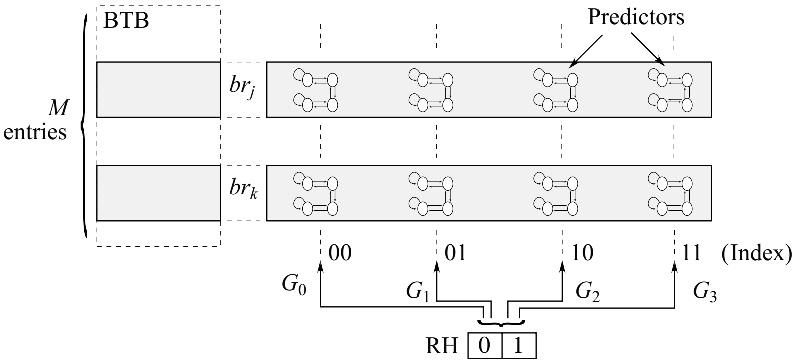


Figure 10.14. *"Two-bit" prediction mechanism with an m = 2-bit history*

The number M of branches that can be handled is independent of the prediction mechanism. It is determined by the number of entries in the BTB.

EXAMPLE 10.15.– [Prediction with history] Consider the program:

```
    a = 0;
do {
    if (a != 0){ .... } % branch br1
    a = (a+1)%3;        % a=a+1 modulo 3
    b = (a+1)%3;        % b=a+1 modulo 3
    if (b != 0){ .... } % branch br2
    ...
    } while (condition)
```

Executing the previous loop yields a periodic sequence of decisions "`nt, t, t, t, nt, t, t, t, t`" (`t` for "taken" and `nt` for "not taken"):

1	2	3	4	5	6	7	8	9	10	11
br1 nt	br2 t	br3 t	br1 t	br2 nt	br3 t	br1 t	br2 t	br3 t	br1 nt	br2 t
12	13	14	15	16	17	18	19	20	21	22
br3 t	br1 t	br2 nt	br3 t	br1 t	br2 t	br3 t	br1 nt	br2 t	br3 t	br1 t
23	24	25	26	27	28	29	30	31	32	...
br2 nt	br3 t	br1 t	br2 t	br3 t	br1 nt	br2 t	br3 t	br1 t	br2 nt	

In this case, the history register is 3 bits in length.

n		RH$_n$	RH$_{n+1}$	t	t/nt	R	G_0	G_1	G_2	G_3	G_4	G_5	G_6	G_7
0	br1	000					00	00	00	00	00	00	00	00
	br2	000					00	00	00	00	00	00	00	00
	br3	000					00	00	00	00	00	00	00	00
1	br1	000	000	nt^1	nt	OK	$\boxed{00}$	00	00	00	00	00	00	00
2	br2	000	100	nt^2	t	–	$\boxed{01}$	00	00	00	00	00	00	00
3	br3	100	110	t^3	t	OK	00	00	00	00	$\boxed{01}$	00	00	00
4	br1	110	111	nt	t	–	00	00	00	00	00	00	$\boxed{01}$	00
5	br2	111	011	nt	nt	OK	01	00	00	00	00	00	00	$\boxed{00}$
6	br3	011	101	nt	t	–	00	00	00	$\boxed{01}$	01	00	00	00
7	br1	101	110	nt	t	–	00	00	00	00	00	$\boxed{01}$	01	00
8	br2	110	111	nt	t	–	01	00	00	00	00	00	$\boxed{01}$	00
9	br3	111	111	nt	t	–	00	00	00	01	01	00	00	$\boxed{01}$
10	br1	111	011	nt	nt	OK	00	00	00	00	00	01	01	$\boxed{00}$
11	br2	011	101	nt	t	–	01	00	00	$\boxed{01}$	00	00	01	00
12	br3	101	110	nt	t	–	00	00	00	01	01	$\boxed{01}$	00	01
13	br1	110	111	nt	t	–	00	00	00	00	00	01	$\boxed{10}$	00
14	br2	111	011	nt	nt	OK	01	00	00	01	00	00	01	$\boxed{00}$
15	br3	011	101	nt	t	–	00	00	00	$\boxed{10}$	01	01	00	01

n		RH$_n$	RH$_{n+1}$	t	t/nt	R	G_0	G_1	G_2	G_3	G_4	G_5	G_6	G_7
16	br1	101	110	nt	t	–	00	00	00	00	00	$\boxed{10}$	10	00
17	br2	110	111	nt	t	–	01	00	00	01	00	00	$\boxed{10}$	00
18	br3	111	111	nt	t	–	00	00	00	10	01	01	00	$\boxed{10}$
19	br1	111	011	nt	nt	OK	00	00	00	00	00	10	10	$\boxed{00}$
20	br2	011	101	nt	t	–	01	00	00	$\boxed{10}$	00	00	10	00
21	br3	101	110	nt	t	–	00	00	00	10	01	$\boxed{10}$	00	10
22	br1	110	111	t	t	OK	00	00	00	00	00	10	$\boxed{11}$	00
23	br2	111	011	nt	nt	OK	01	00	00	10	00	00	10	$\boxed{00}$
24	br3	011	101	t	t	OK	00	00	00	$\boxed{11}$	01	10	00	10
25	br1	101	110	t	t	OK	00	00	00	00	00	$\boxed{11}$	11	00
26	br2	110	111	t	t	OK	01	00	00	10	00	00	$\boxed{11}$	00
27	br3	111	111	t	t	OK	00	00	00	11	01	10	00	$\boxed{11}$
28	br1	111	011	nt	nt	OK	00	00	00	00	00	11	11	$\boxed{00}$
29	br2	011	101	t	t	OK	01	00	00	$\boxed{11}$	00	00	11	00
30	br3	101	110	t	t	OK	00	00	00	11	01	$\boxed{11}$	00	11
31	br1	110	111	t	t	OK	00	00	00	00	00	11	$\boxed{11}$	00
32	br2	111	011	nt	nt	OK	01	00	00	11	00	00	11	$\boxed{00}$
32	...													

NOTE.– References (1), (2) and (3) in the table indicate a "miss" in the BTB. In the first two cases, we are dealing with "forward branches". The prediction is nt (prediction column P). In the third branch (a "backward" branch), the prediction is t. The counter update is performed as indicated by the arrows. Thus, at time $n = 2$, the content RH$_n$ (000) indicates the counter to use (G_0). The branch, which is taken (t in column t/nt), triggers the incrementation of the counter (switch from state A_2 to state A_1 in Figure 10.11). The state RH$_{n+1}$ switches to (100) (shift to the right by inserting a "1" on the left of the branch that was "taken").

After a few initialization cycles, all of the predictions become correct (OK in the result column R). Obviously, if the number K_b of branches in the loop is too large, the system no longer presents an advantage.

10.3.5. *Examples*

EXAMPLE 10.16.– [The TMS320C40] The annul bit technique is implemented in the Texas Instrument's TMS320C40 through the instructions b*cond*af (*annul false*) and b*cond*at (*annul true*).

For b*cond*af, if the *cond* condition is met, the branch is taken and the three instructions that follow are executed. Otherwise, the *execution phases* of these three instructions are inhibited. b*cond*af is well suited for loop exit tests.

For b*cond*at, when the *cond* condition is met, the branch is taken and the *execution phases* of the three instructions that follow are inhibited. Otherwise, we move on to the next instruction. This instruction is well suited for tests at the start of a loop.

In both cases, the three instructions that follow must be free of branches (also, interrupts are inhibited) or run the risk of leaving the pipeline in an unpredictable state:

```
       ldi       *ar1,r0     ;  r0 := (ar1)
       bnegat    lnext       ; branch if negative
       addi      *++ar2,r3   ; (ar2 is incremented)
       nop
       nop
lnext:
```

If the result is negative (the branch is taken), register ar2 is still modified (execution of the decoding phase) and the data are read. On the other hand, r3 is not modified, since the execution phase is not undertaken.

EXAMPLE 10.17.– [The case of the Intel® Pentiums II and III] PII and PIII type architectures are equipped with a BTB with 512 entries [INT 97].

If the address of a branch is located in the BTB, the address of the instruction to execute is directly provided to the *instruction fetch unit*.

If there are no valid entries in the BTB, a static prediction is made based on the direction of the branch (forward or backward). Once the branch is taken, the cache is updated.

– The prediction graph (Figure 10.15) used in these architectures is slightly different from the other graphs we have seen so far.

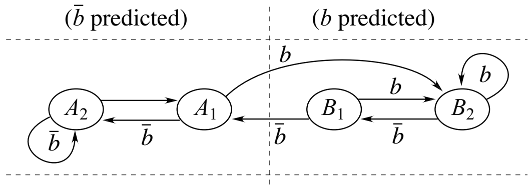


Figure 10.15. *Prediction graph for the PII and PIII*

– In the case of the Intel® Pentium Pro®, the runtime prediction algorithm uses the directions of the four previous branches; in Figure 10.16, the shift register

h_0, h_1, h_2, h_3 contains this information. If HR contains 0101, then the last and ante-penultimate branches were not taken (h_0 and h_2 bits set to 0). Each 4-tuple (h_0, h_1, h_2, h_3) and each branch correspond to a 2-bit prediction graph.

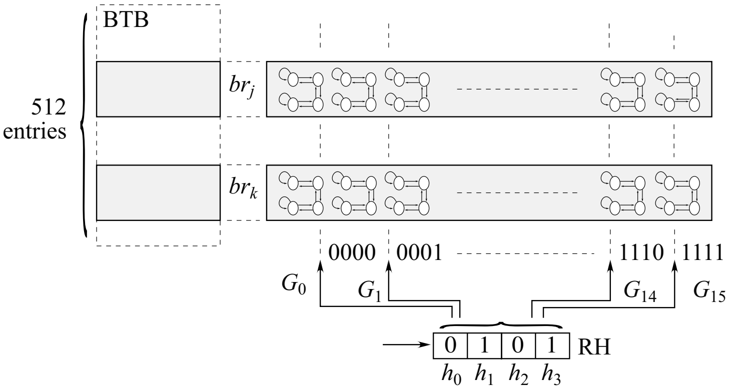


Figure 10.16. *Use of the branch history in an Intel® Pentium Pro®: each configuration is associated with a state (A_1, A_2, B_1, B_2) of the prediction graph*

– Intel® Pentiums also include a cache for predicting return addresses.

– In the Intel® Pentium Pro®, penalties are evaluated on a three-level scale:

i) no penalty, branches “not taken”:

- branches correctly predicted as “not taken”,

- “forward” branches not referenced in the BTB and predicted by default as “not taken”;

ii) minor penalty: branches correctly predicted as “taken” (loss of a fetch cycle);

iii) significant penalty: mispredictions: at least nine cycles are lost (*in-order issue pipeline*), in addition to the time spent “removing” the mispredicted branch instruction. In practice, the penalty is between 10 and 15 cycles (26 at the most).

In the case of static prediction, non-referenced branches are handled as follows:

– unconditional branches are predicted as “taken”;

– conditional backward branches are predicted as “taken”;

– conditional forward branches are predicted as “not taken”.

EXAMPLE 10.18.– Consider the following program:

```
loop1: mov eax,operand
        sub eax,ebx
        add eax,edx
        dec eax
        jne loop1
```

When it is first encountered, `jne` is statically predicted as “taken” (backward address).

EXAMPLE 10.19.– Consider the following program:

```
        mov eax,operand
        and eax,&hffff
        add eax,edx
        shld eax,7
        jc nexti
        mov eax,0
nexti:  call subprog
```

When it is first encountered, `jc` is statically predicted as “not taken” (*fall through*). The `call` instruction is an unconditional branch statically predicted as “taken”.

EXAMPLE 10.20.– [The branch target cache in the AMD 29000] The branch target cache (BTC) of the AMD 29000, is a set-associative cache with two 16-line blocks (Figure 10.17). Its content is indexed using the branch *target* address, with the tag portion of the address compared to the addresses stored in the cache. In case of a *hit*, the branch is predicted as “taken” and the corresponding line of four instructions contains the *target* instruction and the next three instructions.

The processor will therefore fetch its instructions from either the BTC or the IPB (instruction prefetch buffer). The latter receives the instructions four cycles before the beginning of their execution and is used for improving performance when handling branches. When a branch to the address A is first executed, the target instruction and the three instructions that follow are loaded into the BTC. When this branch is executed once again, the processor fetches these four instructions (with addresses A through $A + 3$) from the BTC and at the same time preloads the IPB with the instructions with addresses $A + 4$ through $A + 7$.

All of the branch instructions in the AMD 29000 are delayed branches. The penalty is equal to one. In the following table, the instruction following the branch is denoted by `Id` and the target instruction located in the BTC, and with the actual address A, is denoted by $\mathtt{It}_A$.

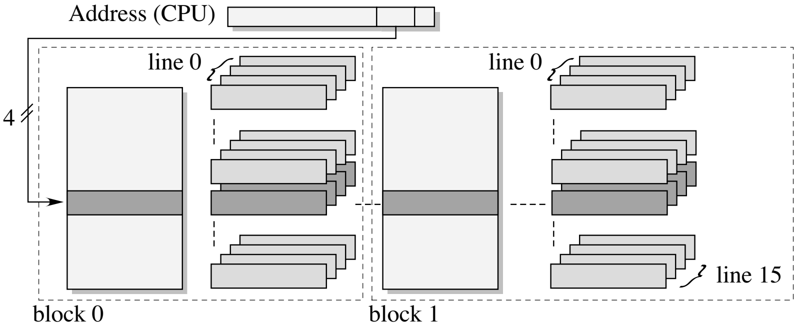


Figure 10.17. *Branch target cache of the AMD 29000 (the two least significant bits must be used before the comparison!)*

`branch`	F	$\boxed{\texttt{D}}^{1}$	$\boxed{\texttt{E}}^{2}$	WB							
`Id`		F	D	E	WB						
$\texttt{It}_A$			F	D	E	WB					
$\texttt{It}_{A+1}$					F	D	E	WB			
$\texttt{It}_{A+2}$						F	D	E	WB		
$\texttt{It}_{A+3}$							F	D	E	WB	
$\texttt{I}_{A+4}$				$\square^{3}$				F	D	E	WB

The calculation of the branch target address is performed during the decoding phase (box (1)). The MMU delivers the corresponding physical address in the execution phase (box (2)). At the same time, the target instruction is fetched $\texttt{It}(A)$. Box (3) indicates the start of the memory access to load the IPB with $\texttt{I}_{A+4}$ and the three instructions that follow.

Each line in the cache includes a valid bit as well as two service bits, one defining the handling context (supervisor/user mode) and the other identifying the space where the instruction is read – ROM or data memory. The replacement management algorithm is “random”, based on the content of the CPU clock.

10.4. Interrupts and exceptions

As we have already seen in Chapter 6 on inputs and outputs, events that modify the execution of a program can be categorized as *faults*, *traps*, and *interrupts*. These events are difficult to handle in a pipeline architecture because their occurrence affects several instructions. Even worse, several events can affect several instructions simultaneously. In the DLX architecture, for example, a page fault and a division by 0 can be triggered in the same cycle.

10.4.1. *Interrupts*

When we come across an interrupt, the instructions in the pipeline are executed. Given the difficulties of dealing with delayed branches of k cycles, interrupts are inhibited until the k instructions that follow the branch are executed.

In most cases, when an interrupt occurs, the instructions in the pipeline are carried out. In certain processors, interrupts cause the instruction in the read cycle to be “canceled” (not carried out). In other cases, the decision to cancel is made during the decoding cycle.

A canceled instruction has to be read again after the interrupt program is executed.

10.4.2. *Traps and faults*

Traps and faults are the most difficult exceptions to deal with. Consider the case of the DLX. “Internal” events that are problematic can be categorized as follows:

Stage	Type of exception
F	Page fault, access denied, alignment problem
D	Illegal instruction code
E	Arithmetic exception: division by 0, overflow, etc.
M	Page fault, access denied, alignment problem

Not only can several exceptions occur in the same cycle, they can also occur in a different order than the execution of instructions. Yet the handler may require that they be taken into account in that order. This is known as a *precise interrupt*.

The mode of operation is as follows: a register (state vector) associated with each instruction receives a code identifying the exception. As soon as an exception code is detected, all write signals are inhibited. In the W phase, the exception register is tested. If there are exceptions, they are handled in this phase. This ensures that the handling is done in the proper order.

Chapter 11

Example of an Architecture

The processor we will use to illustrate the concepts we have introduced is built with an architecture known as DLX (Deluxe). Similar to the architecture of the MIPS processor, it was used by Hennessy and Patterson in their textbook [HEN 96, HEN 11] to present an example of a pipeline architecture. It is a processor with thirty-two 32-bit "integer" registers (the `R0` register always contains 0) and thirty-two 32-bit "floating-point" registers, denoted by `Rk` and `Fk`, respectively. It is equipped with `load` and `store` instructions in the format `lw Rk,address` and `sw address,Rk`. The floating-point unit can handle 32-bit (single precision) or 64-bit (double precision) numbers.

The format of arithmetic instructions is such that `op Rk,Rn,Rm` performs the operation `Rk := Rn op Rm`.

An unconditional branch is denoted by `j address` and the only conditional branches are in the format `beqz Rk,address` and `bnez Rk,address` where the test consists of determining whether the content of the `Rk` register is zero.

11.1. Presentation

11.1.1. *Description of the pipeline*

The DLX has a five-stage pipeline. Figure 11.1 shows its design. The five phases of the DLX are as follows:

– *Instruction Fetch*: fetch the instruction from the program memory M_p;

– *Instruction Decode*: decoding phase and transfer of the registers involved in the instruction to the ALU buffer registers. The sign extension of the immediate operands is performed in this phase;

– *Instruction Execute*: execution of the instruction in the arithmetic logic unit. For arithmetic and logic operations, the ALU receives the operands. For memory access instructions, or for calculated branches, the ALU is used for address calculations. With conditional branches, a unit (denoted by CU, short for compare unit, in Figure 11.1) performs the comparison operation;

– *Memory Reference*: read or write access to the data memory M_d. For memory write operations, the calculated address and the source data are used. For read operations, only the calculated address is used. For branches, this phase corresponds to loading the program counter;

– *Write Back*: the results obtained from executing the arithmetic and logic instructions, and from memory read operations, are written into the registers.

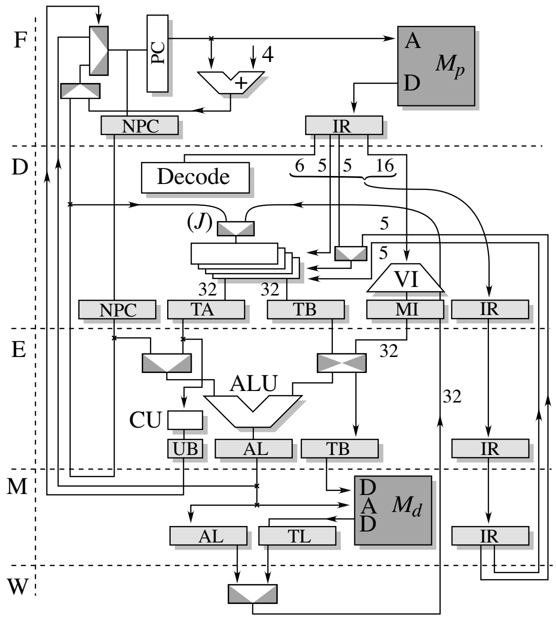


Figure 11.1. *The DLX architecture: each stage is separated from its neighbors by a "layer of registers"*

If there are no particular problems (conflict resolution), the operation is symbolized as follows:

	T	$2T$	$3T$	$4T$	$5T$	$6T$	$7T$
I$_1$	F	D	E	M	W		
I$_2$		F	D	E	M	W	
I$_3$			F	D	E	M	W
...				...			

Table 11.1. *Steps in the pipeline operation*

The internal operation of the pipeline is described in Table 11.2 We use the notation <phase>.<register> to refer to the different registers of Figure 11.1 or, if there are no ambiguities, <register>. Thus, D.TA is simply referred to as TA. The registers that receive the instruction code (or part of the instruction code) are F.IR, D.IR, E.IR and M.IR.

Stage	Instruction type		
	All instructions		

F	F.IR ← MEM(IR)
	if (opcode(E.IR)=branch and UB) then E.AL else PC+4		

D	D.TA ← R(F.IR(6:10))
	D.TB ← R(F.IR(11:15))
	D.NPC ← F.NPC
	D.IR ← F.IR
	D.MI ← F.IR(16:31) with sign extension		

	ALU instruction	Load /Store	Branch

E	E.IR ← D.IR
	E.AL ← (TA op D.TB)
	or (TA op MI)
	UB ← 0
		E.IR ← D.IR
		E.AL ← TA+MI
		UB ← 0
		E.TB ← D.TB
			E.AL ← D.NPC+MI
			UB ← (D.TA op 0)

M	M.IR ← E.IR
	M.AL ← E.AL
		M.IR ← E.IR
		TL ← MEM(E.AL)
		or
		MEM(E.AL) ← E.TB	

W	R(M.IR$_{16:20}$) ← M.AL or R(M.IR$_{11:15}$) ← M.AL	R(M.IR$_{11:15}$) ← TL (only for load)	

Table 11.2. *Operation without forwarding*

REMARK 11.1.– In the decoding phase:

– TA ← R(F.IR(6:10)) means that the TA (or D.TA) register of stage D is loaded with the register whose number is given by bits six through 10 of the instruction register (IR or F.IR);

– D.TB ← R(F.IR(11:15)) means that the TB (or D.TB) register of stage D is loaded with the register whose number is given by bits 11 through 15 of the instruction register (IR or F.IR);

– D.MI ← F.IR(16:31) "with sign extension" is used to recover an element of "immediate data" and to extend it to 32 bits (the sign bit is copied into the 16 most significant bits);

– D.NPC ← F.NPC corresponds to a copy of F.NPC ("New Program Counter") into D.NPC, etc.

In phase M, there are two cases:

– in the case of arithmetic or logic instructions, there is no memory access, only transfers M.IR ← E.IR and M.AL ← E.AL;

– in the case of a load operation, TL ← MEM(E.AL), the content of the memory word with the address E.AL is transferred into the M.TL register. In the case of a write operation, MEM(E.AL) ← E.TB, the content of the E.TB register is transferred into the memory word with the address E.AL. In both cases, the copying operation M.IR ← E.IR is performed.

Not all of the constraints on proper operation are shown in Table 11.2. In particular, we will see in section 10.4.2 that phase W is conditioned on elements other than simple transfers. We have already mentioned this type of problem in example 10.2.

11.1.2. *The instruction set*

The DLX instructions [SAI 96] involve up to three operands. Memory accesses are only performed through `load` and `store` instructions.

There are three types of integer data: byte, half-word (16 bits), and word (32 bits). Access is performed with the most significant bit first (big-endian). The data are always "aligned": words are aligned with $4n$ addresses, half-words with $2n$ addresses.

The DLX addressing modes are as follows:

– immediate addressing involving constants: for example, `add r3,r5,#6` triggers the operation `r3:=r5+6`;

– indirect addressing with offset: `lw r3,6(r1)` loads register `r3` with the content of the memory word with the address `r1+6`;

– implicit addressing: `and r3,r5,r1` places the result from applying the logical AND to the content of registers `r5` and `r1` in register `r3`;

– direct addressing: this last mode is obtained using an offset with register `r0`, for example `lw r2,3264(r0).`

The main instructions are given in the Appendix, section A2.

11.1.3. *Instruction format*

DLX instructions are organized in four groups:

– Type I: Instructions involving immediate operands or offsets (branches). This includes `load` and `store` instructions;

– Type R: Instructions that do not require an immediate value, an immediate address or an offset. This includes all register to register operations, offsets, indirect branches (`jalr` and `jr`);

– Type J: This group includes the two direct branching instructions `j` and `jal`. Their operand is an address;

– Instructions that rely on the arithmetic coprocessor.

The instruction format is as follows:

– Immediate addressing (example: `andi r4,r12,#0hff00` specifies a 16-bit mask, but uses 32 bits (`ffffff00`) because of the sign extension:

0 5	6 10	11 15	16 31
Instruction code	Source register	Target register	Immediate Value

– Implicit addressing (example: `sub r5,r6,r7` executes `r5:=r6-r7`):

0 5	6 10	11 15	16 20	21 25	26 31
Instruction code	Source 1 register	Source 2 register	Target register	Offset	Function code

If we refer to instruction coding in MIPS processors, the instructions in this category (including `jalr` and `jr`) are coded as `000000`. They are then identified by the function code in the 6 bits 26 through 31.

– Unconditional branches (example: `jal function`):

0 5	6 31
Instruction code	Offset (26)

– Conditional branches (example: `bnez r3,label`):

0 5	6 10	11 15	16 31
Instruction code	Register		Offset (16)

The format for conditional branching is `beqz rk,address` or `bnez rk,address`. The test is performed in the content of a register (`rk` $=$ 0 or $\neq$ 0), the value of which is initialized by the instructions `slt` (`set if lower than`), `slti`, `sgt`, `sgti`, etc.

11.2. Executing an instruction

11.2.1. *Reading and decoding an instruction*

Each instruction starts with phases F and D (Figure 11.2). During phase F, the instruction is read in memory and the program counter is updated, either to the current value + 4, or based on the result of an address calculation performed because of a branch.

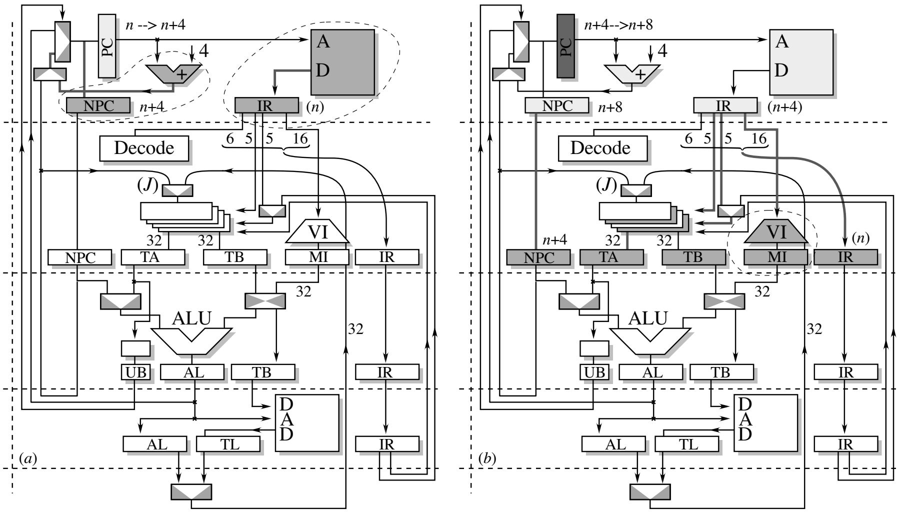


Figure 11.2. *a) Reading and b) decoding phases of the instruction with the address n. We have assumed that the following instruction is at the address $n + 4$.*

REMARK 11.2.–

– Phase F includes two half-phases denoted by dashed lines in Figure 11.2 (a): the first corresponds to the reading of the instruction from M_p to F.IR, the second to the incrementation of the PC (PC and NPC receive $n + 4$). PC behaves like the registers in Example 10.2.

– During the decoding phase, the transfer of the operands from the registers to buffers TA and D.TB is performed. If there is an immediate value (operand or offset), it is extended to 32 bits (dashed line in Figure 11.2, (b)). During phase D, the following instruction is read at $n + 4$ and PC is incremented (PC and NPC receive $n + 8$).

11.2.2. *Memory read*

Consider the instruction `lw r1,10(r3)` that reads in memory a 32-bit word at the address `10+[r3]` and stores it in register `r1` (Figure 11.3).

– In phase D (Figure 11.3 (a)), `r3` is transferred to TA and the offset 10, extended to 32 bits, is stored in MI.

– In phase E (Figure 11.3 (b)), the address is calculated by adding the offset and `r3` (loaded in E.TA during phase D). The instruction code (or at least the number of the target register) is copied into E.IR.

– During phase M (Figure 11.3(c)), the operand whose address is in E.AL is loaded into buffer M.TL. The instruction is copied into M.IR.

– During phase W (Figure 11.3(d)), the operand is copied into the target register `r1`, whose number is provided by M.IR.

Phase W is problematic. `r1` is modified while another instruction is being decoded (structural conflict). A solution is to perform write operations in registers in the first half-cycle and read operations in the second (see Figure 10.6).

11.2.3. *Memory write operations*

Consider the instruction `sw 10(r3),r1` that stores `r1` at the address `10+[r3]` (Figures 11.4 and 11.5).

– In the decoding phase, `r3` is transferred to TA and `r1` to TB. The rest is similar to decoding for the memory read operation.

– The address calculation is performed in the execution phase.

– The memory write operation (Figure 11.5) of the element of data is performed in phase M. Phase W does not lead to any action being taken.

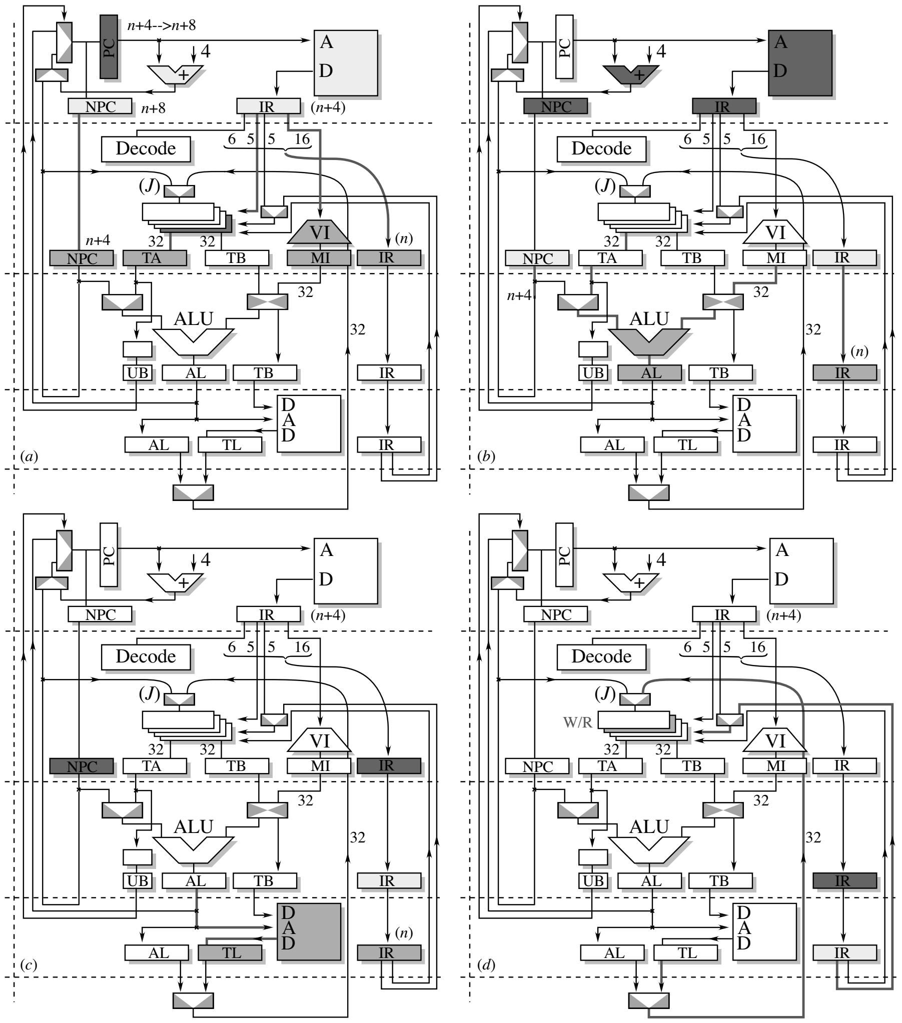


Figure 11.3. *Execution of a memory read operation*

11.2.4. *Register to register operations*

Consider the instruction `add r1,r2,r3` that adds the contents of `r2` and `r3` and stores the result in `r1` (Figures 11.6 and 11.7).

– In phase D, `r2` is transferred to TA and `r3` to TB. The rest is similar to decoding in the memory read operation.

– The sum is performed in the execution phase.

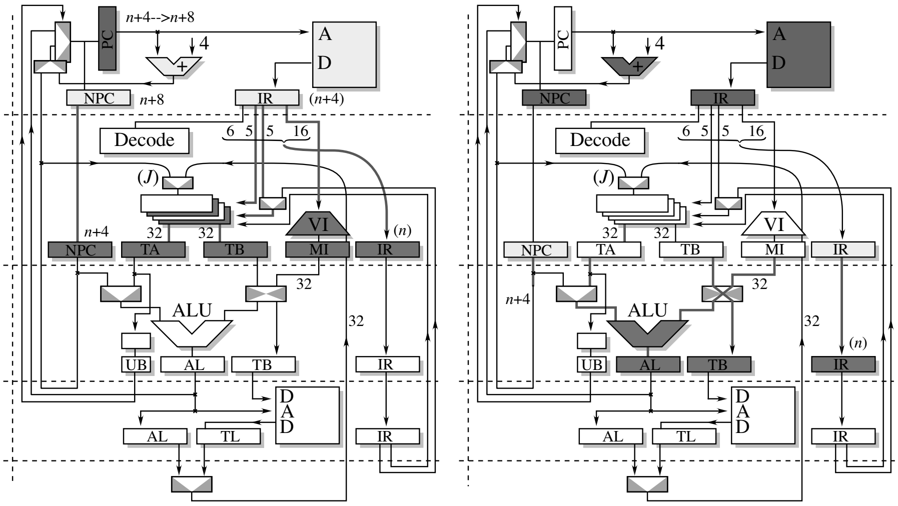


Figure 11.4. *Execution of a memory write operation: phases D and E*

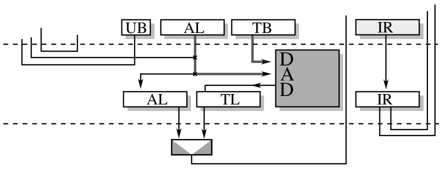


Figure 11.5. *Execution of a memory write operation: phase M*

– Phase M gives rise to a simple transfer.

– The rewrite in `r1` is performed in phase W.

11.2.5. *Conditional branching*

Consider the instruction `beqz r3,tag`. If the content of `r3` is equal to 0, the branching is performed at the address `tag`, otherwise we proceed to the instruction in $\mathrm{PC}+4$. `tag` is an offset d coded in 16 bits and the calculated address is $\mathrm{PC}+4+d$. We assume that the branch is normal, which in this case means that pipeline operation is suspended until the result of the test is known (Figures 11.8 and 11.9).

– The branch decoding puts on hold the updates of PC and F.NPC. The instruction read operation that follows the branch is performed but its handling is suspended. The pipeline *stalls*, i.e. switches to a “wait state” (first penalty).

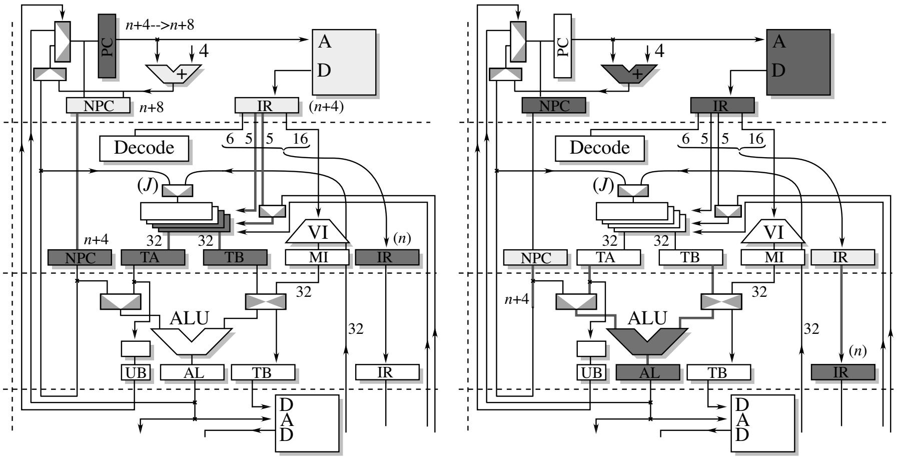


Figure 11.6. *Register to register operation, phases D and E*

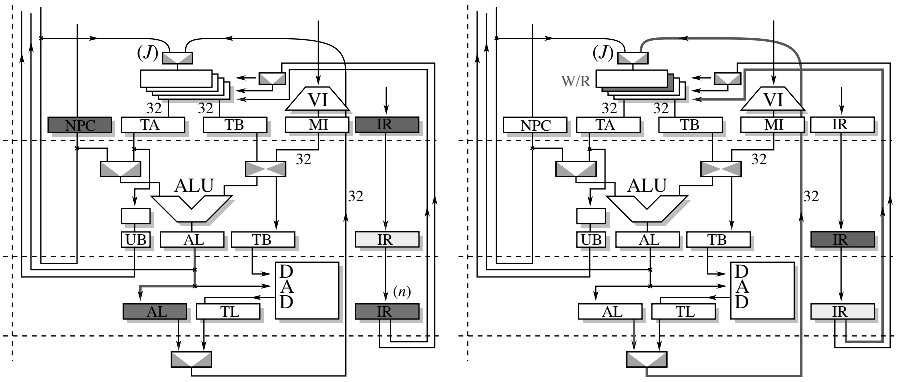


Figure 11.7. *Register to register operation, phases M and W*

– The calculation of the following address is performed in phase E, at the same time as the test of `r3` (register E.TA). The following instruction is always put on hold (second penalty).

– In phase M the loading of PC depends on UB, which was previously set by a `set` type instruction and is used by the branch.

The update of PC is not performed early enough to allow for the following instruction to be read in the same phase. This read operation can only be done in the next phase (third penalty).

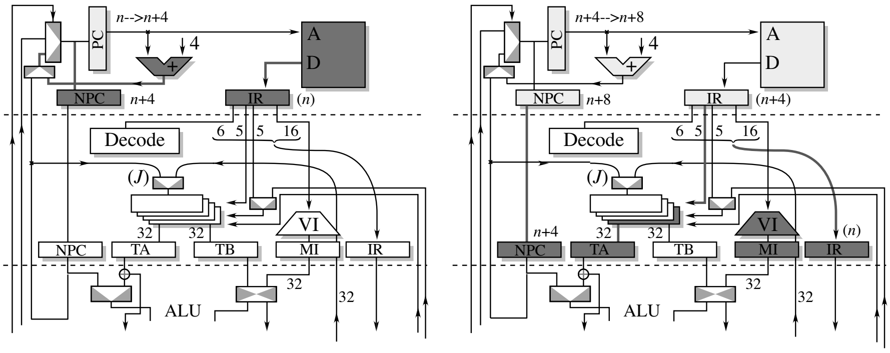


Figure 11.8. *Branch execution: phases F and D*

Figures 11.9 (b) and (c), respectively, show the situations “branch taken” and “branch not taken”.

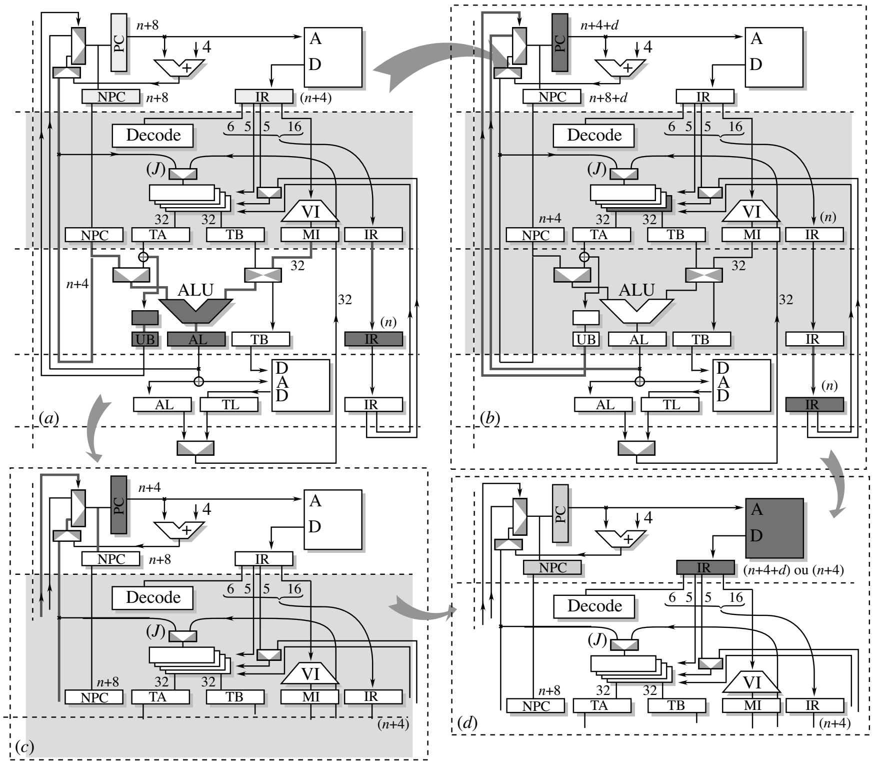


Figure 11.9. *Branch execution: phases E, M and W*

REMARK 11.3.– The instruction set includes two function calls: `jal tag` (*Jump and Link*) and `jalr rk` (*Jump and Link Register*). The return address PC+4 is stored in register `r31`. The return can be performed by `jr r31` (*Jump Register*), which loads the program counter with the content of `r31`. The link (J) in Figure 11.1 is used to recover the return address in `r31`. The task of handling a stack mechanism is left to the program.

11.2.6. *Instruction with immediate addressing*

The instruction `add r1,r3,#3` adds the constant 3 to the content of register `r3` and writes the result in register `r1`.

– The 32-bit extension to the immediate operand #3 is added to the content of the buffer register TA, and stored in E.AL in the execution phase.

– The two cycles M and W complete the transfer to the target register `r1`.

11.3. Conflict resolution in the DLX

Consider these two instructions:

```
add r1,r2,#2   ; r1:=r2+2
and r3,r3,r1   ; r3:=r3+r1
```

They are in conflict over register `r1`. A solution is to delay the second instruction by three cycles (Figure 11.10).

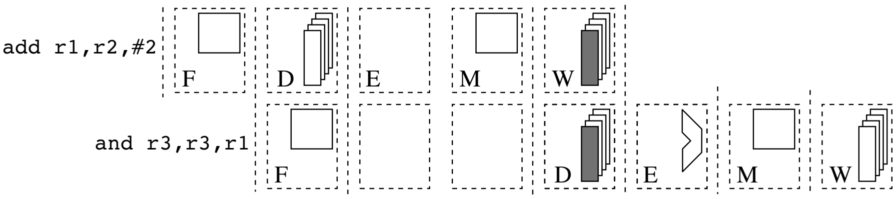


Figure 11.10. *Delay introduced in the sequence*

The conflict is resolved with a significant penalty. The forwarding technique makes it possible, through a hardware modification, to overcome this difficulty.

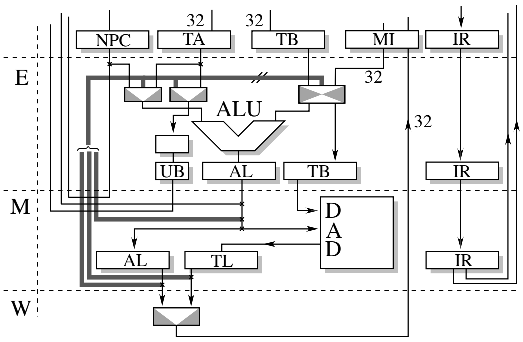


Figure 11.11. *The DLX architecture with forwarding*

11.3.1. *Forwarding techniques*

The basic DLX architecture can be modified to reduce the effect of hazards that lead to one or two suspended cycles (Figure 11.11).

This solution rests on the availability of results *before* they are written in phase W. The result of a calculation in E.AL, or an element of data read from memory and available in M.AL or M.TL, is transferred to the ALU as input. This modification is referred to as *forwarding* or *bypassing*.

Consider example 10.4. again

```
add r1,r2,#2  ; r1:=r2+2
and r3,r3,r1  ; r3:=r3+r1
sub r7,r1,r4  ; r7:=r1-r4
or  r8,r1,r5  ; r8:=r1 or r5
```

Figure 11.12 illustrates the program operation.

– Figure 11.12(a) shows the `add` instruction in its execution phase;

– Figure 11.12(b) illustrates the role of forwarding: the result of the sum, in E.AL, is not in `r1` yet, but is available for the `and`;

– Figure 11.12(c) shows the second instance of forwarding, which is used to execute the `sub` instruction. The result in M.AL is piped into the ALU;

– the last instruction is executed normally, using the content of register `r1` (Figure 11.12(d)).

Forwarding is not a solution to all dependency conflicts. When accessing the memory, there is no way around adding wait states to solve the problem. Consider the following example:

	1	2	3	4	5	6	7
`lw r1,0(r1)`	F	D	E	M	W		
`or r3,r1,r2`		F	D	...	E	M	W

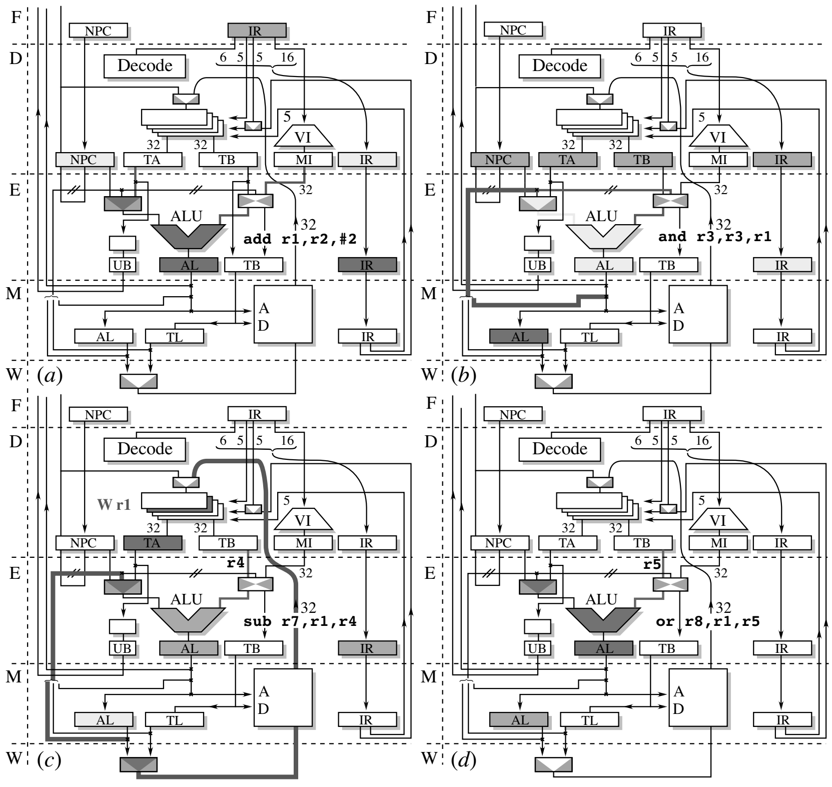


Figure 11.12. *Example of a program with forwarding*

One cycle can be gained with forwarding, but penalties cannot be completely avoided. The data read from memory are only available in M.TL in the second half-cycle of phase M. It can therefore only be used by phase E in the following cycle.

11.3.2. *Handling branches*

To decrease the number of penalties caused by branching, two calculations must be forwarded, a calculation involving the transfer, and the other the condition bit UB. This requires including an extra adder and transferring the handling of conditions to the decoding phase (Figure 11.13).

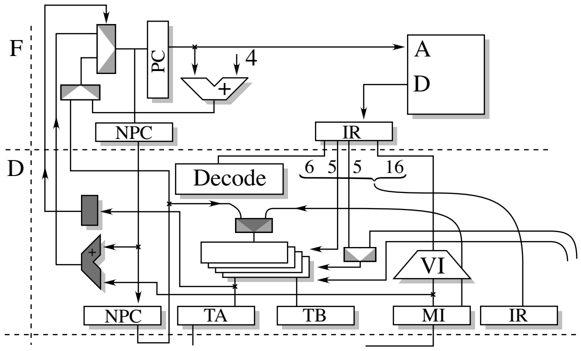


Figure 11.13. *The DLX architecture with forwarding*

Consider the instruction `beqz r1,tag`. If `r1` is equal to 0, the branch is taken to the instruction with the address `PC+4+tag`. If there is no dependency conflict over `r1`, and if the branch is taken, we get the following sequence:

I1		beqz r1,tag	F	D	E	M	W			
I2		lw r3,(r4)		F	.					
I3	tag:	add			$\boxed{\text{F}}$	D	E	M	W	
I4		sub				F	D	E	M	W

Figure 11.14 describes the different steps of the program.

Figure 11.14 shows phase F (`I1`) as two half-phases (Figures 11.14(a) and 11.14(b)), followed by the two half-phases of F(`I2`) (Figure 11.14(c) and (d)) and phase F(`I3`) (Figure (e)):

– Figure 11.14(a): the instruction is read at the address n (`beqz`);

– Figure 11.14(b): the program counter is updated ($n \rightarrow n + 4$);

– Figure 11.14(c): the instruction is read at the address $n + 4$ (`lw`);

– Figure 11.14(d): the program counter is updated ($n + 4 \rightarrow n + 4 + d$);

– Figure 11.14(e): phase F of the target instruction.

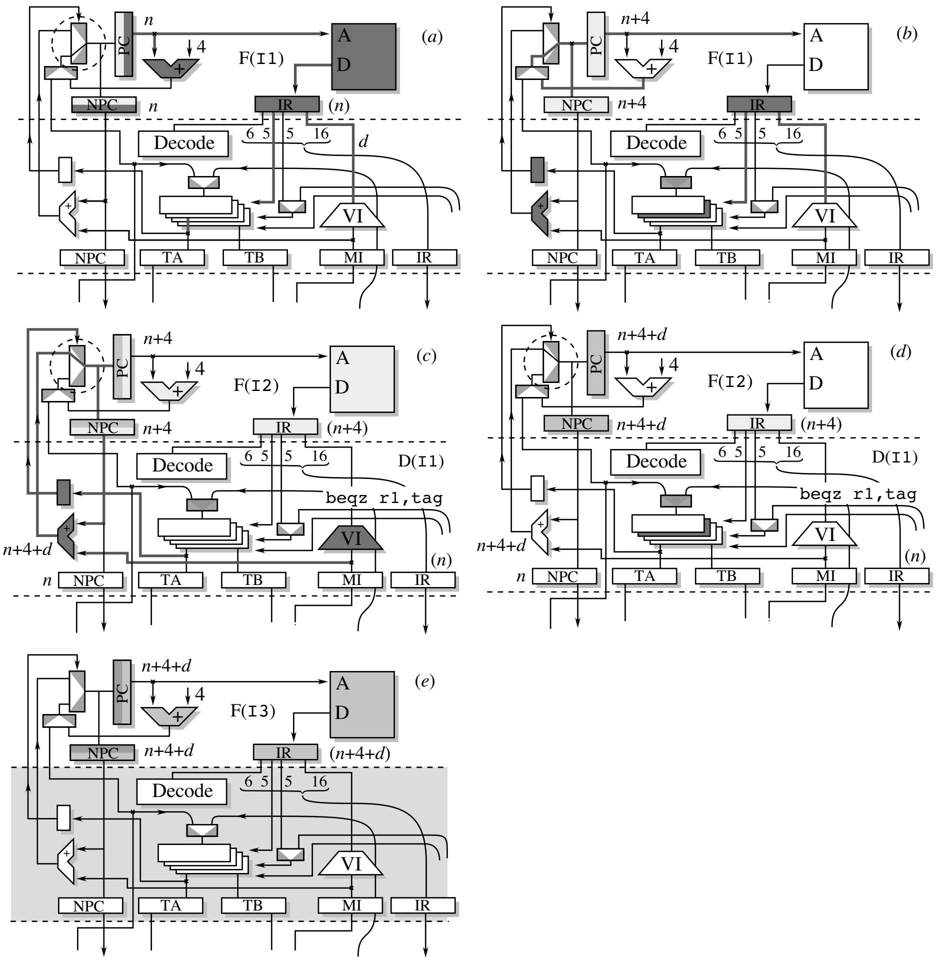


Figure 11.14. *The case of a taken branch: the penalty is equal to one*

The resulting penalty is equal to one. The only way to cancel this penalty is to handle it via software (see section 10.3.1 and example 11.2.).

EXAMPLE 11.1.– [Function calls and penalties] Consider the following program:

```
lw r2,0(r1)   ; r2 is used as an index
```

```
sll  r2,#2   ; r2 := r2 * 4
add r3,r3,r2 ; calculation of the address r3+r2
jal r3       ; jump and link
```

This program calls a function whose address is in r3. How many penalties are incurred from the execution of this program, if we assume that there is no dependency on the previous instructions?

SOLUTION.–the target instruction of jal is denoted by I_c:

– The instruction I2 is in conflict with I1 over r2. Forwarding is no help in this case (memory access).

– Because of the forwarding mechanism, the sum has access to the offset result with no penalty.

– The jal instruction cannot be handled because r3 is not available. The forwarding mechanism cannot completely eliminate the penalty.

I1	lw r2,0(r1)	F	D	E	M	W						
I2	sll r2,#2		F	D	D	E	M	W				
I3	add r3,r3,r2			F	.	D	E	M	W			
I4	jal r3					F	D	D	D	E	M	W
I5							F	.				
I_c								.	.	F		

The program generates a total of four penalties. In this case, the only way to solve this problem would be to move instructions that precede or follow these four instructions (section 10.3.1).

EXAMPLE 11.2.– Let us examine the following program:

```
loop1: lw r1,0(r2)    ; memory read ==> r1
       add r1,r1,r5   ; data processing
       sw r1,0(r2)    ; writing r1 into memory
       addi r2,r2,#4  ; next address
       subi r4,r4,#1  ; counter r4:=r4-1
       bneqz r4,loop1
       I1
```

We assume that forwarding is applied to the data and the branching. The execution of the first iteration can be summarized as follows (the first instruction after the loop is denoted by I1):

– a wait cycle is introduced by the processor after detecting the conflict between the `lw` and the `add` that follows;

– two cycles are lost because of the `bneqz` while waiting for `r4` to be updated, then two more cycles to perform the address calculation (penalty from the conditional branch).

```
loop1:  lw r1,0(r2)      | F  D  E  M  W
        add r1,r1,r5     |    F  D  D  E  M  W
        sw r1,0(r2)      |       F  .  D  E  M  W
        addi r2,r2,#4    |             F  D  E  M  W
        subi r4,r4,#1    |                F  D  E  M  W
        bneqz r4,loop1   |                   F  D  D  D  E
        I1               |                      F  .  .  F
```

There is a penalty of four for each iteration, with each loop consisting of 10 cycles. The penalty generated by the `lw` can be eliminated by placing it before an “independent instruction”. By placing the `subi` after the `lw`, we also gain two branch penalties, ending up with a penalty of one instead of four:

```
loop1:  lw r1,0(r2)      | F  D  E  M  W
        subi r4,r4,#1    |    F  D  E  M  W
        add r1,r1,r5     |       F  D  E  M  W
        sw r1,0(r2)      |          F  D  E  M  W
        addi r2,r2,#4    |             F  D  E  M  W
        bneqz r4,loop1   |                F  D  E
        I1               |                   F  F
```

The execution of a loop then takes seven cycles.

11.4. Exercises

Exercise 11.1 (Rescheduling) (hints on page 335). Consider the program:

```
a := b + c;
d := a - f;
e := g + h;
```

executed on a DLX without any optimization, and with every variable coded as 32-bit integers.

We will not worry for now about optimizing the code. We assume that a compiler generates the sequence:

```
;===== a=b+c
lw r1,b[r0]
lw r2,c[r0]
add r3,r1,r2
sw a[r0],r3
;===== d=a-f
lw r4,f[r0]
lw r3,a[r0]
sub r5, r3,r4
sw d[r0],r5
;===== e=g+h
lw r6,g[r0]
lw r7,h[r0]
add r8,r7,r6
sw e[r0],r8
```

1) Estimate the penalties.

2) Go over the program scheduling and try to minimize the penalties incurred from dependencies.

Exercise 11.2 (Branches) (hints on page 335). Consider the DLX architecture, and assume that no waiting period is necessary to access data. The initial value of `r3` is assumed equal to `r2`$+400_{10}$.

```
loop: lw r1,0(r2)
      addi r1,r1,#1
      sw 0(r2),r1
      addi r2,r2,#4
      sub r4,r3,r2
      bnez r4,loop
```

1) Consider the DLX in Figure 11.1 (forwarding, which will be discussed later, is not implemented). We assume that the branch "empties" – loss of two cycles, which causes a delay of three cycles, since a *Fetch* operation is again executed – the pipeline and that a read operation and a write operation are performed on the register bank in the same clock cycle.

Evaluate the number of cycles necessary to execute this loop.

loop: lw r1,0(r2)																		
addi r1,r1,#1																		
sw 0(r2),r1																		
addi r2,r2,#4																		
sub r4,r3,r2																		
bnez r4,loop																		

2) Now consider the DLX architecture with forwarding (Figure 11.11). We do not wait for the *Write* phase before using the results available in E.AL, M.AL or M.TL.

We assume that we are using a “normal” branch (three penalty cycles).

Evaluate the number of cycles it takes to execute this loop (bear in mind that forwarding does not solve every conflict!).

loop: lw r1,0(r2)																		
addi r1,r1,#1																		
sw 0(r2),r1																		
addi r2,r2,#4																		
sub r4,r3,r2																		
bnez r4,loop																		

3) We now assume that, in addition to the previous forwarding, we implement branch forwarding (penalty = 1) represented in Figure 11.13.

The branch used at the end of the loop is *delayed*. Evaluate the number of cycles needed to execute this loop.

Simply modify the program to optimize its execution time (by moving around one or several instructions).

loop: lw r1,0(r2)																		
addi r1,r1,#1																		
sw 0(r2),r1																		
addi r2,r2,#4																		
sub r4,r3,r2																		
bnezd r4,loop																		

Exercise 11.3 (Forwarding) (hints on page 337). Consider an 8-level DLX architecture where phases F, X, and M are now executed over two cycles, respectively, F1/F2, X1/X2, and M1/M2.

F1	First *Fetch Instruction* cycle.
F2	Second *Fetch Instruction* cycle.
D	Decoding, registers are loaded, calculation of the branch address begins.
X1	Execution starts, the condition is tested, and the calculation of the address branch ends.
X2	Execution ends, the effective address or the result of the calculation in the ALU is available.
M1	Memory access cycle starts, and Write Back of the calculation result is performed.
M2	End of the memory access cycle.
W	Write Back for a memory read instruction.

Draw the execution diagrams for the following sequences of instructions. Then, minimize the penalties by suggesting data and branch forwarding.

1) Example 1:

`add r1,r2,r3`												
`sub r4,r5,r6`												
`bnez r1,suite`												

2) Example 2:

`lw r1,0(r2)`												
`sub r4,r5,r6`												
`...`												
`...`												
`bnez r1,suite`												

3) Example 3:

`add r1,r2,r3`												
`sw 0(r4),r1`												

Exercise 11.4 (Cache operation) (hints on page 337) Processors dedicated to digital signal processing (DSPs, short for *digital signal processor*) are widely used in the field of telecommunications. Every cell phone is equipped with these components, in particular for handling communication channel equalization and voice coding/decoding. One such processor is the ADSP-2106x Sharc, which features a 32-bit floating-point unit (IEEE-754). It is built with a Harvard internal architecture and three buses: for addresses (24 bits), for data (40 and 48 bits for data memory and program memory, respectively) and for inputs and outputs. Instructions are handled in three "pipelined" cycles corresponding to the following phases: *Fetch* (F), the

instruction is read in the instruction cache or in program memory, *Decode* (D), the instruction is decoded, and *Execute* (X), the instruction is executed. Memory access is performed in this last phase. 48 bits are used for coding instructions, 32 for data memory, and 24 for the address buses.

The Sharc processor's performances were given 40 MIPS and 80 to 120 MFLOPS for a 40 MHz clock (these figures correspond to the earliest versions of this processor).

The cache memory of the Sharc processor is described in the documentation of its manufacturer, *Analog Devices*, as follows:

> The ADSP-2106x's on-chip instruction cache is a two-way, set-associative cache with entries for 32 **instructions**. Operation of the cache is transparent to the programmer. The ADSP-2106x caches only instructions that conflict with program memory data accesses (over the PM Data Bus, with the address generated by DAG2 on the PM Address Bus). This feature makes the cache considerably more efficient than a cache that loads every instruction, since typically only a few instructions must access data from a block of program memory.
>
> Because of the three-stage instruction pipeline, if the instruction at address *n* requires a program memory data access, there is a conflict with the instruction fetch at address $n + 2$, assuming sequential execution. It is this fetched instruction ($n + 2$) that is stored in the instruction cache, not the instruction requiring the program memory data access.
>
> If the instruction needed is in the cache, a "cache hit" occurs – the cache provides the instruction while the program memory data access is performed. If the instruction needed is not in the cache, a "cache miss" occurs, and the instruction fetch (from memory) takes place in the cycle following the program memory data access, incurring one cycle of overhead. This instruction is loaded into the cache, if the cache is enabled and not frozen, so that it is available the next time the same instruction (requiring program memory data) is executed.

1) Draw a diagram representing the execution of three instructions in the pipeline, and outline the problem caused by accessing the program memory.

2) Based on the documentation, describe the characteristic features of this cache memory.

3) What does the phrase "if the cache is enabled and not frozen" mean (see Chapter 8)?

4) The diagram representing the cache in the ADSP 2106x is given in the documentation as Figure 11.15.

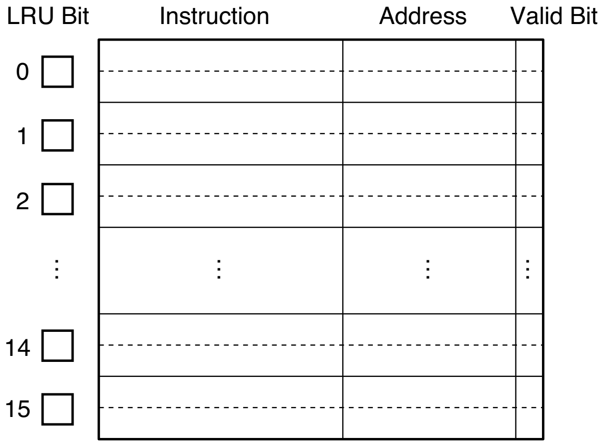


Figure 11.15. *Cache architecture in the ADSP 2106x*

- Draw a diagram similar to the diagrams in Chapter 8, emphasizing the comparators.

- What is the purpose of the *valid bits*?

- What is the purpose of the *LRU bits*?

5) Consider the program (given in the documentation for the 2106x) consisting of a loop inside which a subroutine `subrt` is called.

```
0x0100        bclcpt=1024, do endlp until lce ; start loop
0x0101        r0=dm(i0,m0), pm(i8,m8)=f3 ; dm --> r0
0x0101                                   ; and f3 --> pm
0x0102        r1=r0-r15;
0x0103        if eq call (subrt)         ; subroutine call
0x0104        f2=float r1                ;
0x0105        f3=f2*f2                   ;
0x0106 endlp: f3=f3+f4                   ;==== end of loop
0x0107        pm(i8,m8)=f3               ;
...
...
0x0200 subrt: r1=r13;
0x0201        r14=pm(i9,m9)              ; pm --> r14
...
0x0211        pm(i9,m9)=r12              ; r12 --> pm
...
0x021F        rts
```

Instructions that require access to the program memory are those that include the symbol `pm`.

Explain why this program is particularly poorly suited to this cache structure by specifying, for each access instruction to `pm`, the index used.

Exercise 11.5 (Branch forwarding) (hints on page 339). Consider the `max` function defined by:

$$\texttt{if}(\texttt{r1} > \texttt{r2})\texttt{thenr3} = \texttt{r1elser3} = \texttt{r2endif}$$

in a DLX with branch anticipation (penalty equal to 1). The unoptimized program is the following:

```
       sgt r4,r1,r2  ; set r4 if r1 > r2
       beqz r4,label1
       nop
       add r3,r2,r0
       j label2
       nop
label1: add r3,r1,r0
label2: ...
```

Write an optimized version of this code.

Exercise 11.6 (Loop with annul bit) (hints on page 339). Consider the loop:

```
; for (i=0;i<nmax;i++){I1; ...; Ip;}
        lw r5,nmax(r0)
        add r4,r0,r0
        j label2
        nop
label1: J1
        ...
        Jn
        add r4,r4,#1
label2: slt r6,r4,r5
        beqz r6,label1
        nop
```

The sequence J1, ..., Jn results from the assembly of I1, ..., Ip. We assume that the j and beqz instructions are delayed branches. The content of r0 is 0. Optimize this program fragment using the instructions j,a and beqz,na (notations from section 10.3.2.1).

Exercise 11.7 (Executing a subroutine call instruction) (hints on page 340). The instruction jalr rk stores the address of the next instruction in register r31 (return address) and branches to the address contained in rk. Using the architecture shown in Figure 11.1, describe the execution of the instruction jalr r1.

Chapter 12

Caches in a Multiprocessor Environment

In a multiprocessor architecture, managing information can lead to problems of *coherence*: the various copies of the same information can have different values. How these problems are handled depends on the architectures where they occur. We will make a distinction between two cases, *strongly coupled multiprocessors* and *weakly coupled multiprocessors*. In the former, the processors and their caches are connected to a shared memory bank via a bus, an architecture now typical of most microprocessors on the market (Figure 12.1).

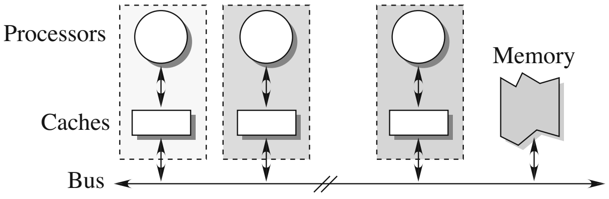


Figure 12.1. *Strongly coupled multiprocessors*

In these systems, *synchronous* devices can ensure coherence. On the other hand, when exchanges between systems cannot be synchronized, it is as if information is traveling on an interconnection network (Figure 12.2).

Note that coherence is not unique to multiprocessors. On a single processor system, *input–output operations* can also lead to coherence issues: accessing a memory word during an input–output operation, for example, can modify the value of a variable that is also referenced in the memory cache.

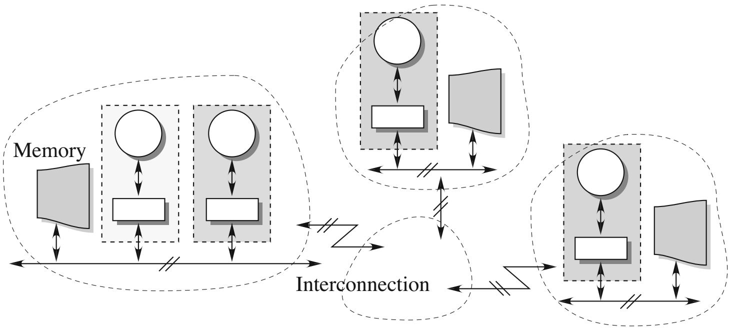


Figure 12.2. *Weakly coupled processors*

In broad terms, coherence protocols must guarantee that:

– the execution of a program by any processor is “observed” in the same order by the other processors;

– operations “executed” by a series of instructions produce results in the order defined by this series of instructions;

– the values read are the “last values written” (the “most recent”) or indicated as such (this concept of the last written value will be specified later).

12.1. Cache coherence

The *coherence* of information between caches and memory is a fundamental element in the operation of strongly coupled multiprocessor systems, since it ensures that a processor has access to the “last value” that was written, relative to the order in which the information needs to be written.

12.1.1. *Examples*

EXAMPLE 12.1.– Consider two processors P_A and P_B equipped with the caches C_A and C_B, respectively. These caches share the same memory space M, and their lines can contain two words (Figure 12.3(a)). We will assume that the caches operate in the *write-through* mode.

The execution sequence is as follows:

– P_A accesses the memory word with the address A, the tag X, containing the value 10. A line is loaded into C_A (Figure 12.3(b)).

– P_B accesses the same memory word. C_B is updated with this value Figure 12.3(c).

– P_A writes the value 0 in A (Figure 12.3(d)) in the *write-through* mode (we would face the same problem in *write-back*). There is a loss of coherence. Any access to A by P_B will trigger a *hit* on a value that is wrong, since it is not the result of the "latest" update.

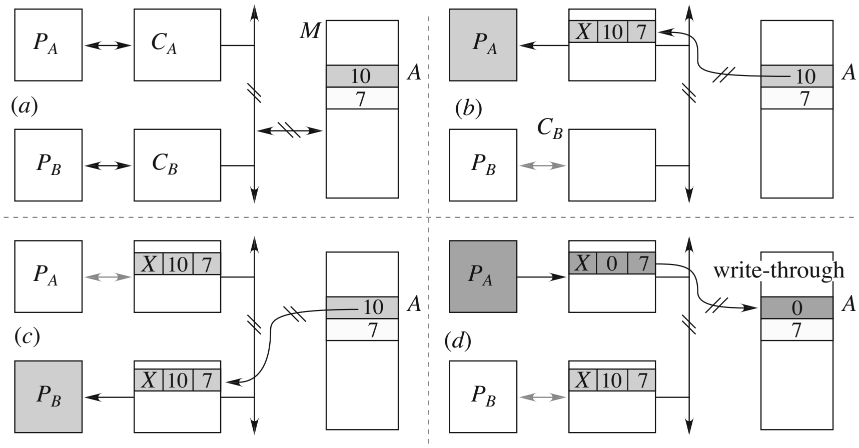


Figure 12.3. *Loss of coherence*

Note that the same problem is true when performing a write operation at the address $A + 1$, since the information is located on the same line with the tag X.

EXAMPLE 12.2.– Consider again the previous case of two processors P_A and P_B, but in the *write-back* mode (Figure 12.4).

– C_A and C_B have both been loaded (Figure 12.4 (a)) with the content of the memory word with the address A_1 (tag X, content 10).

– P_A modifies this information by setting it to 0 (Figure 12.4(b)).

– We assume that P_A needs line L with tag X (the line chosen by an LRU algorithm) to load into it the information at the address A_2 and with tag Y. Since the information contained in line L has been modified, this line must be copied into memory (Figure 12.4(c)). The memory word with the address A_1 now contains the value 0.

– The line containing the information at the address A_2 is loaded into line L (Figure 12.4(d)). As in example 12.2, coherence is not maintained. The values in A_1 and C_B are different. C_B should also be updated.

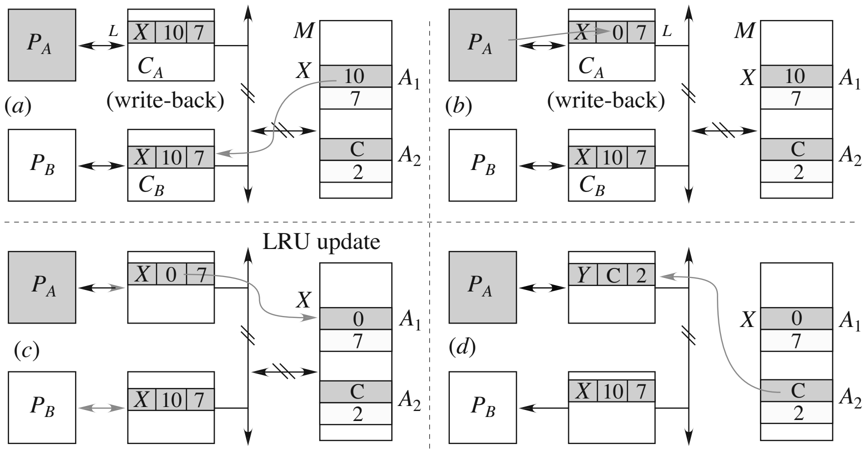


Figure 12.4. *Loss of coherence during an LRU replacement*

12.1.2. *The elements to consider*

The mechanisms implemented to ensure the coherence of information must take into account a certain number of "local" or "distant" elements or events. By local "relative to a processor", we mean an element or event that depends on that processor:

– the "normal" operating mode, in other words handling *misses* and the algorithm implemented for replacing lines, generally LRU;

– memory accesses that do not involve the cache, for example inputs and outputs;

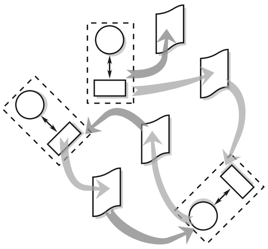
– instructions that directly manipulate the content of the caches.

"Distant" elements or events are related to the operation of other processors and affect the local processor. In example 12.1, the modification by P_A of the shared line is, to P_B, a distant event.

12.1.3. *Definition of coherence*

We indicated that accessing an element of information should return its most "recent" value, in other words the value after the last update. In the example illustrated by Figure 12.3(d), processor P_B has access to an "old" version of the information. To avoid this problem, it would have had to know that this data had been modified elsewhere.

The concept of old information used here should be defined more precisely, since it refers to an *order*. This can be the order of the instructions in a program, the order in which instructions "end" (the results are ready to be stored in memory and are "sent"), or the order of the actual writing (*termination*) of the results. The propagation times are never the same. We will discuss these points in greater detail in section on 12.5.



12.1.4. *Methods*

There are a variety of methods used to ensure coherence. The two parameters to consider are ease of implementation and performance expectations. Among others, we can mention the following methods:

– If information is modified in one of the caches, the decision can be taken to modify it in all of the caches that hold it. This method is known as *broadcasting/hardware transparency*.

– It is also possible to prevent shared information from being copied into the caches. This is referred to as *non-cacheable memory systems* (this applies in particular to all of the input–output device addresses).

– In order to account for the difficulties related to propagation time, it may be useful to make a copy of a line as soon as it is modified (*dirty* or *modified*) and to mark as *invalid* any possible copies.

These methods are rather inefficient in terms of cache use. In practice, it is preferable to preserve information as long as possible and to limit exchanges. There are two main categories of methods used for maintaining coherence, depending on whether the information necessary to this coherence is distributed (local) or centralized (global):

– In the first case – local information – we will refer to the concepts of *broadcasting* and *snooping*. A snooping tool is used to inform a cache of problematic accesses. Protocols that rely on such a device, known as a *snooper* or a *bus watcher*, are referred to as *snooping protocols*. The advantage of implementing this scheme is that it requires modifying only the cache controller, not the processor or the memory.

We can define two strategies:

- The first consists of invalidating shared information when it is modified by another processor. The writer acquires an *exclusive copy* of the information. This is known as a *write-invalidate* algorithm.

- The second consists, for the processor that modifies data, of notifying the other processors that it is performing a write operation. The other processors update their cache to maintain coherence. This is known as a *write-update* algorithm;

– In the second case – global information – we use a *directory* (*directory-based coherence*), or table, where each entry corresponds to a line in memory, and contains information indicating the presence of this information in one of the caches.

The main pros and cons of the two methods are the following:

– The scalability of snooping – essentially changes in the number of caches – is poor, and a large portion of the bandwidth is used on the shared bus. However, the protocol is reactive. These methods are more often used with strongly coupled microprocessors.

– Directories scale better, bandwidth use is limited, but the latency is high. These methods are more often used with weakly coupled multiprocessors.

12.1.4.1. *Broadcasting and snooping, an example*

Consider the system in example 12.1, assuming that write operations are performed using the *write-back* method (Figure 12.5).

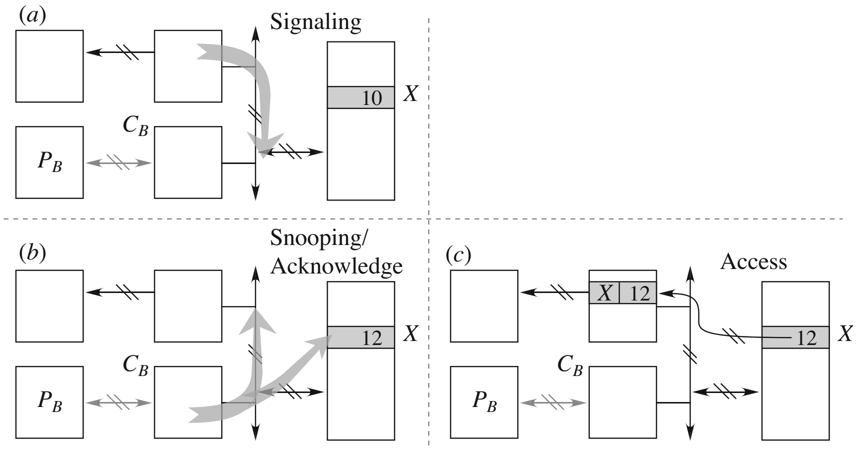


Figure 12.5. *Snooping and updating before access agreement*

– P_A needs to read X in memory (Figure 12.5(a)). Its read request is indicated on the control bus. The read operation can only be performed with agreement from the other cache controllers.

– If they have X, the other controllers detect this read attempt (by *snooping*). The processor P_B which modified the data, copies this information X into memory (Figure 12.5(b)) and acknowledges the signal.

– After the acknowledgment, P_A can perform the memory access (Figure 12.5(c)).

12.2. Examples of snooping protocols

In order to ensure information coherence, caches need to exchange information. The fastest way consists of using a certain number of communication lines. The minimum requirement is a valid line (line V, validity: $V = 1$), a line indicating a modification (line M, modification: $M = 1$) and a line indicating that information has been shared (line S, sharing: $S = 1$). This set of three lines on the bus is enough to distinguish eight states for the cache lines, with only five possible states (Table 12.1).

V	M	S	State	Comment
0	ϕ	ϕ	$\mathbb{I}$	The line was never loaded.
1	0	0	$\mathbb{E}$	Only one cache has this line. There is coherence with memory.
1	0	1	$\mathbb{S}$	At least one other cache has this line. Coherence with memory is not certain.
1	1	0	$\mathbb{M}$	The line has been modified and is the only one in this state. Coherence depends on the update policy (*write-back* or *write-through*).
1	1	1	$\mathbb{O}$	The line has been modified and is also in other caches with the same content.

Table 12.1. *Cache states*

In the following sections, we will present the MSI, MEI, MESI and MOESI protocols, which use these states. We will assume that the caches operate in the *write-back* mode and that a *miss* in a write operation triggers a cache update. The rules governing coherence with memory will be specified for each of these protocols.

In what follows, P_0 will refer to the *local* processor, C_0 to its cache, while P_i will refer to *distant* processors and C_i to their caches.

12.2.1. *The MSI protocol*

The MSI (modified, shared, invalid) protocol applies to caches that operate in *write-back*, in which each line is in one of three states: *modified* ($\mathbb{M}$), *shared* ($\mathbb{S}$) or *invalid* ($\mathbb{I}$), the significance of which is as follows:

– $\mathbb{I}$: invalid line. The information stored here is not reliable. Any access results in a *miss*.

– $\mathbb{M}$: if a line of the cache is in the $\mathbb{M}$ state, it means that it is the *only* valid copy. The processor is referred to as the *owner*. The copy in memory may or may not be valid. The line must be copied into memory before being replaced.

– $\mathbb{S}$: the line is in the cache and has never been modified. Other caches may host a copy of this line. *Coherence is ensured* with memory.

12.2.1.1. *Interactions with the LRU protocol*

Line replacements triggered by the execution of the LRU protocol interfere with the coherence protocol, since LRU modifies the state of the lines when they are replaced. The operation of the LRU protocol can be represented by the diagram in Figure 12.6. A transition corresponds to a read miss (RM) or a write miss (WM).

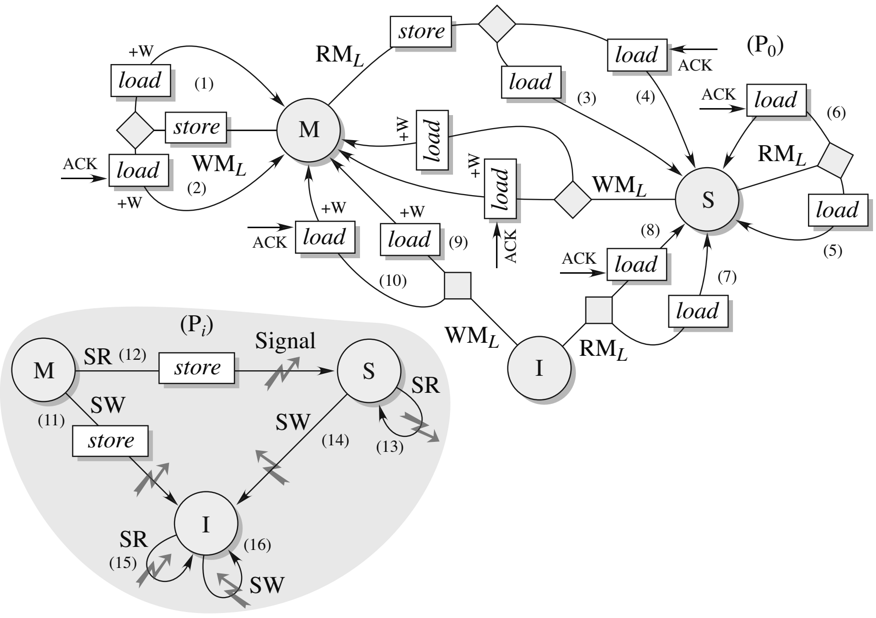


Figure 12.6. *Part of the protocol implemented because of LRU*

The overall protocol operation must account for signals originating from the snooping module. The notations are as follows:

– RM_L (*Read Miss* LRU) and WM_L (*Write Miss* LRU);

– SW (*Snoop Write*) detection of a write operation performed on another processor;

– SR (*Snoop Read*) detection of a read operation performed on another processor;

– ACK (acknowledgment): the symbol $\boxed{\textit{load}}\xleftarrow{\text{ACK}}$ indicates that a line in the cache is imported from memory upon acknowledgment from another processor;

– *store*: storage in memory;

– *load*: a line in memory is loaded into the cache. The notation "+W" indicates that this line has been modified.

Consider for example a read operation that results in a *miss* (event RM_L) on a line ℓ in the $\mathbb{S}$ state. The line needs to be replaced. This event can lead to two cases (transitions (5) and (6)):

– Case (5): no other processor P_i has the new line, or has it in state $\mathbb{S}$ (we have coherence). The line simply needs to be loaded from memory ($\boxed{\textit{load}}$).

– Case (6): another processor P_i has the new line in the $\mathbb{M}$ state. P_i must copy it into memory (*store*) before it can switch to the $\mathbb{S}$ state (SR detection, transition (12)). Once the update has been performed, P_i sends an acknowledgment and P_0 can load the new line from memory ($\boxed{\textit{load}}\xleftarrow{\text{ACK}}$).

If the line that needs to be replaced is in the $\mathbb{M}$ state, the mechanism is the same, but P_0 must first save the line in memory ($\boxed{\textit{store}}$).

REMARKS 12.1.– When P_i updates the memory, it can also simultaneously "send" the line to P_0, since the line travels on the shared bus, and is therefore accessible to C_0. This saves time in the execution of the protocol.

12.2.1.2. *The coherence protocol*

The protocol is described in Figure 12.7. Thin-line arrows (⟶) correspond to transitions between states, while the others (jagged arrows ↗) correspond to signals used for communication between C_0 and C_i. Events taken into account are referred to as RH (Read Hit), RM_I (Read Miss on Invalidation), WH (Write Hit), and WM_I (Write Miss on Invalidation).

Transitions correspond to the following actions:

– Transitions (1) and (2): these transitions correspond to successful read and write operations. The line stays in the $\mathbb{M}$ state and nothing is broadcast.

– Transition (3): write operation on a line in the $\mathbb{S}$ state. The line in question switches to $\mathbb{M}$ and the corresponding lines in the other caches (and therefore in the $\mathbb{S}$

state) are invalidated (response to the event ↯, SW is taken into account, transition (14)).

– Transition (4): read operation on a line in the $\mathbb{S}$ state. Nothing is broadcast and the line stays in the $\mathbb{S}$ state.

– Transitions (5) and (6): the line is in the $\mathbb{I}$ state. It is therefore going to be overwritten with new information. If this line exists nowhere else (or in the $\mathbb{I}$ state), or if it is in the $\mathbb{S}$ state in P_i (denoted by S+I in the diagram), the cache is simply going to be loaded with a line read from memory. If it is in the $\mathbb{M}$ state in a P_i, then that processor must update the memory (*store*, transition (11)) before P_0 loads the line in the cache to modify it. The state switches to $\mathbb{I}$ in P_i in any case.

As previously indicated, the information provided by C_i to update the memory can be simultaneously loaded into C_0.

– Transitions (7) and (8): the situation is the same as in transitions (5) and (6). The only difference is that no write operation is performed in the line after loading.

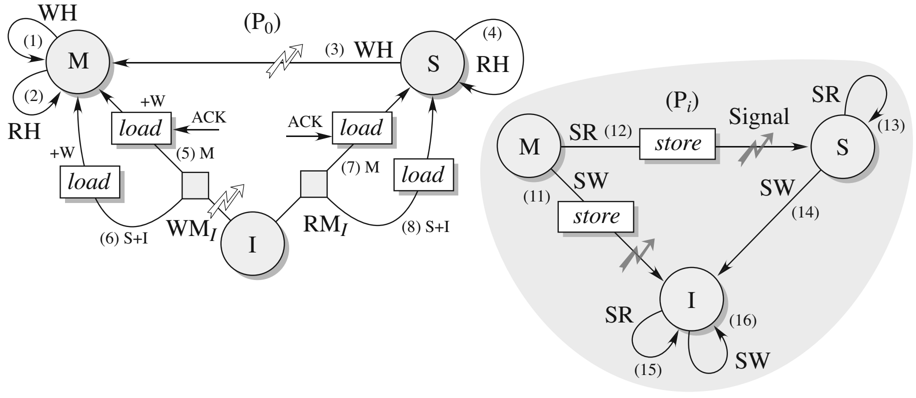


Figure 12.7. *Description of the MSI protocol*

12.2.2. *The MEI protocol*

In the MSI protocol, when a processor modifies a line that is absent from all the other caches, this line is loaded into memory and set to the $\mathbb{S}$ state. It then switches to the $\mathbb{M}$ state and an SW signal, which is not necessary in this case, is sent. It should be noted that this situation where a non-shared line is modified, along with the broadcasting of unnecessary SW signals, is very common.

The MEI protocol replaces the $\mathbb{S}$ state with an $\mathbb{E}$ state, indicating that this line is read by P_0 alone (*exclusivity* principle). As a result, access to a line in this state by

P_0 does not lead to either a transaction or a waiting period on the bus. All of the other copies (lines with the same tag) are in the $\mathbb{I}$ state.

Figure 12.8 sums up the operation of the protocol. It includes the case of a memory access that does not involve caches (input–output, DMA). However, it does not show events related to the LRU protocol.

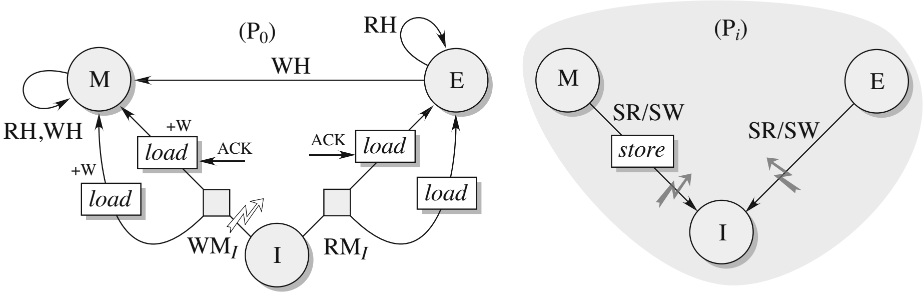


Figure 12.8. *Description of the MEI protocol: thin-line arrows correspond to transitions between states, the others (jagged arrows) to acknowledgement inputs*

The states of the MEI protocol are as follows:

– $\mathbb{I}$, *invalid* line: the information stored in this line is not reliable. Any access results in a *miss*.

– $\mathbb{M}$, the processor is the *owner* of the line. It is the *only* one that has a valid copy of this line. The copy in memory may or may not be valid. The line is copied into memory in case of a replacement.

– $\mathbb{E}$, the line involved, is found only in this cache. The corresponding information is *coherent* with the memorization system.

The cache in the MPC603e processor by Motorola® operates according to the MEI protocol, which is a subset of the modified exclusive shared invalid (MESI) protocol (section 12.2.3). This makes it possible to run PowerPCs, which implement it, with processors supporting the MESI protocol.

12.2.3. *The MESI protocol*

The MESI protocol is also known as the *Illinois protocol*. It is implemented in particular in Intel®'s Pentium 4 IA32 series processors, in the PPC601 by Motorola® and in the AMD486. Each line of the cache is in one of the four following states:

– *Modified* ($\mathbb{M}$): the line is *valid* and the associated information can only be found in this cache. The data have not been updated in the memory.

– *Exclusive* ($\mathbb{E}$): the line can only be found in this cache. The corresponding information is *coherent* in the memorization system.

– *Shared* ($\mathbb{S}$): the line is *valid* and the corresponding information can be found in at least one other cache. A "shared" line is always *coherent* in the memorization system.

– *Invalid* ($\mathbb{I}$): the information contained in this line is not reliable.

Figure 12.9 summarizes the operation of the MESI protocol. It shows a transition corresponding to the execution of a cache management instruction.

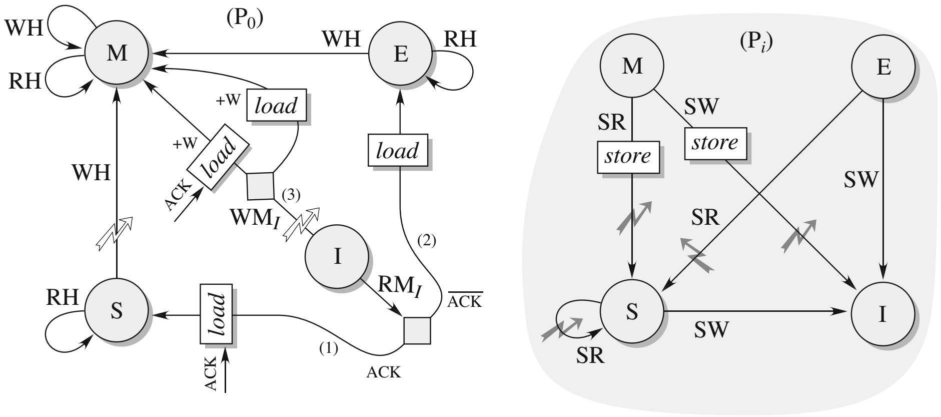


Figure 12.9. *Description of the MESI protocol*

– Transitions (1) or (2): the current state is $\mathbb{I}$. There is an RM. The switch to $\mathbb{S}$ or $\mathbb{E}$ depends on the broadcasts originating from the C$_i$. If the corresponding line is in the $\mathbb{M}$, $\mathbb{E}$ or $\mathbb{S}$ state, then the state switches to $\mathbb{S}$ in P$_0$. If there is no response, the state switches to $\mathbb{E}$.

– The other behaviors can be inferred directly from the MSI and MEI protocols.

REMARKS 12.2.–

– In principle, the $\mathbb{M}$ state is reflected by a line in the control bus. In PowerPCs™, this line is referred to as $\overline{\text{GBL}}$ and indicates (P_0 output line) to the other processors P_i (input lines of the P_i) whether the coherence needs to be checked. If there is a switch from $\mathbb{E}$ to $\mathbb{M}$, it is not necessary to broadcast it, which saves time when checking the coherence. In the case of MSI, this would not be possible. The switch from $\mathbb{S}$ to $\mathbb{M}$ is always broadcast.

– The $\mathbb{M}$ and $\mathbb{E}$ states are always *precise*, meaning that they must always reflect the true state of the line. The $\mathbb{S}$ state can be *imprecise*: if another processor switches the corresponding line to the $\mathbb{I}$ state, the local line does not switch to $\mathbb{E}$ even if it is in the situation specific to that state (it would be difficult to implement).

– The *Firefly* protocol, implemented on the Firefly processor by DEC, works with the MESI protocol, but in the *Write-Update* mode. When a line is loaded, its modification by a processor triggers the *Write-Through* and the update of all the other copies [THA 87]. All of the lines in the $\mathbb{S}$ state stay in the $\mathbb{S}$ state. Another variation, the MESIF protocol (MESI forward), is used in Intel's Nehalem architecture (level 3 cache). If a line is requested, the cache that has it in the $\mathbb{E}$ mode delivers it and switches to the $\mathbb{S}$ state, while the requester switches to the $\mathbb{F}$. It is the latter that will deliver the line the next time it is needed, meaning that the "provider" changes after each request.

EXAMPLE 12.3. (The PowerPC 601) – The PowerPC 601 is equipped with a "data-instruction" cache, which is set-associative with eight 32-kB blocks, and can operate in the *write-through* or *write-back* mode. Each line of a block contains sixteen 32-bit words organized in two 8-word *sectors*. Cache updates are done one half-line at a time. If, during an update, another half-line is invalid, its automatic update is performed with a low priority. This possibility can be inhibited by software.

In a multiprocessor environment, the cache update protocol is the MESI protocol. In the cache, each half-line (sector) is assigned two bits that code the state.

12.2.4. *The MOESI protocol*

The case where we accept that shared information may not satisfy coherence with the memory is called the MOESI protocol (Exclusive-Modified, Shared-Modified (Owned), Exclusive-Clean, Shared-Clean, Invalid) [BAU 04]. There can therefore be information that is modified and shared. This is done by introducing a *cache to cache* communication mechanism, which is used to avoid systematic memory updates whenever there is an attempt to access a modified line. The MOESI protocol is implemented in particular in the P6 by Intel, the successor to the first Pentium architecture, and in the AMD64 architecture. It is also used, with *Write-Update*, under the name Dragon protocol in the Xerox$^{\text{TM}}$ Dragon multiprocessor.

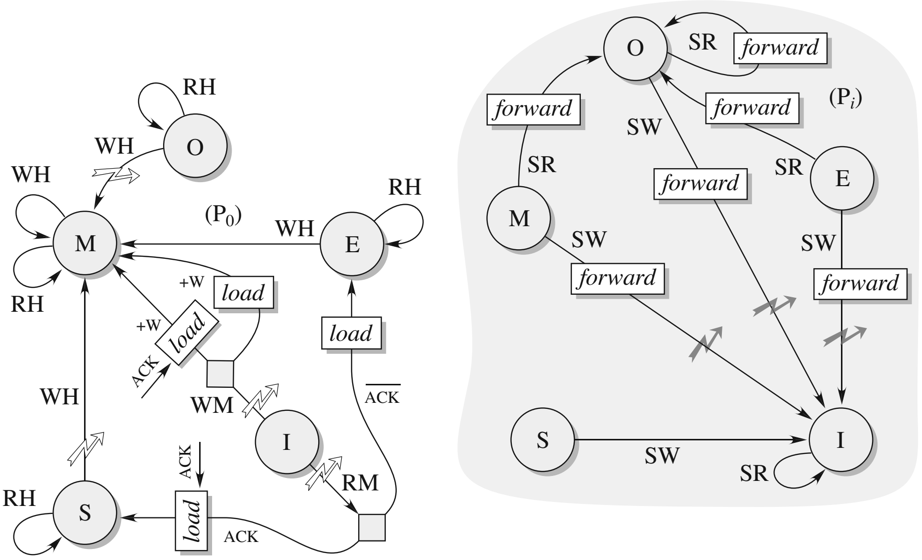


Figure 12.10. *Description of the MOESI protocol*

The states are as follows (Figure 12.10):

– $\mathbb{M}$: the current cache is the only cache with a *valid* copy of the line, and it has modified it. There is no *coherence* with memory.

– $\mathbb{O}$: the line is *valid* in the current cache and in at least one other cache. There is *no coherence* between cache and memory. The cache that has the line *owns* it: it owns the latest modified version and has propagated its content.

– $\mathbb{E}$: the line is *valid* in the current cache, and only in this cache. Let us assume that P_0 requests a line held by P_i in the $\mathbb{E}$ state. P_i provides this line without a memory write and switches it to $\mathbb{O}$. P_0 retrieves the line and switches it to $\mathbb{S}$. There is no coherence between caches and memory.

– $\mathbb{S}$: the line is *valid* in the current cache and in at least one other cache. There is not necessarily coherence with memory, unlike the other protocols discussed previously. Let us assume that P_0 is in the $\mathbb{I}$ state and requests a line in the $\mathbb{M}$ state in C_i. P_i provides it without a memory write and switches it to $\mathbb{O}$. P_0 retrieves the line and switches it to $\mathbb{S}$. There is coherence between caches, but not between caches and memory.

– $\mathbb{I}$: the line is *not valid*.

REMARKS 12.3.–

– If a line in P_i in the $\mathbb{O}$, $\mathbb{E}$ or $\mathbb{M}$ state is invalidated, its content is propagated and its state in P_0 switches to $\mathbb{M}$. There is *no copying* into memory.

– The only case of copying into memory is due to the LRU: if a line is $\mathbb{O}$ or $\mathbb{M}$, an LRU replacement leads to the line being copied into memory even if, in the case of $\mathbb{O}$, this is not necessary.

– The same case, for a line in the $\mathbb{S}$ or $\mathbb{E}$ state, is sometimes called a *silent evict*, because there is no copying into memory.

– $\mathbb{O}$, $\mathbb{E}$ and $\mathbb{M}$ all three convey a state of *ownership*: the cache that contains a line in this state has the responsibility to propagate its content if necessary.

– If a line ℓ_k is in the $\mathbb{O}$ state, the lines in other caches with the same tag will be in the $\mathbb{S}$ state. If ℓ_k is overwritten because of LRU, the memory becomes the owner and any new access to this line will be done through memory, and not through caches.

– When a processor modifies a line ℓ_k in the $\mathbb{S}$ state and switches it to $\mathbb{M}$, an alternative solution would be to switch to $\mathbb{O}$ after propagating the modification.

12.3. Improvements

A variety of methods have been proposed for improving the protocols described above:

– Decreasing bus traffic: broadcasting to a limited number of caches, performing only partial invalidations (*word invalidation protocol* [MIL 96]), etc.

– Using hybrid protocols (updates and invalidation): the (RST) protocol is an example of such a method. It is based on the Dragon and Firefly protocols [PRE 91] (see exercise 12.5).

– Observing read and write operations during the execution: if there are few writes between the reads, an update protocol is advisable. If it is rare to read the same data, or if there are many writes between reads, an update protocol should not be used. The read write broadcast (RWB) protocol is one of the first examples of this method [RUD 84].

12.4. Directory-based coherence protocols

Directory-based coherence protocols [TAN 76] manage caches using one or several controllers that send the commands to transfer from cache to memory, or from cache to cache, by limiting traffic to only what is strictly necessary (Figure 12.11).

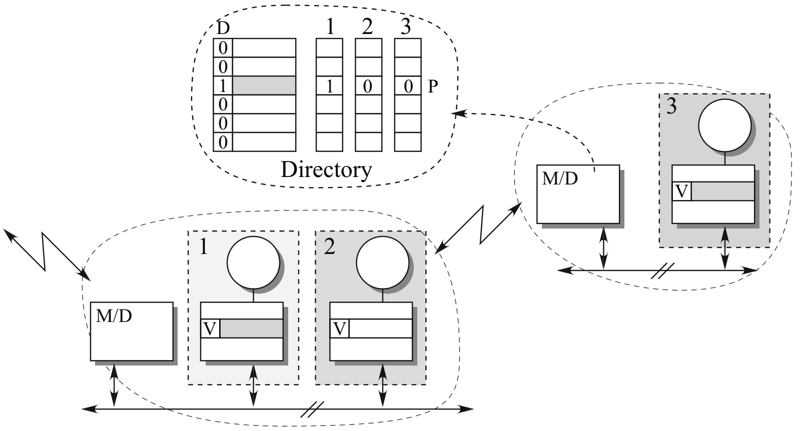


Figure 12.11. *Example of a directory structure: valid bit V, presence bit P, modify bit D, memory/directory M/D. A reference is present in caches P_1 and P_3. The line was modified ($D = 1$) in cache C_1*

Controllers maintain all of the information necessary to satisfy the coherence protocol: the references to the lines used are recorded into a table in memory, or in a memory annex, called a *directory*.

Each entry in the directory includes a modify bit D indicating whether the corresponding line has been modified, and a presence bit associated with each cache.

The original protocol proposed in [CEN 78] only allowed for one modified copy to be present in the system (invalidation protocol). Several structures have been proposed to limit the directory size without excessive increases in the volume of exchanges or latency. Another problem to consider is the number of processors (*scalability*).

EXAMPLE 12.4.– We focus on P_1, P_2 and P_3, which share the same memory space. The variable x is located in the memory attached to P_2. Lines can exist in the $\mathbb{M}$, $\mathbb{S}$ or $\mathbb{I}$ state. The following diagrams illustrate the execution of the sequence $R_1(x)$ (x is read by P_1), $W_1(x)$ (x is modified by P_1), $R_3(x)$ and $W_3(x)$:

– Initial situation: the indication “0 1 0” associated with line ℓ indicates that it is only present in the memory M_2 attached to P_2 (Figure 12.12).

– x is read by P_1: the read is broadcast to P_2, owner of line ℓ. The controller for P_2 sets to 1 the presence bit associated with P_1 in input ℓ; P_2 transmits ℓ to P_1, which changes it to the $\mathbb{S}$ state in C_1 (Figure 12.13).

– x is written by P_1: the write is broadcast to P_2, there is no invalidation to send to other processors since P_1 and P_2 are the only processors that have ℓ. In the directory for P_2, bit D switches to 1 and the presence bit switches to 0 for P_1 (Figure 12.14).

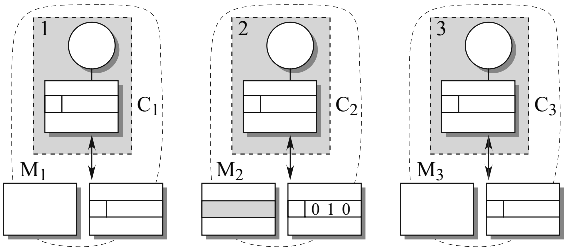


Figure 12.12. *Initial situation*

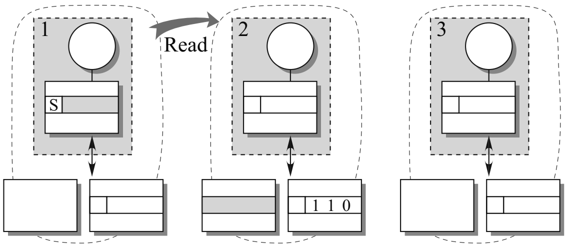


Figure 12.13. *x is read by P_1*

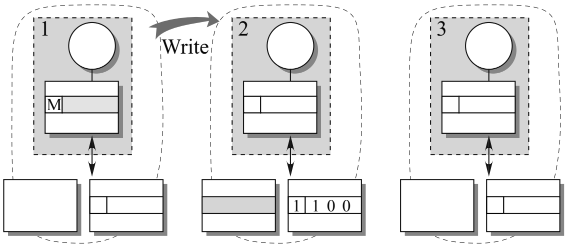


Figure 12.14. *x is modified by P_1*

– x is read by P_3: the read is broadcast to P_2, P_2 provides P_3 with the identity of the owner (P_1). P_3 requests ℓ from P_1, which returns the line to P_2 (in memory) and P_1. ℓ switches to the $\mathbb{S}$ state in P_1 and P_3. There is coherence between memory and caches (Figure 12.15).

– x is written by P_3: the write is broadcast to P_2, P_2 sends an invalidation signal to P_1. P_2 sets D to 1, along with the presence bit associated with P_3. ℓ switches to $\mathbb{M}$ in C_3 (Figure 12.16).

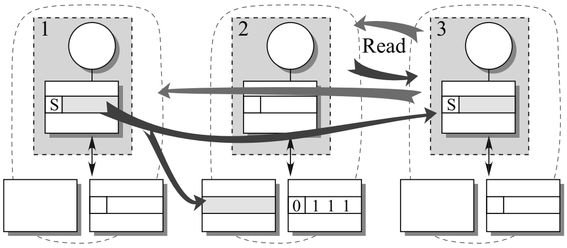


Figure 12.15. *x is read by P_3*

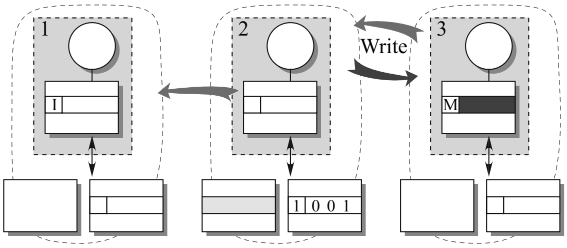


Figure 12.16. *x is modified by P_3*

12.5. Consistency

12.5.1. *Consistency and coherence*

Consistency defines rules of behavior that processors must follow when accessing the memory. We have to distinguish the access request – read or write – of an instruction from its actual completion.

The consistency model is critical: it imposes constraints on the technological and operational choices, for parameters such as the cache coherence protocol, the size of memory controller buffers, etc.

The following are a few consistency models:

– atomic consistency;

– sequential consistency;

– causal consistency;

– weak consistency.

12.5.2. *Notations*

The following notations specify the nature of the operations in memory and their order:

– nature of the operations:

- $R_p(m)v$: the content of the memory word m is read by the processor P, and the value v is obtained;

- $W_p(m)v$: the value v is written in the memory word m by P;

– order of operations:

- an operation OP_1 precedes an operation OP_2 in the program order: $OP_1 \prec OP_2$;

- an operation OP_1 precedes an operation OP_2 in the order of visibility (of the results): $OP_1 \Rightarrow OP_2$;

– an operation OP_1 precedes an operation OP_2 (denoted by $OP_1 \prec OP_2$) if one of the following conditions is met:

- type (a) precedence: OP_1 and OP_2 are on the same processor and OP_1 is executed before OP_2;

- type (b) precedence: OP_2 is a read operation performed by another processor at the address where OP_1 performed a write operation (and there was no write operation on x between OP_1 and OP_2);

- $OP_1 \prec OP_2$ if there is an $OP3$ such that $OP_1 \prec OP3$ and $OP3 \prec OP_2$;

– OP_1 and OP_2 are parallel (or concurrent) if there is no precedence relationship between them.

When we define the order of events, we are assuming that we know their beginning and end. However, while we know when a memory access request is sent, it is more difficult to define when it ends. The concept of the termination of an operation will specify what constitutes the end of an event:

– the operation consisting of writing the value i in cell x by processor P_B, that is $W_B(x)i$, is said to have terminated for processor P_A if $R_A(x)i$, or if P_A reads the value deposited by a write operation following $W_B(x)i$;

– the operation consisting of reading the value i in cell x by processor P_B, that is $R_B(x)i$, is said to be terminated if that value can no longer be altered by any write operation on x.

A memory access is said to be terminated if it is terminated for *all* of the processors.

12.5.3. *Atomic consistency*

This is the most restrictive model. The behavior it imposes (with respect to memory access) is similar to what occurs in a single-processor architecture:

– write instructions are described as *atomic*: the requested update operation is immediately performed;

– reading cell x returns the value that was stored in this cell by the last write *request* (the most recent, which implies the presence of a global synchronization).

An architecture in which several processors without a cache (or equipped with a write-through cache) are connected by a bus to a single memory meets all the criteria outlined above.

EXAMPLE 12.5.– The following two scenarios (we assume that x is initialized as zero) are correct:

P_A		$W_A(x)1$			
P_B				$R_B(x)1$	$R_B(x)1$

P_A		$W_A(x)1$			
P_B	$R_B(x)0$			$R_B(x)1$	

EXAMPLE 12.6.– The following scenario (we assume that x is initialized as zero) is incorrect:

P_A		$W_A(x)1$			
P_B				$R_B(x)0$	$R_B(x)1$

This last scenario is incorrect because the updates performed by a processor must be immediately propagated and be visible to the others.

This is also called *atomic consistency*: the modification made by a write operation is immediately seen by all the processors and a read request returns the last value that was written.

12.5.4. *Sequential consistency*

Sequential consistency imposes fewer constraints on multiprocessor architectures. Leslie Lamport [LAM 79] defined it as follows: "a multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program".

If P$_i$ executes I_1, I_2, I_3 and P$_j$ executes J_1, J_2, J_3, the sequences $\{I_1$, J_1, I_2, I_3, J_2, $J_3\}$ and $\{I_1$, I_2, J_1, J_2, I_3, $J_3\}$, for example, are correct. But if a processor observes the first sequence, all of the other processors must observe it as well.

REMARKS 12.4.–

– At the end of the execution, there will be several possible values for the shared variables.

– The execution of instructions meets the criteria for sequential consistency if: $OP_1 \prec OP_2$ implies $OP_1 \Rightarrow OP_2$.

EXAMPLE 12.7.– The following scenario, which is incorrect in atomic consistency, is correct in sequential consistency. The variable x is initialized as zero.

t	0	1	2	3	4
P_A		$W_A(x)1$			
P_B				$R_B(x)0$	$R_B(x)1$

If the write instruction is not atomic, it may be that the write operation performed by P_A is not immediately visible to P_B, which is illustrated by the example.

EXAMPLE 12.8.– Another scenario that is correct in sequential consistency:

t	0	1	2	3	4	5
P_A		$W_A(x)2$				
P_B	$W_B(x)1$					
P_C			$R_C(x)2$		$R_C(x)1$	
P_D				$R_D(x)2$		$R_D(x)1$

Processors C and D both see the content of x change from 2 to 1, even though the write instructions are not in that order.

The sequential consistency model is comparable to the transmission conditions on a network with the following assumptions: messages sent from two sites to the same third site are not necessarily received in the order they were sent, and two messages sent by the same site never become "out of sequence".

12.5.5. *Causal consistency*

One more constraint is relaxed: only instructions that are causally dependent, by which we mean instructions that share a precedence relationship, must be seen in the same order by all of the processors.

In all of the examples that follow, x and y are initialized as zero.

EXAMPLE 12.9.– The following scenario does not observe the rule for sequential consistency, but is correct in causal consistency. This is because the events $W_A(i)1$ and $W_B(i)2$ are not causally dependent (they are parallel).

t	0	1	2	3	4	5
P_A	$W_A(x)1$					
P_B		$W_B(x)2$				
P_C			$R_C(x)2$		$R_C(x)1$	
P_D				$R_D(x)1$		$R_D(x)2$

C and D do not see the updates of x in the same order.

EXAMPLE 12.10.– The following scenario is also correct in causal consistency:

t	0	1	2	3
P_A	$W_A(x)1$			$W_A(x)3$
P_B		$R_B(x)1$	$W_B(x)2$	
P_C				
P_D				

t	4	5	6	7	8	9
P_A						
P_B						
P_C	$R_C(x)1$		$R_C(x)3$		$R_C(x)2$	
P_D		$R_D(x)1$		$R_D(x)2$		$R_D(x)3$

There is a type (b) relationship between $W_A(x)1$ and $R_B(x)1$, then a type (a) relationship between $R_B(x)1$ and $W_B(x)2$. The different processors P_i must therefore read $R_i(x)1$ first, then $R_i(x)2$, which is, in fact, the case. There is also a type (a) precedence between $W_A(x)1$ and $W_A(x)3$. The values 1 and 3 are performed in the right order by C and D.

EXAMPLE 12.11.– The following scenario observes the rule for causal consistency, but not for sequential consistency:

t	0	1	2	3	4	5	6	7
P_A	$W_A(x)0$			$W_A(x)1$			$R_A(y)0$	$R_A(y)2$
P_B		$W_B(y)0$	$W_B(y)2$		$R_B(x)0$	$R_B(x)1$		

Sequential consistency forces memory accesses to be performed in the same order as in a single-processor machine. In this case, since P_B reads $x = 0$ then $x = 1$, this means that $W_B(y)2$ has been executed. P_A cannot read $y = 0$ then $y = 2$ and instead must execute $R_A(y)2$, $R_A(y)2$.

12.5.6. *Weak consistency*

Handling the propagation of modifications is left to the programmer, who must rely on instructions [DUB 98] specific to controlling this propagation.

The constraints imposed by weak consistency are the following:

– sequential consistency is only ensured for accesses to synchronization variables;

– no access to a synchronization variable can be done by a processor before all of the previous memory accesses, for all of the processors, are terminated;

– operations on a synchronization variable must be terminated by a processor before it can access memory again.

We use S to denote access to a synchronization variable.

EXAMPLE 12.12.– Consider the following sequence:

P_A	$W_A(x)a$	S			
P_B				S	$R_B(x)a$

The read operation must return the value found in the last write request, which is a. This is because P_A has to wait for all of the accesses to shared variables to be terminated before accessing S. P_B does the same.

In the case where:

P_A	$W_A(x)a$	S			
P_B				$R_B(x)?$	S

the value read in x is not necessarily a, and may instead be a previous value.

12.6. Exercises

Exercise 12.1 (MSI protocol) (hints on page 341)

Consider the system from example 12.2, where write operations are performed in the *write-back* mode. At first, all of the lines are marked "invalid". The value 10 is stored in memory at the address X. The two processors execute two sequences of instructions described in the following table.

n	P_A	P_B	Comment
1	`mov r1,[X]`	...	P_A: `r1:=[X]`
2	...	`mov r4,[X]`	P_B: `r4:=[X]`
3	`mov [X],#0`	...	P_A: `[X]:=0`
4	...	`mov r5,[X]`	P_B: `r5:=[X]`
5	...	`mov [X],#20`	P_B: `[X]:=20`

Describe, using a diagram similar to the diagram in Figure 12.17, the sequence of operations that ensure cache coherence according to the MSI protocol. Comment on each phase.

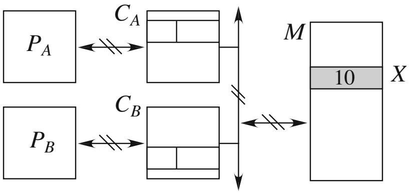


Figure 12.17. *Strongly coupled architecture used in the exercises*

Exercise 12.2 (MSI protocol, update sequences) (hints on page 342) Consider three machines equipped with direct-access caches consisting of sixteen 8-word lines.

These machines execute "read-write" sequences in the order described in the table below. n is the time when instructions are executed. `X` contains A0C0$_{16}$. At first, the line is in state $\mathbb{I}$.

n	P_1	P_2	P_3
1	`ld r1,[X]`	...	...
2	`add r1,1`	...	...
3	`st r1,[X]`	...	...
4	...	`ld r2,[X+2]`	...
5	...	`and r2,0FH`	...
6	...	`st r2,[X+2]`	...
7	...	...	`ld r3,[X+6]`
8	...	...	`sub r3,r1,r5`
9	...	...	`st r3,[X+6]`

Describe the changes in state of the line involved in the three caches.

Exercise 12.3 (MESI protocol) (hints on page 342)

Consider the system shown in Figure 12.17 in the *write-back* mode. At first, all of the lines are “invalid”. The value 10 is stored at the address X in memory.

n	P_A	P_B	Comment
1	`mov r1,[X]`		P_A: `r1:=[X]`
2		`mov r4,[X]`	P_B: `r4:=[X]`
3	`mov [X],#0`		P_A: `[X]:=0`
4		`mov r5,[X]`	P_B: `r5:=[X]`
5		`mov [X],#20`	P_B: `[X]:=20`

The cache coherence protocol is MESI. Give the series of states of the two lines that correspond to `X` in caches C_A and C_B and the memory content.

Exercise 12.4 (Berkeley protocol) (hints on page 343)

The Berkeley protocol [KAT 85, ARC 86], the successor to the Synapse protocol [FRA 84], relies on the concept of “ownership”, as seen in the case of the MOESI protocol. This protocol has the following states:

– *Modified* or *dirty* ($\mathbb{M}$): the associated line is valid, and is only found in this cache; the cache is its owner. The data have not been updated in memory.

– *Potentially shared* or *shared-dirty* ($\mathbb{P}$): the line has been modified and may be shared. The cache is its owner.

– *Shared* or *Valid* ($\mathbb{S}$): the line is *valid*, has not been modified and may be shared.

– *Invalid* ($\mathbb{I}$): the information in the line is not reliable.

REMARKS 12.5.–

– It is always the owner – the memory or the cache – that "sends" the line. If no cache has this line in the $\mathbb{M}$ or $\mathbb{P}$ state, a memory access is necessary.

– When a *Read Miss* LRU occurs, if the line ℓ_o that is chosen is in state $\mathbb{M}$ or $\mathbb{P}$, it is copied into memory. If another cache has the new line ℓ_n in the $\mathbb{M}$ or $\mathbb{P}$ state, that cache provides it, otherwise the memory does. The line ℓ_n switches to $\mathbb{S}$.

– In the case of a *Write Miss* LRU, as with a *Read Miss* LRU, the line is provided by the owner. The line that the owner has is then invalidated, along with any copies that might exist.

Draw the graphs illustrating the Berkeley protocol, and comment on the transitions between states. You may consult the references cited above, in particular [ARC 86].

Exercise 12.5 (Firefly protocol) (hints on page 343)

The Firefly protocol was designed by DEC$^{\text{TM}}$ for its multiprocessor workstation Firefly [THA 88]. It constitutes a subset of the MESI protocol in which there is no invalidation. The three states we previously referred to as $\mathbb{M}$, $\mathbb{E}$ and $\mathbb{S}$ are in this case known as "Dirty" ($\mathbb{D}$), "Valid-Exclusive" ($\mathbb{V}$) and "Shared" ($\mathbb{S}$), respectively. The $\mathbb{S}$ and $\mathbb{V}$ states are such that coherence is preserved.

The caches of the processors P_i share a physical line called the *SharedLine*. It changes state during snooping to indicate that P_i has the line referenced in P_0. Cache to cache communication is assumed to be possible.

Draw the state diagram for this protocol. Note that a *Write Miss* can be handled like a *Read Miss* followed by a *Write Hit*.

Chapter 13

Superscalar Architectures

In Chapter 10, we saw how to parallelize the execution of instructions using a pipeline architecture. This parallelism leads to hazards. Handling these hazards requires not only the use of optimized compilers, but also the ability to detect them, as well as the implementation of hardware solutions to minimize their effects.

An additional level of parallelization can improve performance further, however it increases the degree of complexity, imposes new constraints and requires new solutions. We present two solutions to this problem: *superscalar* architectures and *multiple instruction* architectures.

13.1. Superscalar architecture principles

A superscalar architecture is built with several pipeline-type units, which are fed with instructions as their inputs. Figure 13.1 illustrates how this is organized. It shows the execution pipelines of a PowerPC$^{\mathrm{TM}}$ 603. This simplified representation is useful to get a good idea of how the architecture for this type of processor operates.

The choice of a "superscalar" architecture is dictated by two observations. The first is that instructions vary in their duration, sometimes dramatically (on an Intel® Pentium-4, a floating-point addition or multiplication can take one to three clock cycles, whereas a division takes 39 cycles). The second is that these instructions can often be executed in parallel.

As a consequence, the order in which results are produced with this type of architecture is not necessarily the same as the order of the instructions in the program. It is however an absolute requirement that the order of these write operations be respected in a given memory word or register.

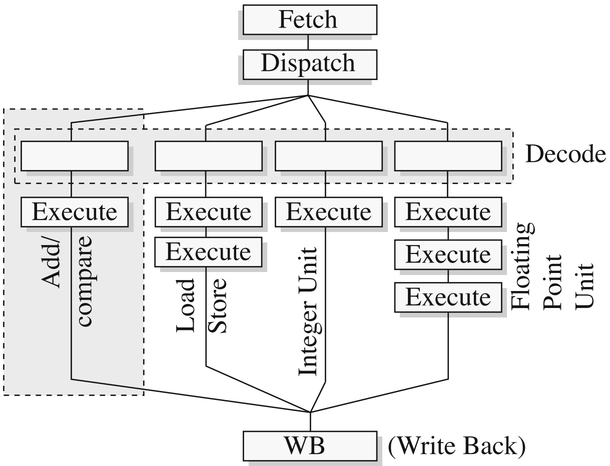


Figure 13.1. *An example of a superscalar architecture, the PPC-603*$^{\mathrm{TM}}$

The use of several pipelines with different or duplicate features increases the degree of parallelism and improves performance, but it inevitably comes at a cost, in the form of higher design complexity. An immediate consequence involves the "Decoding" stage. This stage is fed only after proceeding to a partial decoding phase ("dispatch" in Figure 13.1). The decoding is then completed in each pipeline.

13.1.1. *Hazards*

There are two types of resource that lead to hazards: registers and memory. In superscalar architectures, access to a given piece of information by two instructions is a potential source of error. Figure 13.2 lists all possible cases. If instruction I_i precedes instruction I_j ($I_i \prec I_j$), there can be, for example, a RAW-type hazard if I_i uses the register R_d^i as its destination, and instruction I_j uses the same register as its source ($R_s^j = R_d^i$). In the figure M_s^i and M_d^i refer to addresses in memory corresponding to a source and a destination, respectively.

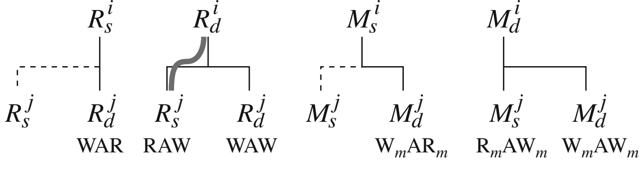


Figure 13.2. *List of hazards that can occur in a superscalar architecture. Instruction I_i precedes I_j. The bold line indicates the possibility of a RAW hazard if $R_s^j = R_d^i$*

A distinction is made between hazards that involve registers and those that involve the memory. Read and write operations are denoted by (R, W) for registers and by (R_m, W_m) for memory words. This is because memory reads and writes occur in the *last execution cycle* of the corresponding pipeline, and not in the W cycle.

The following two examples illustrate the case of WAR and WAW cycles.

EXAMPLE 13.1. (A WAR hazard)– Consider the architecture in Figure 13.3.

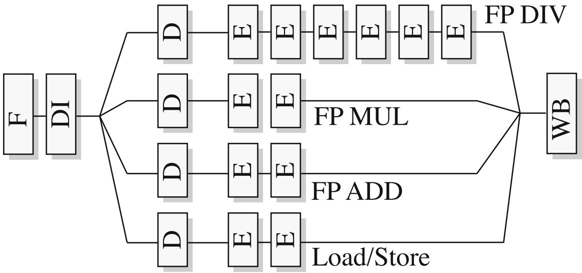


Figure 13.3. *An example of a superscalar machine*

and this excerpt from a program:

```
I1: divf  f5,f4,f1  ; f5 := f4 / f1
I2: mulf  f1,f4,f5  ; f1 := f4 * f5
I3: addf  f4,f2,f2  ; f4 := f2 + f2
```

The RAW hazard between `mulf` and `divf` is resolved by introducing penalty cycles, as was the case with a single pipeline.

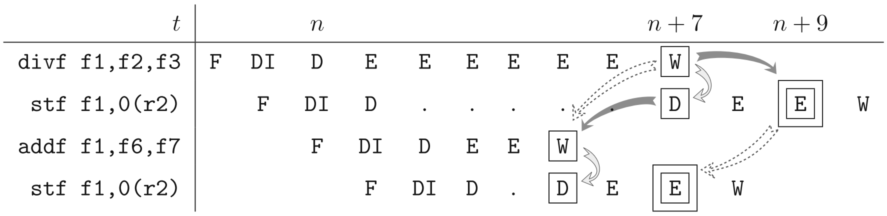
t			n				$n+5$					
`divf f5,f4,f1`	F	DI	D	E	E	E	E	E	E	$\boxed{\mathtt{W}}$			
`mulf f1,f4,f5`		F	DI	D	D	D	D	D	D	$\boxed{\mathtt{D}}$	E	E	W
`addf f4,f2,f2`			F	DI	D	E	E	$\boxed{\mathtt{W}}$					

But there is also an unresolved WAR-type hazard, since `addf` modifies the content $\texttt{f4}_n$ of `f4`, which is also used by `mulf`, at time n. But this last instruction must wait for `f5` to be calculated by `divf`. As a result, `addf` will be executed before `mulf` and will modify `f4` at time $n + 5$. `mulf` will therefore be using $\texttt{f4}_{n+5}$ instead of $\texttt{f4}_n$.

EXAMPLE 13.2. (A WAW hazard)– Consider the following excerpt from a program on the same processor as in example 13.1:

```
I1: divf f1,f2,f3   ; f1 := f2 / f3
I2: stf  f1,0(r2)   ; store f1 --> mem(0(r2))
I3: addf f1,f6,f7   ; f1 := f6 + f7
I4: stf  f1,0(r2)   ; store f1 --> mem(0(r2))
```

The execution diagram, where RAW hazards are resolved, is as follows:



t			n							$n+7$		$n+9$	
divf f1,f2,f3	F	DI	D	E	E	E	E	E	E	$\boxed{\text{W}}$			
stf f1,0(r2)		F	DI	D	.	.	.	.	.	$\boxed{\text{D}}$	E	$\boxed{\boxed{\text{E}}}$	W
addf f1,f6,f7			F	DI	D	E	E	$\boxed{\text{W}}$					
stf f1,0(r2)				F	DI	D	.	$\boxed{\text{D}}$	E	$\boxed{\boxed{\text{E}}}$	W		

The sum is completed before the division. The results are not produced in the right order. The second `stf` writes the values of `f1` into memory, at time $n + 7$, before the first `stf`, which performs its write operation at time $n + 9$. This violates the principle according to which *write operations in a given memory word must be performed in the order of the program.*

13.2. Seeking solutions

Dependency hazards can be handled in several different ways. As with single pipeline architectures, the task can either be left to the code generator or the hardware. It is the latter case we are interested in.

13.2.1. *Principles*

Consider again example 13.1.:

```
I1: divf  f5,f4,f1  ; f5 := f4 / f1
I2: mulf  f1,f4,f5  ; f1 := f4 * f5
I3: addf  f4,f2,f2  ; f4 := f2 + f2
```

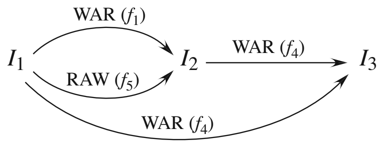


Figure 13.4. *Expressing relations of precedence between instructions*

The *dependency graph* (Figure 13.4) is a representation of the constraints we are faced with.

However, the mechanisms involved can only be understood if they are expressed with a little more finesse. For example, knowing that the write operation takes place at the end of the instruction, we can point out that I_3 could start before I_2 is completed, an observation that is not represented in Figure 13.4.

To improve our model, dependencies must be represented with a higher level of detail. The phases of instruction I_i are denoted by F_i, D_i, E_i^k and W_i, while precedence in time is denoted by $\prec$ (precedes) or $\preceq$ (precedes or in parallel). An XXX conflict on a register f_k will be denoted by XXX(f_k). To solve the conflicts in example 13.1, we can express the conditions of precedence as follows:

– $D_1 \prec W_2$: WAR(f_1) conflict between `divf` and `mulf`;

– $W_1 \preceq D_2$: RAW(f_5) conflict between `divf` and `mulf`;

– $D_1 \prec W_3$: WAR(f_4) conflict between `divf` and `addf`;

– $D_2 \prec W_3$: WAR(f_4) conflict between `mulf` and `addf`.

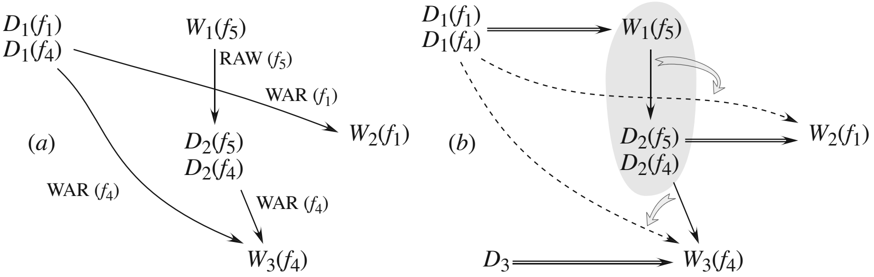


Figure 13.5. *Expressing relations of precedence between phases*

Figure 13.5(a) is a representation of these dependencies. In Figure 13.5(b):

– “double” arrows indicate hardware precedence in the execution of an instruction;

– dashed arrows correspond to hazards resolved by satisfying other constraints. Resolving the RAW(f_5) hazard implies staying within the constraints set by WAR(f_1).

Two methods are applied to the previous examples. The first consists of introducing wait states to comply with the constraints of precedence. The second, which is more effective, relies on the use of *register renaming* (section 13.3.5).

13.2.1.1. *Introducing wait states*

To resolve the WAR hazard between I_2 and I_3, instruction I_3 can proceed, but must wait for D_2 to be completed before executing W_3:

t			n					$n+5$					
`divf f5,f4,f1`	F	DI	D	E	E	E	E	E	E	$\boxed{\text{W}}$			
`mulf f1,f4,f5`		F	DI	D	.	.	.	.	.	$\boxed{\text{D}}$	E	E	W
`addf f4,f2,f2`			F	DI	D	E	E	.	.	.	W		

The resolution of a WAW hazard would be expressed as $E_1^n \prec E_2^n$, where E_1^n and E_2^n are the last execution cycles of I_1 and I_2, respectively.

13.2.1.2. *Register renaming*

While it is not possible to eliminate the wait cycles due to RAWs, we can limit the effects from WARs and WAWs. This is done by performing the write operations in copies of the registers, known as *avatars*, rather than in the registers themselves. The avatar for f_i is denoted by $f_i^{'}$.

With this new technique, the execution diagram becomes:

t			n					$n+5$					
`divf f'5,f4,f1`	F	DI	D	E	E	E	E	E	E	$\boxed{\text{W}}$			
`mulf f'1,f4,f5`		F	DI	D	.	.	.	.	.	$\boxed{\text{D}}$	E	E	W
`addf f'4,f2,f2`			F	DI	D	E	E	W					

Note that the write operation is performed in $f_4^{'}$ and not in f_4, that the WARs have disappeared, and that the `addf` instruction can be completed earlier.

Note also that once the addition is decoded, it can be executed independently of the two other instructions.

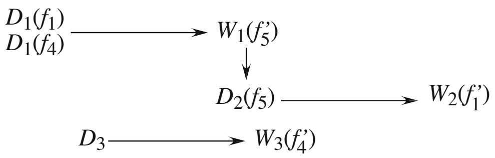


Figure 13.6. *Expressing precedence with avatars*

13.2.2. *Example*

Consider again the program excerpt from example 13.2:

```
I1: divf f1,f2,f3   ; f1 := f2 / f3
I2: stf  f1,0(r2)   ; store f1 --> mem(0(r2))
I3: addf f1,f6,f7   ; f1 := f6 + f7
I4: stf  f1,0(r2)   ; store f1 --> mem(0(r2))
```

13.2.2.1. *Introducing wait states*

The potential hazards are the following:

- $W_1 \preceq D_2$: RAW(`f1`) hazard between `divf` and the first `stf`;
- $W_1 \preceq D_4$: RAW(`f1`) hazard between `divf` and the second `stf`;
- $D_2 \prec W_3$: WAR(`f1`) hazard between `addf` and the first `stf`;
- $W_3 \preceq D_4$: RAW(`f1`) hazard between `addf` and the second `stf`;
- $W_1 \prec W_3$: WAW(`f1`) hazard between `divf` and `addf`;
- $E_2^2 \preceq E_4^2$: hazard between the two `stf` writing *at the same address* `0(r2)`.

Compliance with the constraints is represented in Figure 13.7.

We get the following execution diagram. The `E` cycle with a double frame around it in the diagram corresponds to what we called W_m. Phase W_3 of `addf` is delayed so that it takes place after D_2.

13.2.2.2. *Register renaming*

Note that there are two register copies, the avatars f_1' and f_1''. This allows `divf` and `addf` to be executed in parallel. The addition is completed before the division. Note that the constraint due to concurrent memory accesses imposes a large penalty. The execution diagram is:

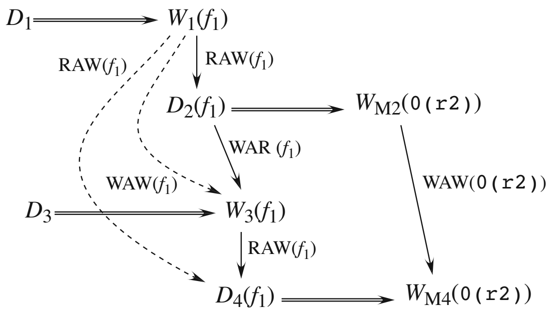


Figure 13.7. *Expressing relations of precedence between phases*

t			n							$n+7$		$n+9$	
`divf f1,f2,f3`	F	DI	D	E	E	E	E	E	E	$\boxed{\texttt{W}}$			
`stf f1,0(r2)`		F	DI	D	.	.	.	.	.	$\boxed{\texttt{D}}$	E	$\boxed{\boxed{\texttt{E}}}$	
`addf f1,f6,f7`			F	DI	D	E	E	.	.	.	$\boxed{\texttt{W}}$		
`stf f1,0(r2)`				F	DI	D	.	.	.	.	$\boxed{\texttt{D}}$	E	$\boxed{\boxed{\texttt{E}}}$

t			n							$n+7$		$n+9$	
`divf f'1,f2,f3`	F	DI	D	E	E	E	E	E	E	$\boxed{\texttt{W}}$			
`stf f'1,0(r2)`		F	DI	D	.	.	.	.	.	$\boxed{\texttt{D}}$	E	$\boxed{\boxed{\texttt{E}}}$	
`addf f"1,f6,f7`			F	DI	D	E	E	$\boxed{\texttt{W}}$					
`stf f"1,0(r2)`				F	DI	D	.	$\boxed{\texttt{D}}$	E	.	.	.	$\boxed{\boxed{\texttt{E}}}$

t			n							$n+7$		$n+9$
`divf f'1,f2,f3`	F	DI	D	E	E	E	E	E	E	$\boxed{\texttt{W}}$		
`stf f'1,0(r2)`		F	DI	D	.	.	.	.	.	$\boxed{\texttt{D}}$	E	$\boxed{\boxed{\texttt{E}}}$
`addf f"1,f6,f7`			F	DI	D	E	E	$\boxed{\texttt{W}}$				
`stf f"1,0(r3)`				F	DI	D	.	$\boxed{\texttt{D}}$	E	$\boxed{\boxed{\texttt{E}}}$		

If the second `stf` writes in `0(r3)` instead of in `0(r2)`, we get the following diagram:

The pair of instructions `addf-stf` is then completed before `divf-stf`.

13.3. Handling the flow of instructions

In the following sections we present two techniques for resolving dependency hazards: *scoreboarding* and the *Tomasulo algorithm*, also known as *register renaming*.

13.3.1. *Principle of scoreboarding*

This approach, introduced with the CDC 6600 [THO 70, THO 63], consists of writing all of the dependencies into a table – a scoreboard. They are then resolved by introducing *stall cycles*, as is done with a single pipeline. We assume that there is no forwarding mechanism in place. The data are always transmitted to the pipelines from the registers. Decoding is divided into two phases (Figure 13.8): DI (*Decode-Issue*) and DR (*Decode-Read*).

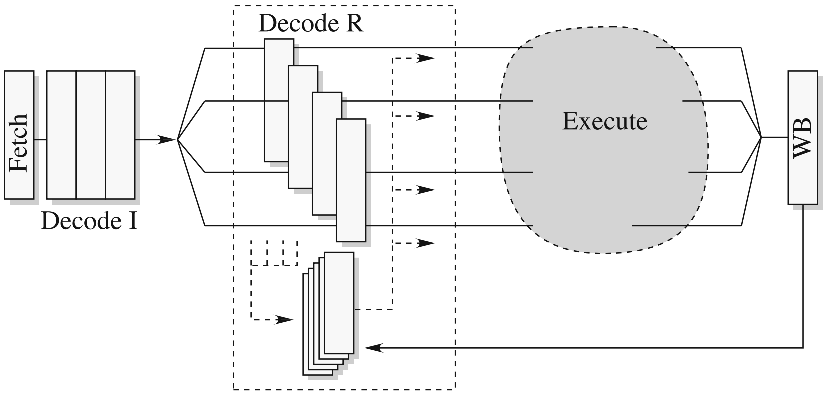


Figure 13.8. *Two decoding phases*

Let $\mathscr{R}$ be the instruction that reads the information (register or memory word) involved in a conflict, and $\mathscr{W}$ is the instruction that modifies it:

– During the DI phase, each instruction is read *in the order of the program*. A search for WAW hazards is performed. If such a hazard is detected, the instruction does not move on to DR. This ensures that the write operations take the dependencies into account.

– During the DR phase of the $\mathscr{R}$ instruction, we check that the source operands are not present in the destination field of one of the $\mathscr{W}$ instructions currently being executed. If a source operand is found, there is a RAW type hazard. The $\mathscr{R}$ instruction is held in DR until the updates have been completed.

– We still have to check that the $\mathscr{W}$ instruction cannot be involved in a WAR-type hazard. If the destination field is the same as the source field of $\mathscr{R}$ instruction, $\mathscr{W}$ proceeds, then *stalls*, before entering its `WB` phase, which lasts until the read operations have been performed on the registers involved (`DR` phase of $\mathscr{R}$).

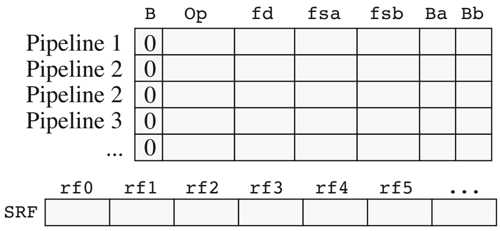
To summarize, if I_1 and I_2 are in conflict, with $I_1 \prec I_2$:

> – During DI: in case of WAW, then I_2 stays in DI waiting for the W of I_1. The fetch operation is put on hold until the WAW has been resolved.
>
> – During DR: in case of RAW, then I_2 stays in DR waiting for the W of I_1.
>
> – In the last phase E^n of `E`: in case of WAR, then I_2 stays in E_2^n as long as I_1 has not completed DR.

Note that a W_mAW_m hazard meets the first condition above since the E_1^n phase (memory write operation) always precedes phase W.

13.3.2. *Scoreboarding implementation*

Hazards are handled using a *scoreboard* (Figure 13.9).



	B	Op	fd	fsa	fsb	Ba	Bb
Pipeline 1	0						
Pipeline 2	0						
Pipeline 2	0						
Pipeline 3	0						
...	0						

	rf0	rf1	rf2	rf3	rf4	rf5	...
SRF							

Figure 13.9. *The scoreboard used for handling the flow of instructions*

The scoreboard contains:

– a status register SRF, each entry of which corresponds to a register and indicates (in Boolean form) if that register is the destination of a current instruction (RAW and WAW detection);

– a table with one or several entries per execution pipeline, with the following elements:

- `B` indicates if the entry is *busy* (Boolean),

- `Op` refers to the code for the operation using the unit,

- $\mathtt{f_d}$ indicates the destination register,

- $\mathtt{f_{sa}}$ and $\mathtt{f_{sb}}$ are the source registers in the same order as in the instruction `Op` $\mathtt{f_d}$,$\mathtt{f_{sa}}$ and $\mathtt{f_{sb}}$,

- `Ba` and `Bb` indicate the availability (Boolean) of $\mathtt{f_{sa}}$ and $\mathtt{f_{sb}}$, respectively. A "1" indicates that the resource is available: the last update is completed.

13.3.3. *Detailed example*

EXAMPLE 13.3.– Consider a processor equipped with four execution pipelines (Figure 13.10). The integer unit (IU) executes integer calculations and memory accesses. ADF, MF and DIVF execute floating-point additions/subtractions, multiplications and divisions, respectively. The scoreboard has only one entry per pipeline.

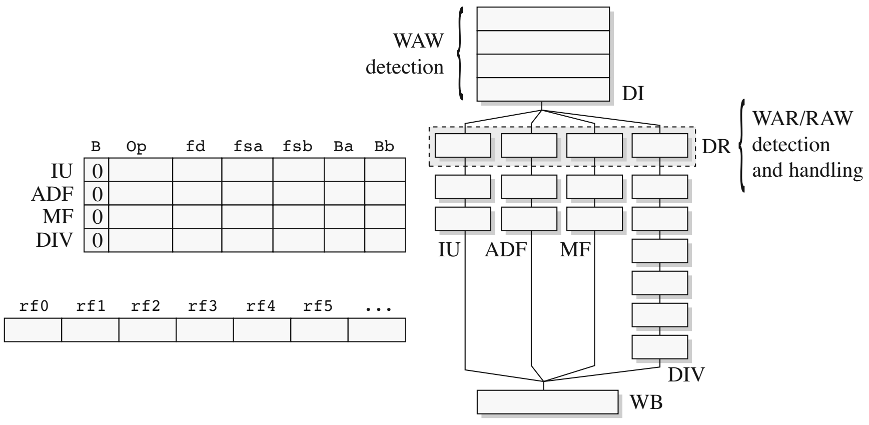


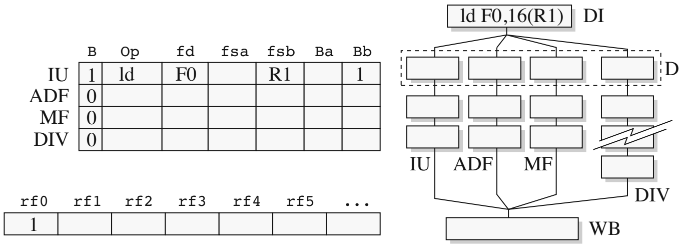
Figure 13.10. *Processor with four execution pipelines*

The following instructions are executed. The comments indicate the different types of hazards that can occur.

```
I1: ld   F0,16(R1)
I2: ld   F1,64(R1)
I3: divf F3,F1,F2   ; RAW(F1) with I2
I4: addf F2,F0,F1   ; WAR(F2) with I3, RAW(F0) with I1,
                    ; RAW(F1) with I2
I5: mulf F5,F3,F0   ; RAW(F0) with I1, RAW(F3) with I3
```

```
I6: st   F5,0(R1)   ; RAW(F5) with I5
I7: subf F5,F4,F1   ; WAW(F5) with I5, RAW(F1) with I2
```

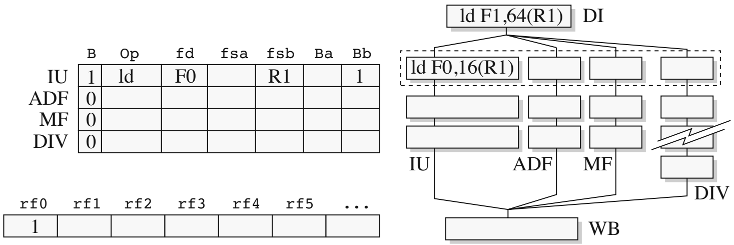
The following figures lay out the details of the program execution cycles.



	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	ld	F0		R1		1
ADF	0						
MF	0						
DIV	0						

rf0	rf1	rf2	rf3	rf4	rf5	...
1						

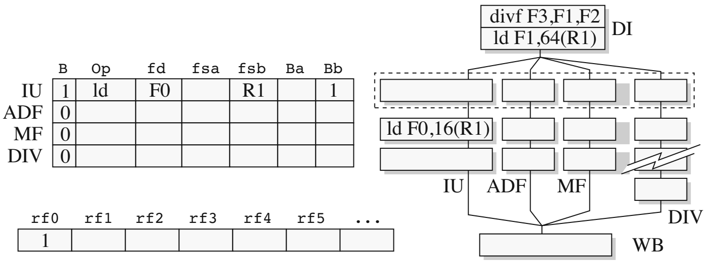
Figure 13.11. *Cycle 1:* `ld F0`*: the IU entry is updated.* `R1` *is tagged "available" (*`Bb` *field) and* `rf0` *"busy".* `ld F0` *can proceed to the DR phase*



	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	ld	F0		R1		1
ADF	0						
MF	0						
DIV	0						

rf0	rf1	rf2	rf3	rf4	rf5	...
1						

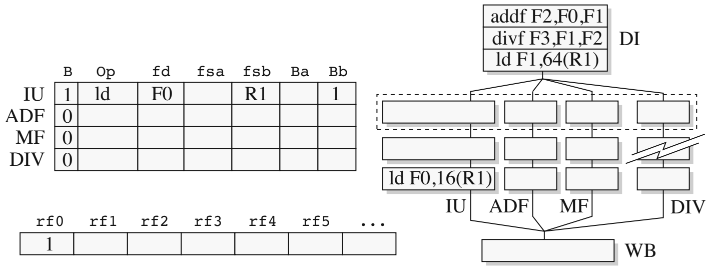
Figure 13.12. *Cycle 2:* `ld F1` *does not proceed to the DI phase because the resource (IU entry) managing the IU pipeline is "busy". One possible improvement would be to add an entry in the reservation table*



	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	ld	F0		R1		1
ADF	0						
MF	0						
DIV	0						

rf0	rf1	rf2	rf3	rf4	rf5	...
1						

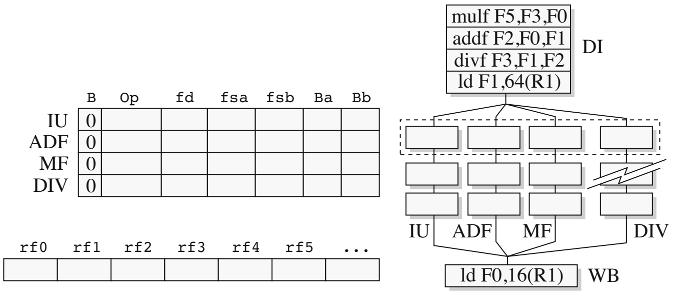
Figure 13.13. *Cycle 3:* `ld F0` *proceeds to the execution phase (address calculation)*



	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	ld	F0		R1		1
ADF	0						
MF	0						
DIV	0						

rf0	rf1	rf2	rf3	rf4	rf5	...
1						

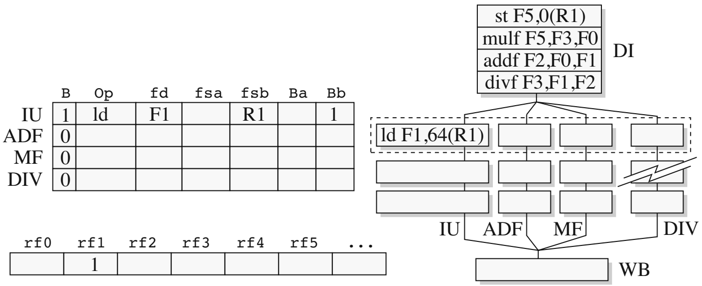
Figure 13.14. *Cycle 4:* `ld F0` *proceeds to the memory access phase*



	B	Op	fd	fsa	fsb	Ba	Bb
IU	0						
ADF	0						
MF	0						
DIV	0						

rf0	rf1	rf2	rf3	rf4	rf5	...

Figure 13.15. *Cycle 5:* `ld F0` *proceeds to the* `F0` *write phase. Once this phase has been executed, register F0 is available and the IU entry is free.* `rf0` *is tagged "free"*



	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	ld	F1		R1		1
ADF	0						
MF	0						
DIV	0						

rf0	rf1	rf2	rf3	rf4	rf5	...
	1					

Figure 13.16. *Cycle 6: the IU entry is updated.* `rf1` *is tagged "busy".* `ld F1` *proceeds to the DR phase*

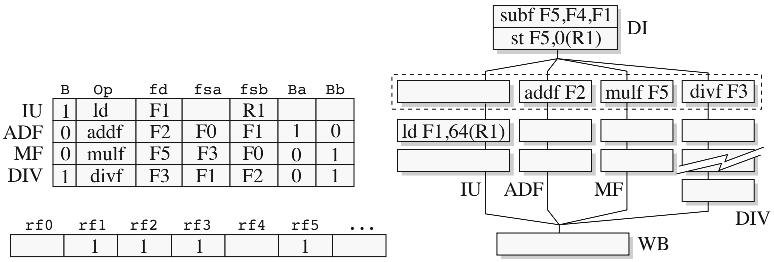
Remarks on Figure 13.17 for cycle 7 are as follows:

– The DIV entry is initialized as `divf`. `Ba` indicates that `F1` is not available for the division (RAW hazard with `ld`). `rf3` is tagged "busy".

– The ADF entry is initialized as `addf`. `rf2` is tagged "busy". There is a second RAW hazard with the `ld F1`.

– The MF entry is initialized as `mulf`. `rf5` is tagged “busy”.

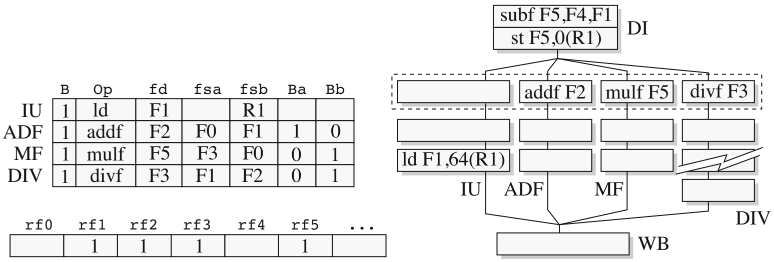
– Note that in this phase, `divf F3,F1,F2` reads the value of `F2`. If there was no dependency involving `F1`, `addf F2` could be executed in parallel with `divf`. In that case, the WAR hazard should be resolved by putting `addf` on hold in the WB phase.



	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	ld	F1		R1		
ADF	0	addf	F2	F0	F1	1	0
MF	0	mulf	F5	F3	F0	0	1
DIV	1	divf	F3	F1	F2	0	1

rf0	rf1	rf2	rf3	rf4	rf5	...
	1	1	1		1	

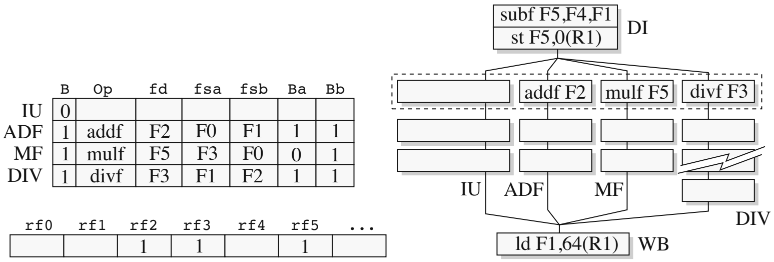
Figure 13.17. *Cycle 7:* `ld F1` *is in the address calculation phase*



	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	ld	F1		R1		
ADF	1	addf	F2	F0	F1	1	0
MF	1	mulf	F5	F3	F0	0	1
DIV	1	divf	F3	F1	F2	0	1

rf0	rf1	rf2	rf3	rf4	rf5	...
	1	1	1		1	

Figure 13.18. *Cycle 8: the memory access is performed*

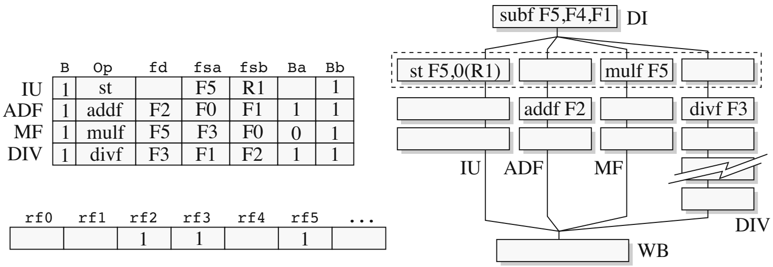


	B	Op	fd	fsa	fsb	Ba	Bb
IU	0						
ADF	1	addf	F2	F0	F1	1	1
MF	1	mulf	F5	F3	F0	0	1
DIV	1	divf	F3	F1	F2	1	1

rf0	rf1	rf2	rf3	rf4	rf5	...
		1	1		1	

Figure 13.19. *Cycle 9:* `ld F1` *writes into register F1. The IU entry becomes free again and F1 is tagged “available” for* `addf` *and* `divf` *(in Bb and Ba, respectively)*

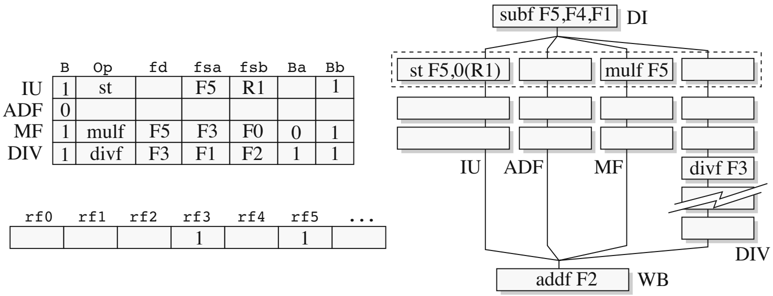
If the machine were capable of modifying the entries of the scoreboard *and* of modifying `F3` in the first half-cycle of this phase, then `mulf` could proceed to the execution phase. In this example, we wait for the next cycle.



	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	st		F5	R1		1
ADF	1	addf	F2	F0	F1	1	1
MF	1	mulf	F5	F3	F0	0	1
DIV	1	divf	F3	F1	F2	1	1

rf0	rf1	rf2	rf3	rf4	rf5	...
		1	1		1	

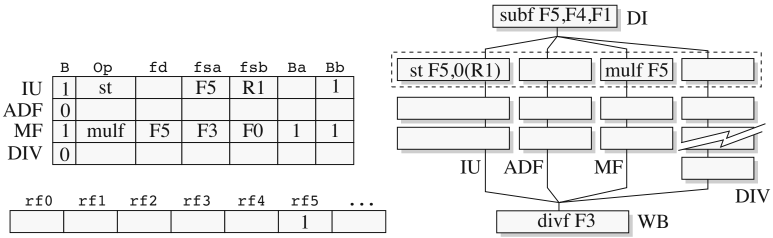
Figure 13.20. *Cycle 10:* `mulf` *is put on hold to wait for F3, just as* `st F5` *is on hold for F5 (RAW). Since F1 has been updated,* `addf` *and* `divf` *proceed to the execution phase.* `subf` *remains in the DI phase, waiting for the ADF entry to become free*



	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	st		F5	R1		1
ADF	0						
MF	1	mulf	F5	F3	F0	0	1
DIV	1	divf	F3	F1	F2	1	1

rf0	rf1	rf2	rf3	rf4	rf5	...
			1		1	

Figure 13.21. *Cycle 12: in cycle 11, the* `addf` *and* `divf` *instructions have proceeded down the pipeline. Then F2 is updated and the ADF entry becomes free*



	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	st		F5	R1		1
ADF	0						
MF	1	mulf	F5	F3	F0	1	1
DIV	0						

rf0	rf1	rf2	rf3	rf4	rf5	...
					1	

Figure 13.22. *Cycle 16: in cycle 13, even though ADF has become free,* `subf` *cannot proceed to the DR phase because it creates a WAW hazard over F5 with* `mulf`. *In cycle* 16 `divf` *reaches the WB phase. The DIV entry becomes free, along with* `rf5`. `Ba` *of* `mulf` *(which corresponds to F3) is tagged "available"*

In cycles 22 and 23, `st` and `subf` end, respectively. The entries become free.

	B	Op	fd	fsa	fsb	Ba	Bb
IU	1	st		F5	R1		1
ADF	1	subf	F5	F4	F1	1	1
MF	0						
DIV	0						

rf0	rf1	rf2	rf3	rf4	rf5	...
					1	

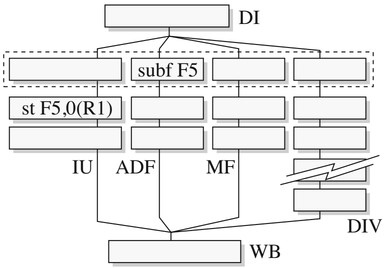


Figure 13.23. *Cycle 20: in cycle 19,* `mulf` *ended by making the entries MF and rf5 free. In the next cycle,* `st` *proceeds to the address calculation phase and* `subf` *proceeds to DR*

13.3.4. *Comments on precedence constraints*

Let us consider the same architecture, with the sequence of instructions:

```
I1: ld   F4,0(R1)  ;
I2: divf F2,F4,F5  ; RAW(F4) with I1
I3: addf F5,F3,F7  ; WAR(F5) with I2
```

The precedence constraints are expressed as $W_1 \preceq D_2 \prec W_3$. If the initial conditions are those in Figure 13.24, than the `ld` and `addf` instructions will reach the WB stage at the same time.

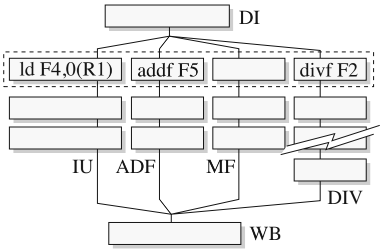


Figure 13.24. *Initial conditions*

There are two possible scenarios (Figure 13.25):

– To stay within the constraints set by precedence, `ld` proceeds to the WB phase and `addf` is put on hold. `ld` updates `F4`, `divf` can proceed, and `addf` becomes free.

– If the precedence constraints are not met, `addf` enters the WB phase but must wait for `divf` to read `F5`. However, `divf` is itself waiting for `ld` to be completed, which is not possible because it cannot proceed to WB. The system is *deadlocked*.

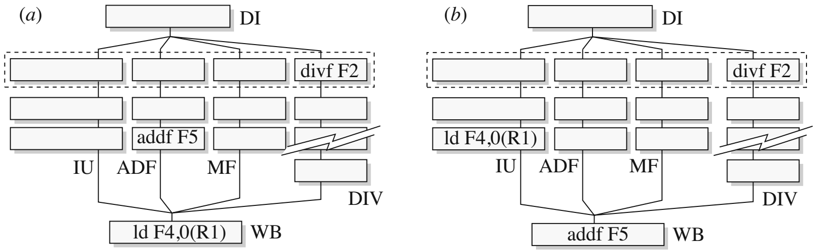


Figure 13.25. *Order of operations (a) when constraints are met and (b) in case of deadlock*

If the WB of `addf` takes place before the WB of `ld`, then the latter must wait for W_3 to occur, producing a loop in the dependency graph (dashed line in Figure 13.26), in other words deadlock.

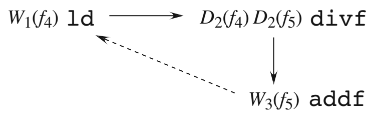


Figure 13.26. *Expressing precedences*

13.3.5. *Principle of the Tomasulo algorithm*

The Tomasulo algorithm achieves dynamic control over the flow of executions of instructions. It was originally used on the IBM 360/91 in 1967 [AND 67]. A verification of the behavior of one of its realizations can be found in [MCM 98].

Hazard handling relies on a set of tables called the *reservation station* (RS) (Figure 13.27).

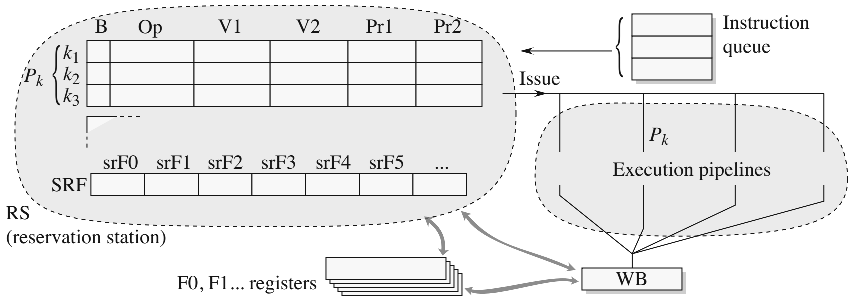


Figure 13.27. *Organization used for the implementation of the Tomasulo algorithm*

RS serves as a stage for handling the issuing of instructions into the different pipelines.

Each pipeline P_A, P_B, etc. is assigned several entries $\{a_i, i = 1, ..., N_A\}$, $\{b_j, j = 1, ..., N_B\}$, etc., in this table (Figure 13.28). These entries are used in the following way: if the instruction I_m handled by P_A depends on I_n handled by P_B for one of its operands k, this dependency is indicated by placing the name of the entry (b_j) associated with I_n in the field Pr$_k$ for the entry a_i associated with I_m (Figure 13.28).

When I_n is completed, the value v$_n$ of the operand k calculated by I_n is stored in the field V$_k$, $k = 1$ or 2, of the entry a_i (Figure 13.28).

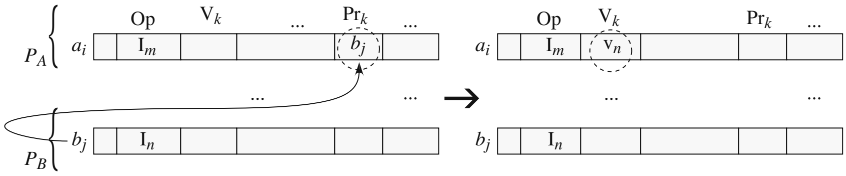


Figure 13.28. *Reservation and assignment of the value to the operand*

13.3.5.1. *Operation*

The handling of the flow of instructions is performed as follows:

1) *Instruction access phase*: the instructions are read in the *same order as in the program* and are stored in an *instruction queue*.

2) *Hazard handling phase* (*issue*): each instruction is assigned an entry a_i if one of them is available:

i) If the instruction modifies a register `Fr`, this a_i is noted in the r entry of a table called the *Status Register File* (SRF).

ii) For instructions other than `load`, information on the availability of the operands is stored in the a_i (Figure 13.28):

- If a source operand is available, its value is copied in a field V$_k$ of a_i: this is known as creating an *avatar* of the register in question (*register renaming*).

- If this operand is not available, we write in a field r$_k$ of a_i the entry b_j associated with the instruction that is supposed to produce the value of the operand. This b_j can be found in the r entry of the SRF.

iii) For memory read and write instructions, the *value* of the address in memory is written into an “Add” field of this entry. For write instructions, we write in a field `Pr` the entry associated with the instruction that is supposed to produce the value of the operand (Figure 13.29).

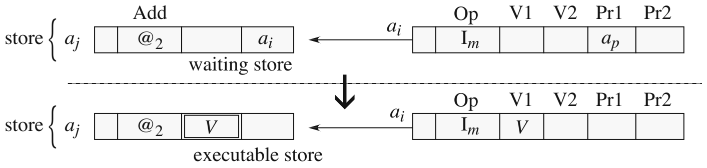


Figure 13.29. *Reservation and assignment of the value of the operand in the case of a write instruction.* $@_2$ *is an address. When* I_m *has calculated* V*, it transmits it to the entry* a_j

- The instruction proceeds to the execution phase when the *values* of the resources are available (avatars or memory addresses).

- When the register update is completed, the calculated value is inserted in the corresponding fields V$_k$ and the r entry in the SRF becomes free.

To summarize:

– WAR hazard: resolved by creating an avatar that memorizes the value that must be used by the read instruction.

– WAW hazard: resolved using the SRF, which guarantees the order in which the write instructions are performed.

– RAW hazard: instructions on hold in the buffer (detection using the SRF).

EXAMPLE 13.4.– [Examples of hazards] Consider the following program with the architecture shown in Figure 13.30:

```
I1: divf f2,f2,f1
I2: stf  f2,r0(r3) ; RAW with I1
I3: mulf f2,f5,f6  ; WAR with I2, WAW with I1
```

As a reminder, here is how the instructions are handled in the case of scoreboarding:

```
divf f2,f2,f1 | F  I  R  E  E  E  E  E  E  W
stf f2,r0(r3) |    F  I  R  .  .  .  .  .  R  E  W
mulf f2,f5,f6 |       F  I  .  .  .  .  I  R  E  E  W
```

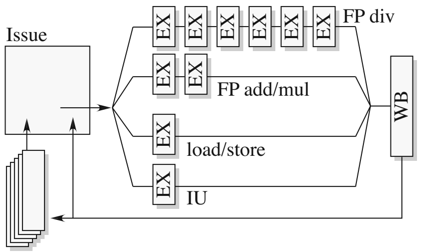


Figure 13.30. *Example of a superscalar architecture*

If we apply the Tomasulo algorithm, we get:

```
divf f2,f2,f1 | F  I  E  E  E  E  E  E  W
stf f2,r0(r3) |    F  I  .  .  .  .  .  I  E  W
mulf f2,f5,f6 |       F  I  .  .  .  .  I  E  E  W
```

We have gained a cycle, which can make a significant difference in the case of a loop.

13.3.6. *Detailed example*

EXAMPLE 13.5.– [Implementation of the Tomasulo algorithm] Consider the architecture described by Figure 13.31.

We have represented the *common data bus* (CDB) that transports all of the data and information related to pipeline handling.

This machine consists of three dedicated pipelines: LS, AS and MD. The first is associated with read and write operations, the second with floating-point additions and subtractions, and the third with floating-point multiplications and divisions. We do not show the pipeline used for handling integers.

The stage inside the dashed line is called the *reservation station* (RS) and consists of many different elements:

– Two sets of registers: the first, associated with the AS pipeline (addition–subtraction), is equipped with three entries as1, as2, as3; the second, associated with the MD pipeline (multiplication–division), is equipped with two entries md1 and md2. These entries are made up of several fields:

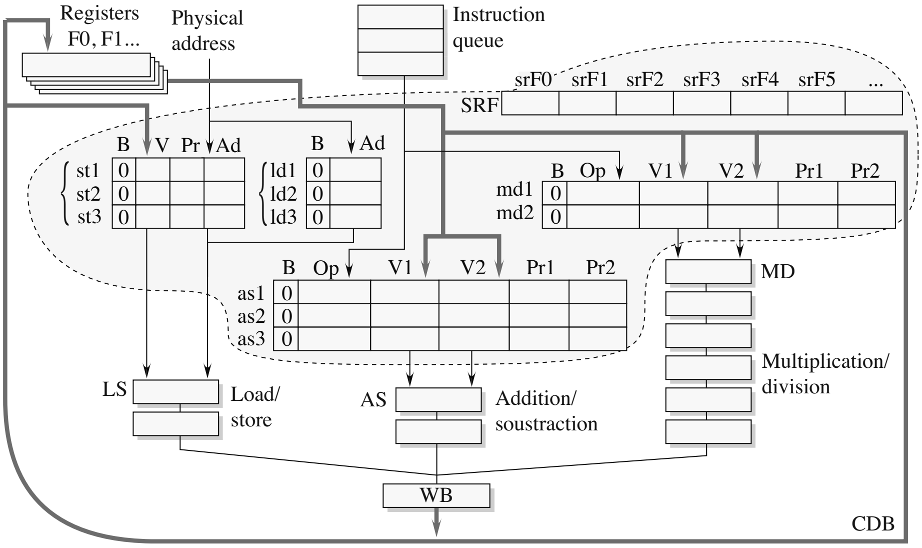


Figure 13.31. *Example of a superscalar architecture that implements the Tomasulo algorithm. The dashed line shows the extent of the reservation station*

- B: a *busy* indicator. When $B = 1$, the entry is busy with an instruction currently being executed.

- Op: the code for the instruction currently on hold for the execution phase.

- V1 and V2 are avatars of the source operand registers; they receive the value of the operand when it becomes available. V1 and V2 correspond to the two source operands `src1` and `src2`, of the instruction `op dest,src1 and src2`.

- Pr1 and Pr2 contain the names of the entries (as1, md2, etc.) of the reservation station associated with the pipeline that is supposed to produce the values of the registers `src1` and/or `src2`.

entry:

B	Op	V1	V2	Pr1	Pr2

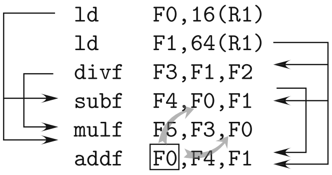
– A set LS of registers associated with the load/store pipeline, consisting of two subsets: the first consists of `ld1`, `ld2` and `ld3`, and the second of `st1`, `st2`, etc. The B (Busy) field has the same purpose as for the fields `as1`, etc.

– A set SRF of registers, srF$_k$, which receives the name of the entry associated with the last instruction that will modify the register $\mathtt{F}_k$.

The following sequence of instructions is executed. Shown in the comments are the different types of potential hazards.

```
I1: ld   F0,16(R1)
I2: ld   F1,64(R1)
I3: divf F3,F1,F2  ; RAW with I2
I4: subf F4,F0,F1  ; RAW with I1 and I2
I5: mulf F5,F3,F0  ; RAW with I1 and I3
I6: addf F0,F4,F1  ; RAW with I2 and I4, WAR with I5 and I4,
                   ; WAW with I1
```

The dependency tree can be represented in Figure 13.32.



```
ld    F0,16(R1)
ld    F1,64(R1)
divf  F3,F1,F2
subf  F4,F0,F1
mulf  F5,F3,F0
addf  F0,F4,F1
```

Figure 13.32. *Dependency tree*

13.3.6.1. *Scenario*

– `load F0` and `load F1`: the entry `ld1` is assigned to the `load F0`. It is used for writing the operand `16(R1)`. `srF0` receives `ld1` to indicate that this is the load that produces the value of `F0`. The same is done for `ld2` (Figures 13.33 and 13.34).

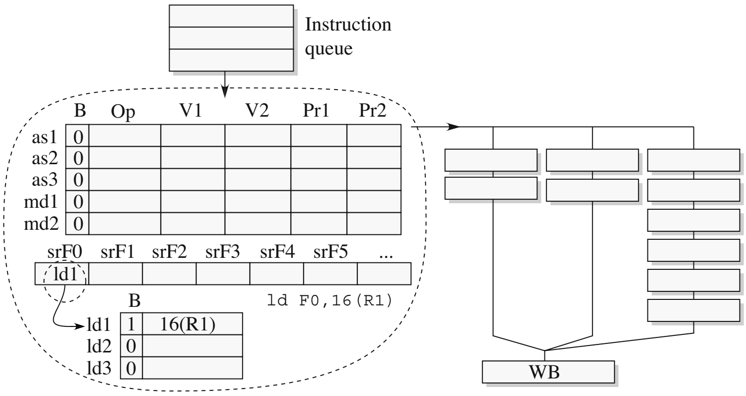
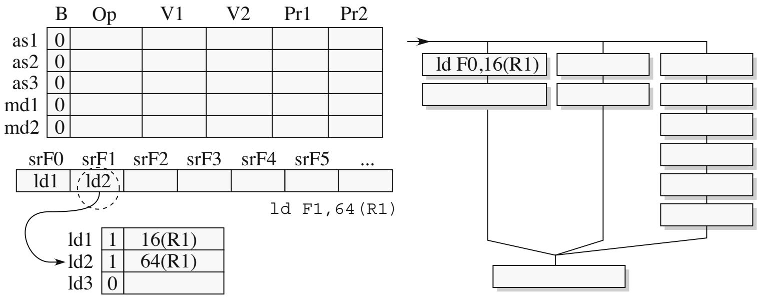


Figure 13.33. *Cycle 1:* `load F0,16(R1)` *in phase R*

– `divf`: the entry `md1` is assigned to the `divf` instruction (Figure 13.35).

The `divf` will have to wait for `load F1`, the status of which is given by the entry `ld2`, to finish executing. `ld2` is therefore moved to the `Pr1` field. The value of `F2` is

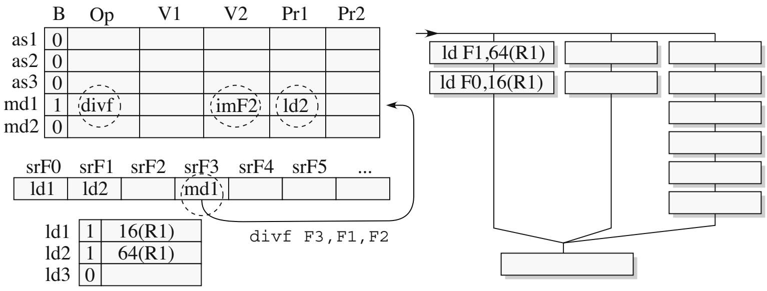


	B	Op	V1	V2	Pr1	Pr2
as1	0					
as2	0					
as3	0					
md1	0					
md2	0					

srF0	srF1	srF2	srF3	srF4	srF5	...
ld1	ld2					

ld1	1	16(R1)
ld2	1	64(R1)
ld3	0	

Figure 13.34. *Cycle 2:* `load 64(R1)` *in phase R*



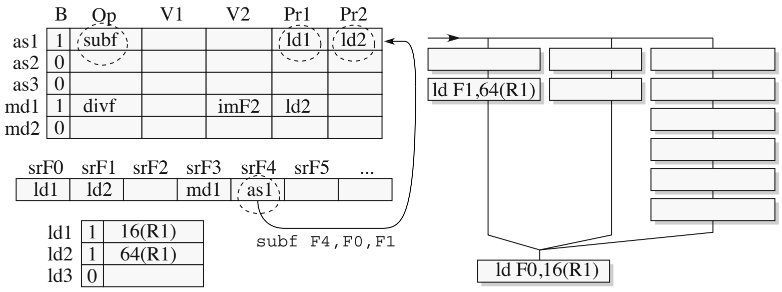
	B	Op	V1	V2	Pr1	Pr2
as1	0					
as2	0					
as3	0					
md1	1	divf		imF2	ld2	
md2	0					

srF0	srF1	srF2	srF3	srF4	srF5	...
ld1	ld2		md1			

ld1	1	16(R1)
ld2	1	64(R1)
ld3	0	

Figure 13.35. *Cycle 3:* `divf` *in phase R.* `ld F0` *is in the memory access phase*

copied in the `V2` field (`imF2` is an avatar of `F2`). `srF3` receives `md1` to indicate that we must wait for the execution of `divf` to be completed before getting the last value of `F3`.



	B	Op	V1	V2	Pr1	Pr2
as1	1	subf			ld1	ld2
as2	0					
as3	0					
md1	1	divf		imF2	ld2	
md2	0					

srF0	srF1	srF2	srF3	srF4	srF5	...
ld1	ld2		md1	as1		

ld1	1	16(R1)
ld2	1	64(R1)
ld3	0	

Figure 13.36. *Cycle 4:* `subf F4,F0,F1` *in phase R*

– `subf F4,F0,F1`: the entry `as1` is assigned to the `subf` instruction, which will have to wait for the execution of `load F0` and `load F1` to be completed (Figure 13.36). The `Pr2` therefore receives `ld2`. `srF4` is set with `as1`.

– `mulf F5,F3,F0`: the `md2` entry is assigned to `mulf` (Figure 13.37). The latter will have to wait for `F3`. The `Pr1` therefore receives `md1`. `srF5` receives `md2`.

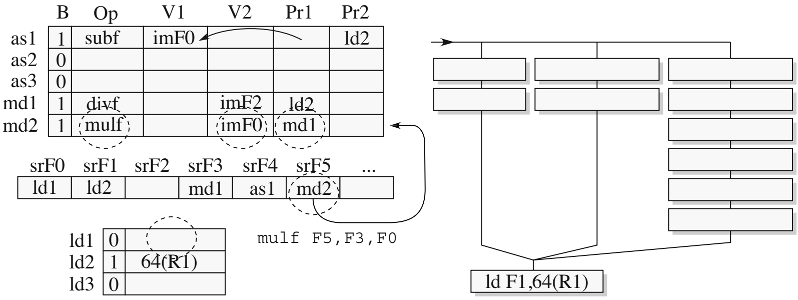


Figure 13.37. *Cycle 5:* `mulf F5,F3,F0` *in phase R*

The `load F0` is completed. The `V1` is now loaded with the expected value (avatar `imF0`).

– `addf F0,F4,F1`: the `as2` entry is assigned to the `addf` instruction, which has to wait for the execution of `subf F4` to be completed (Figure 13.38). The `Pr1` field receives `as1`, `V2` receives the value `imF1` (`load F1` is completed) and `srF0` receives `as2`, meaning that we must wait for the execution of the addition to be completed. Since the `ld2` entry is now free, the `ld2` descriptors become `imF1` in `V` type fields. `subf` and `divf` enter the execution phase. The `imF0` field of the `subf` contains as expected the former value of `F0` (resolution of the WAR hazard using the avatars). There is no risk of the `addf` modifying it.

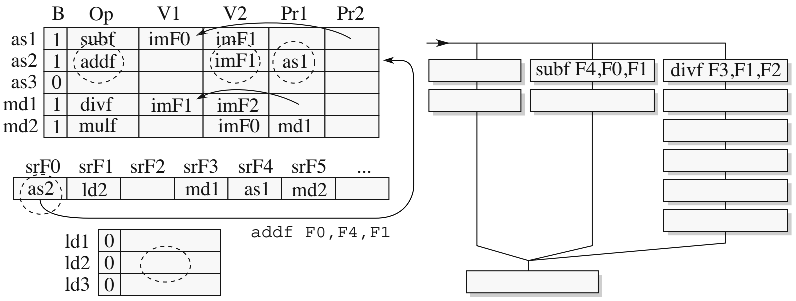
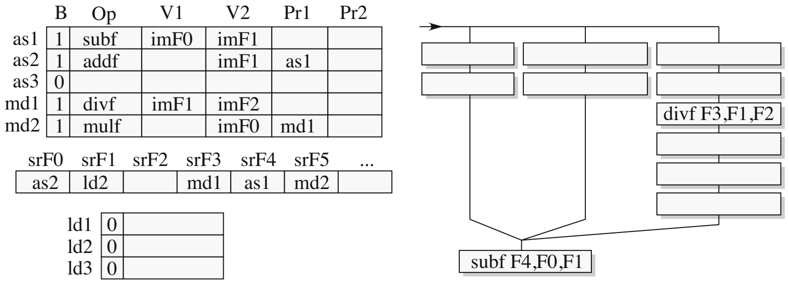


Figure 13.38. *Cycle 6:* `addf F0,F4,F1` *in phase R*

The `mulf` instruction cannot proceed because `divf` is not completed.

– `subf F4,F0,F1` reaches phase WB (Figure 13.39).



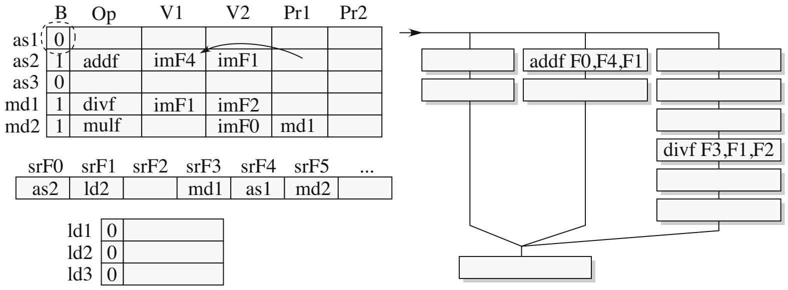
	B	Op	V1	V2	Pr1	Pr2
as1	1	subf	imF0	imF1		
as2	1	addf		imF1	as1	
as3	0					
md1	1	divf	imF1	imF2		
md2	1	mulf		imF0	md1	

srF0	srF1	srF2	srF3	srF4	srF5	...
as2	ld2		md1	as1	md2	

ld1	0	
ld2	0	
ld3	0	

Figure 13.39. *Cycles 7 and 8: execution of* `subf`*and* `divf`

– `F4` is updated in the `V1` of `as2` (Figure 13.40). `addf` can proceed to the execution phase.



	B	Op	V1	V2	Pr1	Pr2
as1	0					
as2	1	addf	imF4	imF1		
as3	0					
md1	1	divf	imF1	imF2		
md2	1	mulf		imF0	md1	

srF0	srF1	srF2	srF3	srF4	srF5	...
as2	ld2		md1	as1	md2	

ld1	0	
ld2	0	
ld3	0	

Figure 13.40. *Cycle 9:* `F4` *is updated, along with the associated fields*

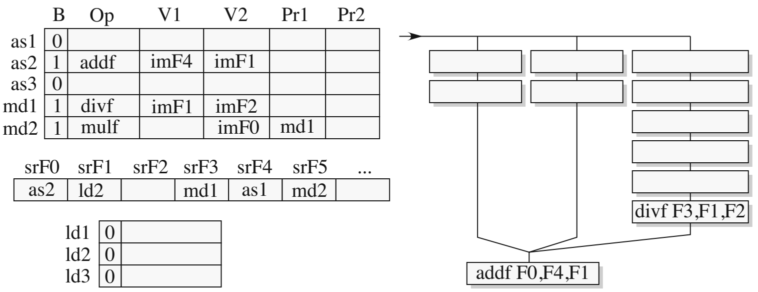
– `divf` and `addf` proceed (Figure 13.41) down their respective pipelines.

– `F0` is now available (Figure 13.42).

– `mulf` can resume because `F3` is available. There is no problem with `F0` (WAR hazard) since `mulf` uses the avatar, and not the last value written by `addf` (Figure 13.43).

The corresponding execution diagram is as follows:

The arrows indicate hazard resolution.

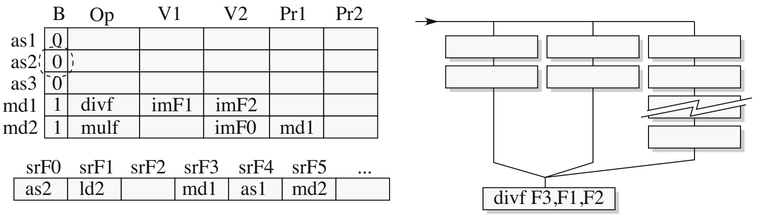


	B	Op	V1	V2	Pr1	Pr2
as1	0					
as2	1	addf	imF4	imF1		
as3	0					
md1	1	divf	imF1	imF2		
md2	1	mulf		imF0	md1	

srF0	srF1	srF2	srF3	srF4	srF5	...
as2	ld2		md1	as1	md2	

ld1	0	
ld2	0	
ld3	0	

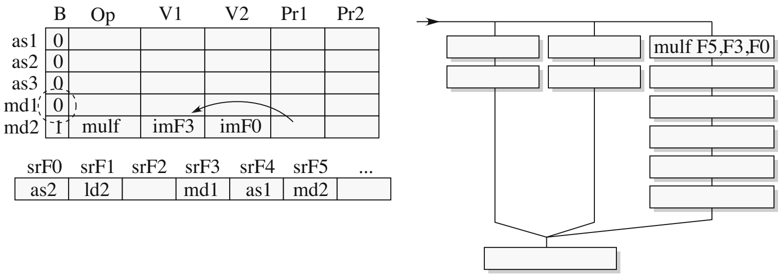
Figure 13.41. *Cycles 10 and 11:* `addf` *in phase WB*



	B	Op	V1	V2	Pr1	Pr2
as1	0					
as2	0					
as3	0					
md1	1	divf	imF1	imF2		
md2	1	mulf		imF0	md1	

srF0	srF1	srF2	srF3	srF4	srF5	...
as2	ld2		md1	as1	md2	

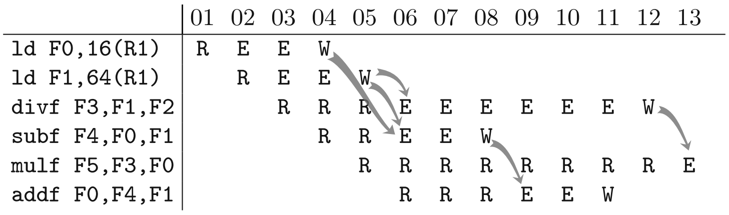
Figure 13.42. *Cycle 12:* `divf` *is in WB*



	B	Op	V1	V2	Pr1	Pr2
as1	0					
as2	0					
as3	0					
md1	0					
md2	1	mulf	imF3	imF0		

srF0	srF1	srF2	srF3	srF4	srF5	...
as2	ld2		md1	as1	md2	

Figure 13.43. *Cycle 13:* `mulf` *resumes*



	01	02	03	04	05	06	07	08	09	10	11	12	13
`ld F0,16(R1)`	R	E	E	W									
`ld F1,64(R1)`		R	E	E	W								
`divf F3,F1,F2`			R	R	R	E	E	E	E	E	E	W	
`subf F4,F0,F1`				R	R	E	E	W					
`mulf F5,F3,F0`					R	R	R	R	R	R	R	R	E
`addf F0,F4,F1`						R	R	R	E	E	W		

13.3.7. *Loop execution and WAW hazards*

Consider the architecture in Figure 13.44.

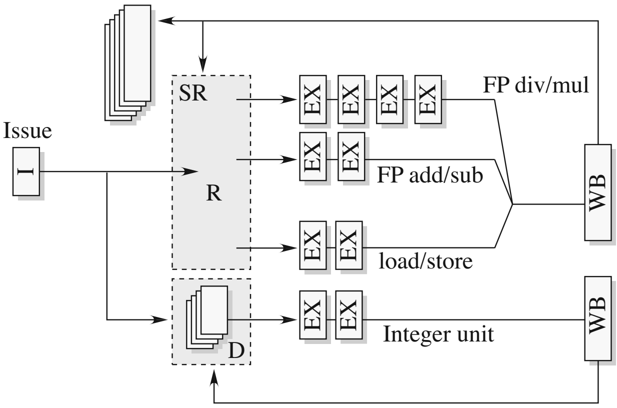


Figure 13.44. *Example of a superscalar architecture*

The "integer unit" pipeline is in charge of branches and of handling processes on the integer registers $\mathtt{r}_k$, as we saw with the DLX.

The reservation station is equipped with three entries associated with each of the pipelines `mulf/divf`, `addf/subf` and `ld/st`. These entries are supposed to be free when the execution of the loop begins.

```
I1: loop1: ld   f4,0(r2)
I2:        divf f1,f4,f2
I3:        stf  f1,64(r2)
I4:        sub  r2,r2,#4
I5:        bnz  r2,loop1
```

We assume that there is a miss on the first access to the data in memory. The delay incurred by this miss is seven cycles long. For each following `ld`, access takes one cycle.

The prediction scheme ensures that the branch is resolved without any penalty.

Let $\mathtt{op}_k$ be the `op` instruction in the k-th iteration of the loop:

– $t = 1$: $\mathtt{ld}_1$ begins SR and sets `srF4`.

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
ld$_1$	R	E$_1$	E$_2$	E$_2$	E$_2$	E$_2$	E$_2$	E$_2$	E$_2$	W															
divf$_1$		R	R	R	R	R	R	R	R	R	E	E	E	E	W										
stf$_1$			R	R	R	R	R	R	R	R	R	R	R	R	R	E$_1$	E$_2$	W							
sub$_1$				D	E	E	W																		
bnz$_1$					D	D																			
ld$_2$						R	R	E$_1$	E$_1$	E$_2$	W														
divf$_2$							R	R	R	R	R	E	E	E	E	W									
stf$_2$								R	R	R	R	R	R	R	R	R	E$_1$	E$_2$	W						
sub$_2$									D	E	E	W													
bnz$_2$										D	D														
ld$_3$											R	R	E$_1$	E$_2$	E$_2$	E$_2$	W								
divf$_3$												R	R	R	R	R	R	E	E	E	E	W			
stf$_3$													R	R	R	R	R	R	R	R	R	R	E$_1$	E$_2$	W
sub$_3$														D	E	E	W								
bnz$_3$															D	D									

– $t = 2$: divf$_1$ begins SR. Since srF4 was set by ld$_1$, divf$_1$ waits for the avatar of its first source operand f4 to be updated by ld$_1$.

– $t = 2$ to $t = 9$: ld$_1$ is on hold until the cache is updated.

– $t = 3$: stf$_1$ proceeds to SR. Since it is involved in a RAW hazard over f1 with divf$_1$, it waits for f1 to become available.

– $t = 4$ and $t = 5$: sub$_1$ and bnz$_1$ proceed to the decoding phase D. The address of the target instruction is less than the address of the branch instruction, the address of the target instruction is loaded into the program counter. We return to the beginning of the loop.

– $t = 6$: ld$_2$ is in ST and performs its own update of srF4 *by overwriting what is stored there* (resolution of the WAW of ld$_2$ with ld$_1$).

– $t = 7$: divf$_2$ proceeds to SR. Since srF4 was set by ld$_2$, divf$_2$ waits for the avatar of the first source operand f4 to be updated by ld$_2$. Note here the role of the avatars: each pair ld$_k$/divf$_k$ works with its own copy of f4 (resolution of the WAW hazard). ld$_2$ is still in SR, waiting for the update of r2 by sub$_1$.

– $t = 8$: ld$_1$ and ld$_2$ are currently being executed, creating two free spots that allow stf$_2$ to proceed to SR. stf$_2$ is blocked because of the RAW hazard with divf$_2$. Note that ld$_2$ cannot progress as long as ld$_1$ has not completed its memory read access (phase E2).

– $t = 9$: end of the memory read access for ld$_1$.

– $t = 10$: WB of ld$_1$ (update of f4 and of the avatar in the entry assigned to divf$_1$). ld$_2$ performs the memory read access.

– $t = 11$: ld$_3$ proceeds to SR (see ld$_1$ and ld$_2$). It completes this phase when r2 becomes available. Note that the ld/st pipeline is accessible for execution.

– $t = 12$: $\texttt{divf}_2$ proceeds to the execution phase. $\texttt{divf}_3$ enters the SR phase and stalls for the same reasons as before.

REMARKS.–

– In the use of a scoreboard, the SRF register is used for detecting RAW and WAW hazards that leave instructions stalled. In the case of the Tomasulo algorithm, SRF is used for resolving hazards.

– The $\texttt{srF}_k$ entry is overwritten in the R phase of the instruction that will modify $\texttt{F}_k$: it is used to indicate the unit that is supposed to produce the value. Thus, an instruction involved in a RAW hazard with the write instruction no longer depends on a register or an instruction, but is simply waiting for a value fetched from the CDB.

– RAW hazards are resolved using avatars (see the example of the loop). On the other hand, $\mathrm{W}_m\mathrm{AW}_m$ hazards (i.e. between `store` instructions that modify the same memory word) always result in stalled instructions.

– The beginning of the R phase occurs in the same order as the elements in the instruction buffer. This guarantees proper handling of the dependencies. Consider the following example:

```
divf f2,f3,f1
addf f2,f5,f6
stf f2,r0(48)
```

The `stf` will find in `srF2` a reference to a unit that processes additions. Although `divf` will probably be completed after the `addf`, the `stf` will in fact read the value that was produced by `addf`, since it will find in `srF2` the reference of the last instruction that modified `f2`.

– If we continue to go through the iterations of the loop used as an example, we start to notice a *periodicity of operation* that differs from the periodicity of the program (see example 13.1).

– The use of avatars makes it possible to simultaneously execute the same instruction in different iterations: for example, the pairs $\texttt{ld}_1$/$\texttt{divf}_1$ and $\texttt{ld}_2$/$\texttt{divf}_2$ are executed in parallel on different copies of `f4`.

13.4. VLIW architectures

13.4.1. *Limits of superscalar architectures*

These systems we have described for improving performances rely on alterations made to classic architectures:

– pipeline architectures provided an initial improvement;

– superscalar architectures built with several pipelines brought on an additional increase in the level of performance.

These improvements are constrained by:

– data hazards;

– problems related to sequence breaks.

The use of a scoreboard or of the Tomasulo algorithm is a solution for handling data hazards, whereas prediction techniques are applied to reduce the number of penalties in the case of branches. It is possible to improve performances further by rearranging the order (O^{3}E, *out-of-order execution*) of the instructions in the *prefetch buffer*. The solutions implemented to achieve this rescheduling rely on very complex hardware and, as a result, have limits to what can be achieved in terms of performance. Another issue is that this hardware can only "see" instructions that are currently being processed or that are present in the buffer. This is known as the "instruction window". This restricted visibility severely limits our ability to optimize the program.

An alternative to improving superscalar architectures consists of:

– building a simpler and better performing hardware architecture (with no rescheduling);

– delegating to the compiler the responsibility of providing a sequence of instructions that is optimized for this architecture. The "instruction window" is then limited only by software.

Very long instruction word (VLIW) architectures were introduced to satisfy these two conditions.

These are the same arguments that were raised in favor of RISC architectures, when computer designers were faced with the overwhelming complexity of CISC architectures.

13.4.2. *VLIW architectures*

The Multiflow Trace-7 (1984) [JOS 83] and the Cydrome Cydra-5 (1984) [DEH 93] are the first two implementations of the VLIW model. The best known realizations today are the Itanium by Intel® and the TMS320C6x$^{\text{TM}}$ by Texas Instruments. These architectures are characterized by the fact that several instructions are read in memory during each clock cycle (seven for the Multiflow, six for the Itanium and eight for the TMS). It is the responsibility of the compiler to provide the instructions that ensure the proper execution of the program.

The Itanium is equipped with nine processing units, including two floating-point vector units, two integer units, two "`load/store`" units and three branch units. The pipelines all have a length of 10 (Figure 13.45).

The TMS320C60 is equipped with eight processing units (Figure 13.46).

Bundles of eight instructions, known as *fetch packets*, or *instruction windows*, are read in memory simultaneously. The instruction window can be divided into several *execute packets*. Each of these contains instructions that can be executed in parallel. The value 1 or 0 of the least significant bit (p bit), also called the *parallel* bit, of the instruction code, indicates if the following instruction (the one "to the right") is part of the packet, or if the packet is closed (see Figure 13.47).

The fetch packet in Figure 13.47 contains four execute packets. The first one consists of the single instruction I_0. The second contains I_0, I_1, I_2, I_3, etc. The configuration corresponds to an execution of the instructions according to the sequence:

```
Cycle Instructions
 n    I0
 n+1  I1 I2 I3 I4
 n+2  I5
 n+2  I6 I7
```

13.4.3. *Predication*

Predication is a method used with VLIW architectures to handle branching instructions. The idea is to let two branches of a branching instruction be executed, and to complete the branch that is the actual target. In each branch, the instructions are assigned a Boolean variable, known as the *predicate*. The execution of these instructions is made conditional on the value of the predicate.

13.4.3.1. *TMS320C6000 architecture*

In the TMS320C6000 architecture [TEX 10], the instructions guarded by a predicate are referred to as *conditional operations*. A 4-bit field is used on the most significant side of the instruction code: 3 bits (*creg*) indicate the register used for the condition (B0, B1, B2, A1 or A2) and 1 bit (z) indicates whether we are testing for an equality with 0 ($z = 1$) or a non-equality with 0 ($z = 0$).

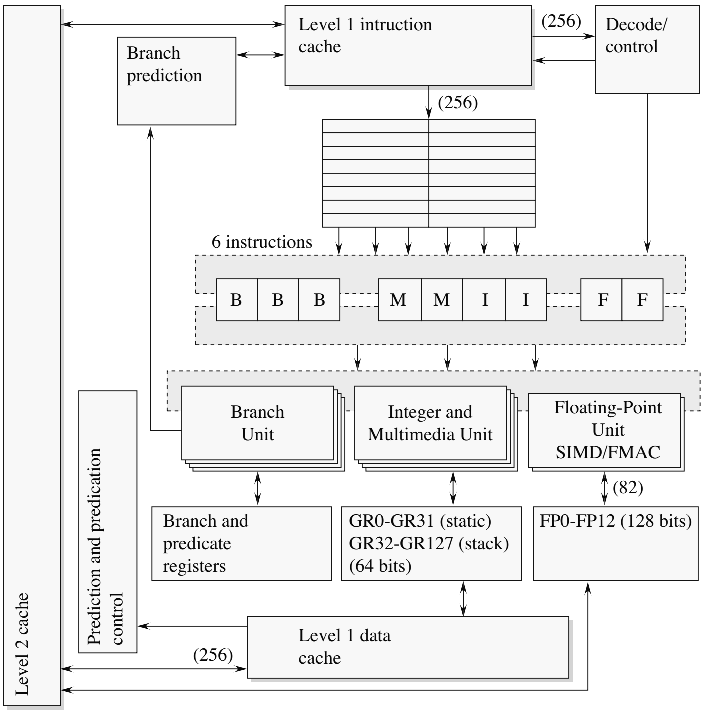

Level 1 intruction cache
(256)
Decode/ control
Branch prediction
(256)
6 instructions
B
B
B
M
M
I
I
F
F
Branch Unit
Integer and Multimedia Unit
Floating-Point Unit SIMD/FMAC
(82)
Branch and predicate registers
GR0-GR31 (static) GR32-GR127 (stack) (64 bits)
FP0-FP12 (128 bits)
Prediction and predication control
Level 1 data cache
(256)
Level 2 cache

Figure 13.45. *Structure of the Itanium®*

Example:

```
║   [B0]  ADD .L1 A1,A2,A3
║ || [!B0] ADD .L2 B1,B2,B3
```

The two addition instructions are mutually exclusive. They can be executed in parallel, since they are independent from a resource perspective, but only one of them, depending on the value of B0, will complete its execution.

13.4.3.2. *Itanium® architecture*

In the Itanium, 6 bits of the instruction code (which is 41 bits in length) indicate the number of a predicate bit pr_k, $k = 0, ..., 63$, associated with that instruction. Bundles of three instructions are delivered to the processor simultaneously according to the format:

Instruction (41 bits)	Instruction (41 bits)	Instruction (41 bits)	Template (5 bits)

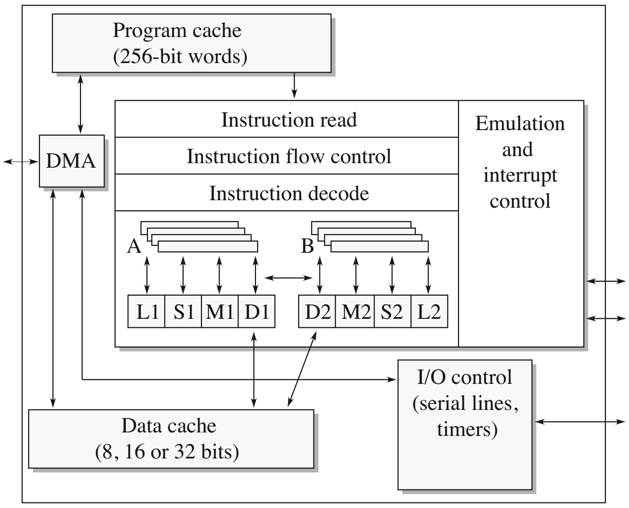


Figure 13.46. *Structure of the TMS320C60*

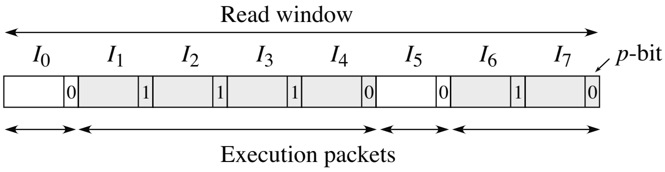


Figure 13.47. *Fetch packet and execute packets in the TMS320C60*

The five *template* bits indicate which instructions can be executed in parallel. The instruction format is as follows:

Operation code (13 bits)	Predicate reg. (6 bits)	Reg. 1 (7 bits)	Reg. 2 (7 bits)	Reg. 3 (7 bits)

Predicates are initialized for each instruction (there are only a few instructions that cannot be *guarded*). If the predicate is equal to 1, the instruction is simply executed as usual. Otherwise, the instruction does not modify the machine in any way.

For conditional branches, all of the instructions in one branch are assigned a predicate bit pr_i equal to 1, while all of the instructions in the other branch are assigned a pr_j equal to 1. The result of the comparison will set these bits to 0 or 1. The comparison instructions have the following syntax:

```
(p) cmp.rel pri,prj = src1,src2;;
```

where pri and prj are the predicate bits and src1 and src2 are the operands of the comparison rel, for example, or eq, ne, etc.

The indication (p) in front of the comparison indicates that the comparison itself is also conditional on a predicate bit.

The instruction can be translated as:

```
if ( p ) {
   if ( src1 rel src2 ) {
      pri = 1; prj = 0;
      }
   else {
      pri = 0; prj = 1;
   }
}
```

Execution is abandoned for instructions that have this bit equal to 0. Example:

```
if (r1)
   r2 = r3 + r4;
else
   r5 = r6 - r7;
```

gives us:

```
      cmp.ne p1,p2 = r1,0;;
(p1)  add    r2 = r3,r4
(p2)  sub    r5 = r6,r7
```

The lack of a predicate p in front of cmp.ne indicates that the instruction is not “guarded”, and therefore always executed.

13.5. Exercises

Exercise 13.1 (Periodicity of operation) (hints on page 344)
Going back to the architecture of Figure 13.44 and the corresponding example (page 313), execute two additional iterations and find the value of the period of operation.

Exercise 13.2 (Examples of hazards) (hints on page 345)
REMINDER: Each processing unit is assigned a set of registers known as a *reservation station* (RS). This station provides a decision mechanism for *issuing* the instructions into different pipelines.

Each RS is equipped with several entries, which are used as follows: if instruction I_m (stored in rs_i) depends on I_n (stored in rs_j) for one of its operands k, this dependency is indicated by writing in the field Pr_k of the entry rs_i associated with I_m the name of the entry (rs_j) associated with I_n (Figure 13.28). When I_n is completed, the value v_n of operand k is stored in field V_k of entry rs_i.

Reservation stations are handled as follows:

– Instructions are read *in the same order as in the program.*

– The flow of instructions is controlled by storing them in a queue.

– An entry rs_i is assigned, if possible (if there are enough entries), in RS, and for each instruction, as indicated above.

– For instructions other than `load`, the information on the availability of the operands is stored in rs_i (Figure 13.28):

- If an operand is available, its value is copied in a field V_k of rs_i: an *avatar* of this register is created (*register renaming*).

- If an operand is not available ($\mathrm{I}_m \prec \mathrm{I}_n$), the entry rs_j associated with the instruction I_m, which is supposed to deliver the value of the operand, is written into a field Pr_k of rs_i (assigned to I_n).

– For memory accesses, the *address* of this entry is memorized (Figure 13.29).

The rs_k associated with the instructions that have these registers as targets are stored in a specific RS called a Status Register File (SRF). rs_k is then copied in the rs_j of the instructions that require this register.

Consider this example:

```
I1: divf f2,f2,f1    ; f2 := f2/f1
```

```
I2: stf  f2,r0(r3)   ; f2 --> mem(r0(r3))
I3: mulf f2,f5,f6    ; f2 := f5*f6
I4: stf  f2,r0(r4)   ; f2 --> mem(r0(r4))
```

with the architecture from Figure 13.48.

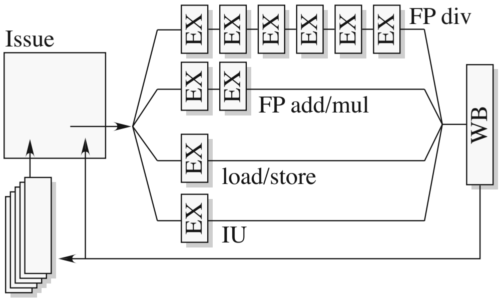


Figure 13.48. *Example of a superscalar architecture*

Applying the Tomasulo algorithm leads to the following execution diagram:

```
divf f2,f2,f1 | F   I   E   E   E   E   E   E   W
stf f2,r0(r3) |     F   I   .   .   .   .   .   I   E   W
mulf f2,f5,f6 |         F   I   E   E   W
stf f2,r0(r4) |             F   I   .   I   E   .   W
```

1) Indicate the types of hazards encountered.

2) Comment on the execution diagram.

3) Write the execution diagram for the sequence:

```
I1: divf f2,f2,f1  ; f2 := f2/f1
I2: stf  f2,r0(r3) ; f2 --> mem(r0(r3))
I3: mulf f2,f5,f6  ; f2 := f5*f6
I4: stf  f2,r0(r3) ; warning 0(r3) instead of 0(r4)
```

Exercise 13.3 (Execution of a loop) (hints on page 345)

Consider the architecture in Figure 13.49.

The following program loop is executed:

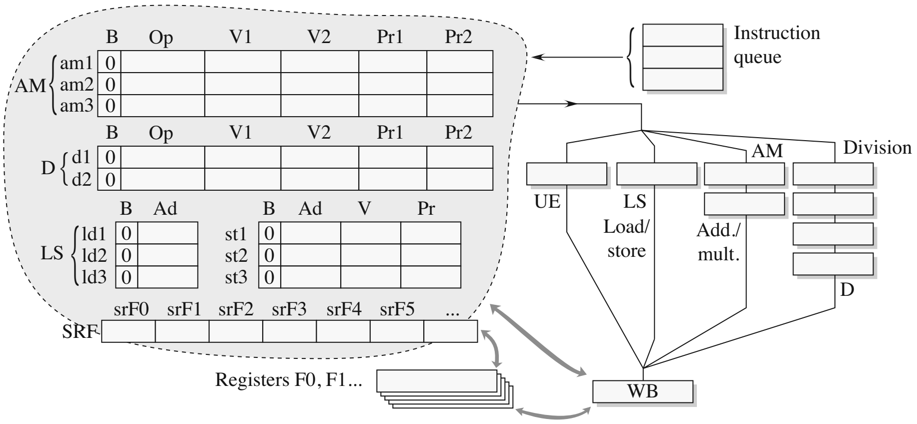


Figure 13.49. *Case study: the gray portion corresponds to the "issue" stage*

```
I1: loop1: ld   f4,0(r2)
I2:        divf f1,f4,f2
I3:        stf  f1,64(r2)
I4:        sub  r2,r2,#4
I5:        bnz  r2,loop1
```

1) We assume that the first access to the memory data results in a miss, and that the delay incurred is eight cycles long (the beginning of the execution phase is indicated by $\boxed{9}$ in the table shown). We also assume that the other `ld` instructions take only one cycle.

2) The instruction buffer is equipped with three entries.

3) To simplify this example, we will not represent the series of steps involved in executing the instructions `sub r2,r2,#4` and `bnz r2,bcle` in every loop: we saw that there is no WAR hazard and we will assume that the branch is resolved without any delay (accurate prediction).

The instructions proceed as follows ($\star$ means that there are neither precedence nor prediction problem with the instruction):

	01	02	03	04	05	06	07	08	$\boxed{9}$	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
ld$_1$ f4,0(r2)	I	E	E	E	E	E	E	E	E	W														
divf$_1$ f1,f4,f2		I	.	.	.	.	.	.	.	I	E	E	E	E	W									
stf$_1$ f1,64(r2)			I	.	.	.	.	.	.	.	.	.	.	.	I	E	W							
sub$_1$				⋆																				
bnz$_1$					⋆																			
ld$_2$ f4,0(r2)						I	.	.	I	E	W													
divf$_2$ f1,f4,f2							I	.	.	.	I	E	E	E	E	W								
stf$_2$ f1,64(r2)								I	.	.	.	.	.	.	.	I	E	W						
sub$_2$									⋆															
bnz$_2$										⋆														
ld$_3$ f4,0(r2)											I	E	W											
divf$_3$ f1,f4,f2																I	E	E	E	E	W			
stf$_3$ f1,64(r2)													I	.	.	.	.	.	.	.	I	E	W	
sub$_3$														⋆										
bnz$_3$															⋆									
ld$_4$ f4,0(r2)																I	I	E	W					
divf$_4$ f1,f4,f2																	I	I	I	E	E	E	E	W

(I: Issue, E: Execute, W: Write Back, the index indicates the iteration number.)

Comment on this diagram for each time $t = 1, 2, \ldots, 24$, with explicit descriptions of any hazards that occur.

PART 5

Appendices

Appendix A

Hints and Solutions

A1.1. The representation of information

Exercise 3.1. (Integer representation) (page 52)

The program:

```
#include <stdio.h>
int main (){
    unsigned char val2 = 0, val1;
    while(1){
        val1 = val2;
        val2++;
        if (val1 > val2){
printf("val1 (max. val.) = %d, val2 = %d\n",val1,val2);
            break;
            }
    }
    return 0;
}
```

produces

```
val1 (max. val.) = 255, val2 = 0
```

The number is coded in 8 bits and represents values between 0 and 255. With an integer declared as `short`, we would have:

`val1 (max. val.) = 32767, val2 = -32768,` which corresponds to two's complement, 16-bit coding.

Exercise 3.2. (Single precision floating-point representation) (page 52)

The representation `{00 00 72 64}`$_{10}$ = `{0000 0000 | 0000 0000 |` $\underline{\texttt{0}}\texttt{100}$ `1000 |` $\underline{\underline{\texttt{0}}\texttt{1000000}}$`}`$_2$. The sign bit (double underlined) is equal to `0` ("+"), the exponent (underlined) to `{1000 0000}`$_2$ (i.e. 128_{10}), hence the actual exponent $= 128 - 127 = 1$. The significand is `{1001 0...}`$_2$, i.e. `1.10010...`$_2 = 1.5625_{10}$, hence the result $2^1 \times 1.5625 = 3.125$.

Exercise 3.3. (Fixed-point and floating-point representations) (page 52)

1) - Sign: `1`;

- Integer part: `11`;

- Fractional part: `1 || 101 0001 1110 1011 1000 0 || 110 1...` (the `||` indicates the period).

```
11.1101 0001 1110 1011 1000 0110 =
1.11101 0001 1110 1011 1000 0110 2**1
```

which gives us, after rounding:

```
1   110 0000 (1...) --> E0 --> E1
0   111 1010        --> 7A
0 | 111 0100        --> 74
1 | 100 0000        --> C0
```

2) We multiply by $2^{12} = 4{,}096$, i.e. $x \times 4{,}096 = -15{,}647$ (after rounding). We convert $15{,}647$ to base 2: `0011 1101 0001 1111`$_2$. We then change the sign. We get, noting the position of the decimal point:

`1100 . 0010 1110 0001`$_2$ = `C.2E1`$_{16}$.

Exercise 3.4. (Short floating-point representation) (page 52)

"Implied most-significant non-sign bit" (IMSNB) should be understood as the first bit (least significant) providing an indication of the sign. Example: 32_{10} is written $\ldots 00100000_2$. The 0 bit that precedes the 1 provides an indication of the sign ($0 \rightarrow +$) and is the IMSNB. Another example: $-16_{10} = \ldots 111011111_2$ and the 1 that precedes the 0 provides the sign ($1 \rightarrow -$) and is the IMSNB.

1) Dynamic (note that the number of significant digits is on the order of 4):

i) The largest positive number is represented by $x = 01.11\ldots1 \times 2^7$:

$$\boxed{\texttt{0111}}\boxed{\texttt{0}}\boxed{\texttt{1111 1111 111}}$$

the sign and significand of which are read as: `0 | 1.1111 1111 11`.

The largest positive number is given by: $x = (2 - 2^{-11}) \times 2^7 = 2.5594 \times 10^2$.

ii) The smallest positive number is represented by $x = 01.00\ldots00 \times 2^{-7}$ since the exponent -8 is reserved:

$$\boxed{\texttt{1000}}\,\boxed{\texttt{0}}\,\boxed{\texttt{1000 0000 000}}$$

the sign and the significand of which are read as:

`0 | 1.0000 0000 00`, i.e. 1×2^{-7}.

The smallest positive number is given by:

$x = 1 \times 2^{-7} = 7.8125 \times 10^{-3}$.

iii) Smallest negative number: we are faced with the problem of representing negative fractional numbers in two's complement. We begin with the significand. Consider the representation $10.f_2$ with $f = 00\ldots0$. We know this is a negative number. The positive value corresponding to this number is what needs to be added to it to get 0 in the format `xx.x... x`, i.e. $10.00\ldots0_2 = 2_{10}$. Therefore, $x = 10.00\ldots0_2 = -2_{10}$.

Switching representations between a positive number x and a negative number $\tilde{x}$ can be done by noting that $x + \tilde{x} = 2^2 = 4$. This amounts to taking the complement of the representation and adding 0.00000000001.

$\|x\|$	Representation	Representation	$-\|x\|$
ϕ	ϕ	`1 0.0000 0000 000`	-2
$2 - 2^{-11}$	`0 1.1111 1111 11`	`1 0.0000 0000 001`	$-2 + 2^{-11}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$1 + 2^{-11}$	`0 1.0000 0000 001`	`1 0.1111 1111 111`	$-1 - 2^{-11}$
1	`0 1.0000 0000 000`	ϕ	ϕ

The representation of the smallest negative number is given by:

$$\boxed{\texttt{1000}}\,\boxed{\texttt{1}}\,\boxed{\texttt{1000 0000 000}}$$

i.e. $x = -2 \times 2^7 = -2.5600 \times 10^2$.

iv) Largest negative number: based on the above table, we get $x = (-1 - 2^{11}) \times 2^{-7} \approx -7.8163 \times 10^{-3}$.

$$\boxed{\texttt{1001}}\,\boxed{\texttt{1}}\,\boxed{\texttt{0 1111 1111 11}}$$

2) Positive number:

$$\begin{aligned} {10^{-2}}_{10} : 0.01 \times 2 &= 0.02\\ 0.02 \times 2 &= 0.04\\ 0.04 \times 2 &= 0.08\\ 0.08 \times 2 &= 1.16\\ 0.16 \times 2 &= 0.32\\ 0.32 \times 2 &= 0.64\\ 0.64 \times 2 &= 1.28\\ 0.28 \times 2 &= 0.56\\ 0.56 \times 2 &= 1.12\\ 0.12 \times 2 &= 0.24\\ 0.24 \times 2 &= 0.48\\ 0.48 \times 2 &= 0.96\\ 0.96 \times 2 &= 1.92\\ 0.92 \times 2 &= 1.84 \end{aligned}$$

i.e. $\texttt{0.0001 0010 1000 11} = 01.0010100011 \times 2^{-4}$.

$$\boxed{\texttt{1100}}\,\boxed{\texttt{0}}\,\boxed{\texttt{1 0010 1000 11}}$$

3) Negative number: we start with the fractional part:

$$\begin{aligned} 0.3_{10} : 0.3 \times 2 &= 0.6\\ 0.6 \times 2 &= 1.2\\ 0.2 \times 2 &= 0.4\\ 0.4 \times 2 &= 0.8\\ 0.8 \times 2 &= 1.6\\ 0.6 \times 2 &= 1.2 \end{aligned}$$

i.e. $64.3 = \texttt{100 0000.01001 1001 1001 ...} = 01.0000000100 \times 2^6$ or $= 01.0000000101 \times 2^6$ after rounding. The representation -64.3 can be obtained directly:

$$\boxed{\texttt{0110}}\,\boxed{\texttt{1}}\,\boxed{\texttt{0 1111 1110 11}}$$

A1.2. The processor

Exercise 5.1. (Instruction sequencing) (page 101)

Access to the second instruction byte (immediate value access) is done in M2. The content of the program counter is used before it is incremented. In cycle M3, the memory write operation uses the pair “HL” as the address.

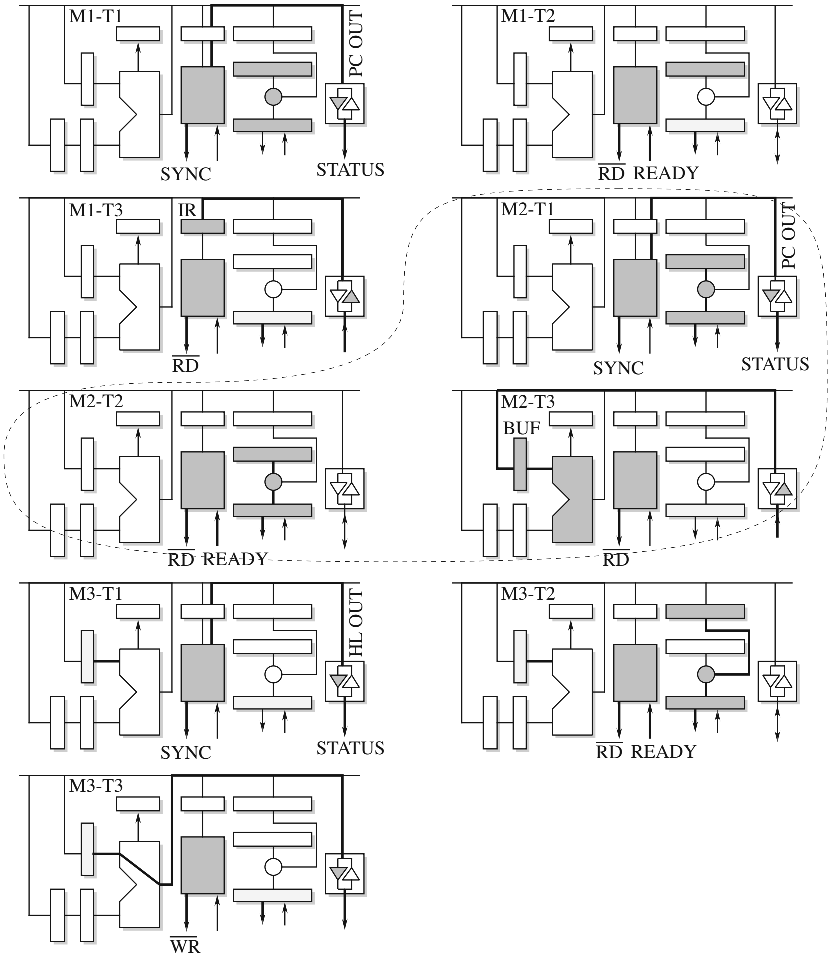


Figure A1.1. *Execution of an instruction*

A1.3. Inputs and outputs

Exercise 6.1. (Serial input) (page 133)

Based on the explanations of Figure 6.16, the read function is written as:

```
; Read function, byte in ah
loop1: mov al,[addrIN]   ; wait
       and al,1          ; test setting to zero
       jnz loop1
```

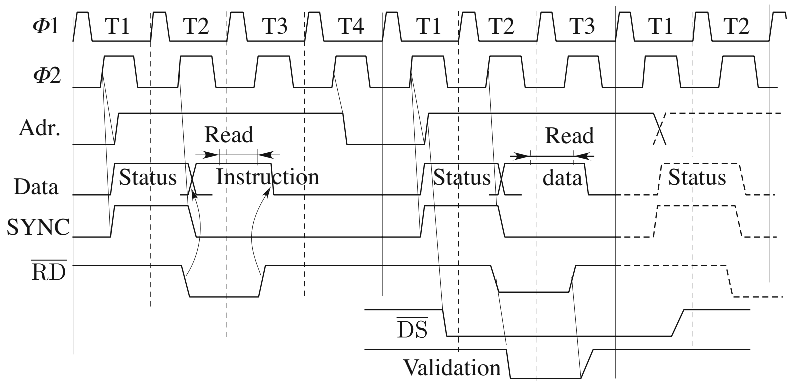


Figure A1.2. *Timing diagram for the read operation*

```
       call waitTs2      ; wait T/2 for check
       mov al,[addrIN]
       and al,1          ;
       jnz loop1
       mov cx,8
acql:  call waitT        ; wait T for next
       mov al,[addrIN]
       rcr al,1          ; store in ah
       rcr ah,1
       loop acql
       ret
```

The setting of the line to zero is tested, and after $T/2$, a check is performed. If it is still 0, the 8 bits of the character are acquired every T. A complete function would take into account all of the possible parameters, parity, number of data bits, number of stop bits, etc.

Exercise 6.2. (Serial output) (page 134)

```
; Write function of the content of ah
wfunc: xor al,al         ; al:=0
       mov [addrOUT],al  ; stop bit
       mov cx,8
wrlp:  call waitT        ; wait
       rcr ah,1
       rcl al,1          ; bit --> D0
       mov [addrOUT],al  ; write in flip-flop D
       loop wrlp
       ret
```

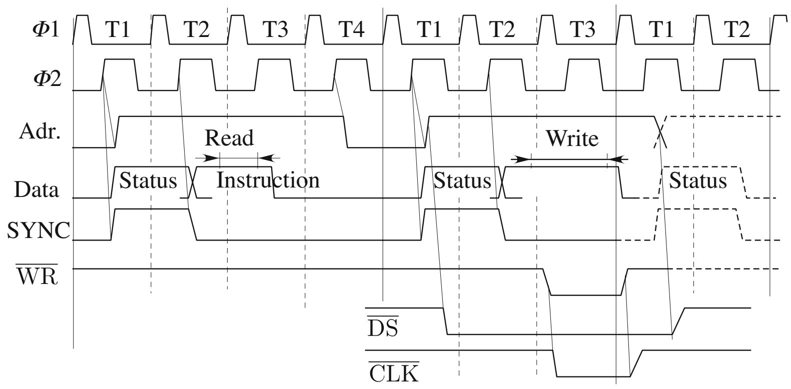


Figure A1.3. *Timing diagram for the write operation*

Figure A1.4(a) and (b) show one possible design for the input and output modules, respectively.

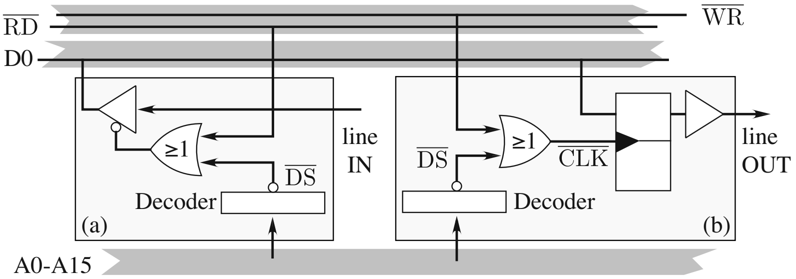


Figure A1.4. *Design diagram*

A1.4. Virtual memory

Exercise 9.1. (Virtual memory of the Am29000) (page 197)

1) General questions:

i) Operational diagram for converting “from virtual address to physical address”: see Figure 9.2.

ii) When the pages are present in memory and referenced in the TLB, accessing physical addresses becomes much faster.

iii) See Chapter 6.

2) MMU parameters:

i) The supervisor mode is a “privileged” mode of operation, in which the processor has access to the entire instruction set, in particular the instructions for accessing the TLB. For security reasons, not every user is authorized to access the TLB.

ii) The *tag field* receives the VPN that will be compared to the tag field of the virtual address sent by the processor. The *Valid* bit is used to ensure a minimum level of coherence between the TLB and the content of the memory, as in a standard cache.

iii) For 1 kB pages, the maximum number of addressable pages is $2^{n-10} = 2^{22} = 4$ Mpages. The physical number of the page is 22 bits in length.

iv) This TLB is a set-associative cache with two blocks. Each line is 64 bits in length.

v) See section III-2 in the documentation: the U is used for handling the age on each line with an LRU algorithm. Depending on its value, the update will be applied to block 0 (TLB Set 0) or block 1 (TLB Set 1).

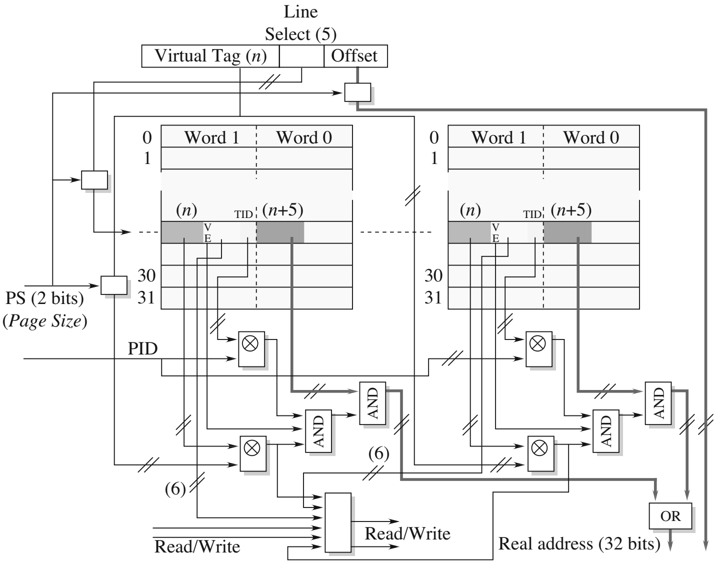


Figure A1.5. *Operation diagram of the TLB in the AMD29000*

vi) The index field still contains 5 bits. The virtual tag number is 14, 15, 16 or 17 bits in length, while the physical page number is 19, 20, 21 or 22 bits.

The TID field indicates the number of the process that owns the page. With the PID of the process currently being executed, it can be used to manage protected page access.

3) Design diagram (Figure A1.5).

A1.5. Pipeline architectures

Exercise 11.1. (Rescheduling) (page 252)

1) Without rescheduling, there are 12 instructions, and therefore 16 cycles, at best.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
lw r1,b[r0]	F	D	E	M	$\boxed{W}$												
lw r2,c[r0]		F	D	E	M	$\boxed{W}$											
add r3,r1,r2			F	D	.	$\boxed{D}$	E	M	$\boxed{W}$								
sw a[r0],r3				F	.	.	D	D	$\boxed{D}$	E	M	W					
lw r4,f[r0]							F	.	.	D	E	M	$\boxed{W}$				
lw r3,a[r0]										F	D	E	M	$\boxed{W}$			
sub r5,r3,r4											F	D	.	$\boxed{D}$	E	M	$\boxed{W}$

							18	19	20	21	22	23	24	25	26	27	28
sub r5,r3,r4	D	.	$\boxed{D}$	E	M	$\boxed{W}$											
sw d[r0],r5	F	.	.	D	.	$\boxed{D}$	E	M	W								
lw r6,g[r0]				F	.	.	D	E	M	$\boxed{W}$							
lw r7,h[r0]							F	D	E	M	$\boxed{W}$						
add r8,r7,r6								F	D	.	$\boxed{D}$	E	M	$\boxed{W}$			
sw e[r0],r8									F	.	.	D	.	$\boxed{D}$	E	M	W

We have a total of 28 cycles, hence a penalty of 12 cycles (12 instructions, therefore 16 cycles at best: (12 "F") + "DEMW").

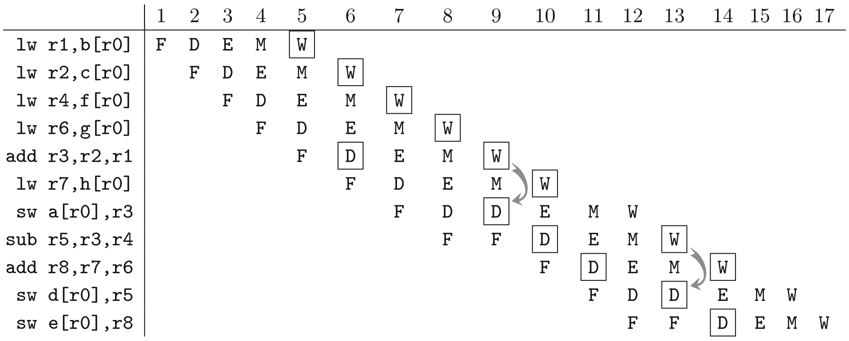
2) With rescheduling (Figure A1.6):

We have a total of 17 cycles, hence a penalty of one cycle compared to the unoptimized program.

Exercise 11.2. (Branches) (page 253)

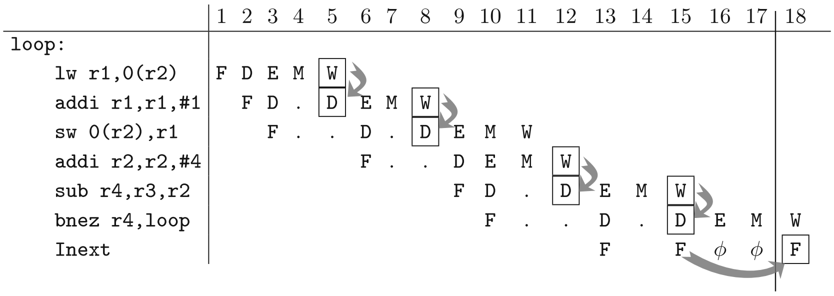
1) Without forwarding (Figure A1.7):

Number of loops: $400/4 = 100$. 17 cycles per loop ⇒ 1,700 cycles in all.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
lw r1,b[r0]	F	D	E	M	$\boxed{W}$												
lw r2,c[r0]		F	D	E	M	$\boxed{W}$											
lw r4,f[r0]			F	D	E	M	$\boxed{W}$										
lw r6,g[r0]				F	D	E	M	$\boxed{W}$									
add r3,r2,r1					F	$\boxed{D}$	E	M	$\boxed{W}$								
lw r7,h[r0]						F	D	E	M	$\boxed{W}$							
sw a[r0],r3							F	D	$\boxed{D}$	E	M	W					
sub r5,r3,r4								F	F	$\boxed{D}$	E	M	$\boxed{W}$				
add r8,r7,r6										F	$\boxed{D}$	E	M	$\boxed{W}$			
sw d[r0],r5											F	D	$\boxed{D}$	E	M	W	
sw e[r0],r8												F	F	$\boxed{D}$	E	M	W

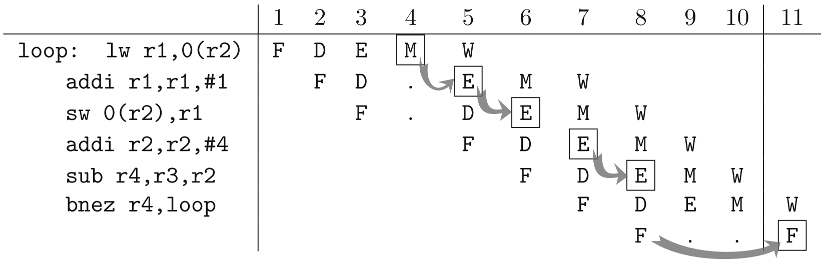
Figure A1.6. *Instruction rescheduling*



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
loop:																		
lw r1,0(r2)	F	D	E	M	$\boxed{W}$													
addi r1,r1,#1		F	D	.	$\boxed{D}$	E	M	$\boxed{W}$										
sw 0(r2),r1			F	.	.	D	.	$\boxed{D}$	E	M	W							
addi r2,r2,#4						F	.	.	D	E	M	$\boxed{W}$						
sub r4,r3,r2									F	D	.	$\boxed{D}$	E	M	$\boxed{W}$			
bnez r4,loop										F	.	.	D	.	$\boxed{D}$	E	M	W
Inext													F		F	ϕ	ϕ	$\boxed{F}$

Figure A1.7. *Loop without forwarding: ϕ indicates the delays incurred because of the branch execution. Until the target address has been calculated and loaded into the program counter, there are two hold cycles. The total number of cycles is 17. On the last line,* $\boxed{F}$ *denotes the fetching of the* `lw` *instruction*

2) With basic forwarding (Figure A1.8), we have a total of $100 \times 10 = 1{,}000$ cycles:



	1	2	3	4	5	6	7	8	9	10	11
loop: lw r1,0(r2)	F	D	E	$\boxed{M}$	W						
addi r1,r1,#1		F	D	.	$\boxed{E}$	M	W				
sw 0(r2),r1			F	.	D	$\boxed{E}$	M	W			
addi r2,r2,#4					F	D	$\boxed{E}$	M	W		
sub r4,r3,r2						F	D	$\boxed{E}$	M	W	
bnez r4,loop							F	D	E	M	W
								F	.	.	$\boxed{F}$

Figure A1.8. *Basic forwarding: the arrows indicate dependencies resolved using forwarding and the delay incurred because of the branch. There are 10 cycles per iteration*

3) With branch forwarding, the instruction following the branch is systematically executed. To illustrate, we move the `sw` instruction to immediately after the `bnezd`.

	1	2	3	4	5	6	7	8			
`loop1: lw r1,0(r2)`	$\boxed{\text{F}}$	D	E	M	W						
`addi r1,r1,#1`		F	D	.	E	M	W				
`addi r2,r2,#4`			F	.	D	E	M	W			
`sub r4,r3,r2`					F	D	E	M	W		
`bnezd r4,loop1`						F	D	E	M	W	
`sw 0(r2),r1`							F	D	E	M	W

We get $100 \times 7 = 700$ cycles.

Instead of keeping the `lw r1` in the loop, we could insert it once before the loop and once after the `bnezd`. We must, of course, make sure that these alterations do not modify the general behavior of the program.

Exercise 11.3. (Forwarding) (page 254)

– Forwarding from EX2 to EX1$_3$:

`add r1,r2,r3`	F1	F2	D	EX1	$\boxed{\text{EX2}}$	M1	M2	WB		
`sub r4,r5,r6`		F1	F2	D	EX1	EX2	M1	M2	WB	
`bnez r1,nexti`			F1	F2	D	$\boxed{\text{EX1}}$	EX2	M1	M2	WB

– Forwarding from M2 to EX1$_5$:

`lw r1,0(r2)`	F1	F2	D	EX1	EX2	M1	$\boxed{\text{M2}}$	WB				
`sub r4,r5,r6`		F1	F2	D	EX1	EX2	M1	M2	WB			
`...`												
`...`												
`bnez r1,nexti`					F1	F2	D	$\boxed{\text{EX1}}$	EX2	M1	M2	WB

– Forwarding from M1(WB) to M1:

`add r1,r2,r3`	F1	F2	D	EX1	EX2	$\boxed{\text{M1(WB)}}$	M2	WB	
`sw 0(r4),r1`		F1	F2	D	EX1	EX2	$\boxed{\text{M1}}$	M2	WB

Exercise 11.4. (Cache operation) (page 255)

The cache is a two-way set associative with one instruction per line. The four least significant bits select a fully associative, two-line cache, and the age is given by the LRU bit.

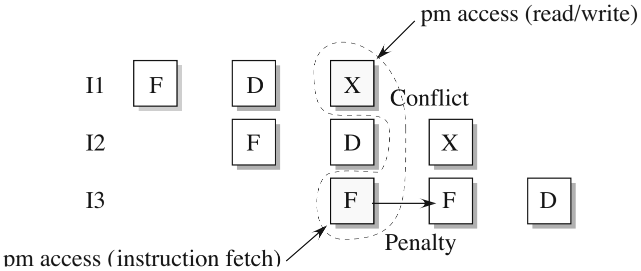


Figure A1.9. *Pipeline and cache*

1) Pipeline operation diagram (Figure A1.9): if `I1` accesses `pm` during the execution phase (Figure A1.9), it will be in conflict with the `I3` instruction, which is in phase `F`.

2) Only certain instructions are placed in the cache: those located at the address $n + 2$ in the event of a conflict with the current instruction with the address n, as indicated in Figure A1.9.

3) *Cache is enabled* indicates that the cache is operational for reading and writing. In the *frozen* mode, only read operations occur in the cache, and all updates are forbidden.

4) Cache structure:

- diagram of the cache (Figure A1.10);
- *Valid Bit*: coherence management;
- *LRU bit*: age management.

5) Consider the program in a loop situation. The first program memory access (instruction at 101) causes the instruction in 103 (in this case the conditional branch) to be stored in the cache with an index equal to 3.

When we enter the subroutine, the first access to 201 likewise causes the instruction in 203 to be stored in the cache with an index equal to 3. The instruction in 211 triggers the same event with the same index. The branch instruction is then overwritten by the instruction in 213. After the execution of the subroutine, moving on to instruction 101 leads to another miss, etc.

In all, we have a total of two *misses* for each execution of the loop. The cache operation is inefficient for this particular sequence (which has, however, a low probability of occurring).

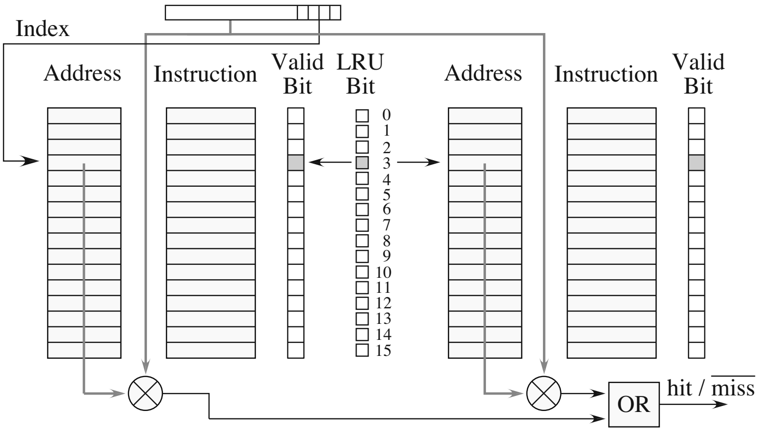


Figure A1.10. *The cache in the ADSP 2106x*

Exercise 11.5. (Branch forwarding) (page 258)

```
                 sgt r4,r1,r2
                 beqz r4,label1       |          sgt r4,r1,r2
                 nop                  |          beqz r4,label1
                 add r4,r2,r0     →   |          add r3,r1,r0
                 j label2             |          add r4,r2,r0
                 nop                  | label1:
      label1:    add r3,r1,r0
      label2:
```

Exercise 11.6. (Loop with annul bit) (page 258)

```
         lw r5,nmax(r0)
         add r4,r0,r0
         j label2            |          lw r5,nmax(r0)
         nop                 |          j,a label2
label1:  J1                  | label1:  J2
         ...             →   |          ...
         Jn                  |          Jn
         add r4,r4,#1        |          add r4,r4,#1
label2:  slt r6,r4,r5        | label2:  slt r6,r4,r5
         beqz r6,label1      |          beqz,na r6,label1
         nop                 |          J1
```

We gain one instruction per iteration.

Exercise 11.7. (Executing a subroutine call instruction) (page 259)

As in the case of branches, we lose three cycles (Figure A1.11).

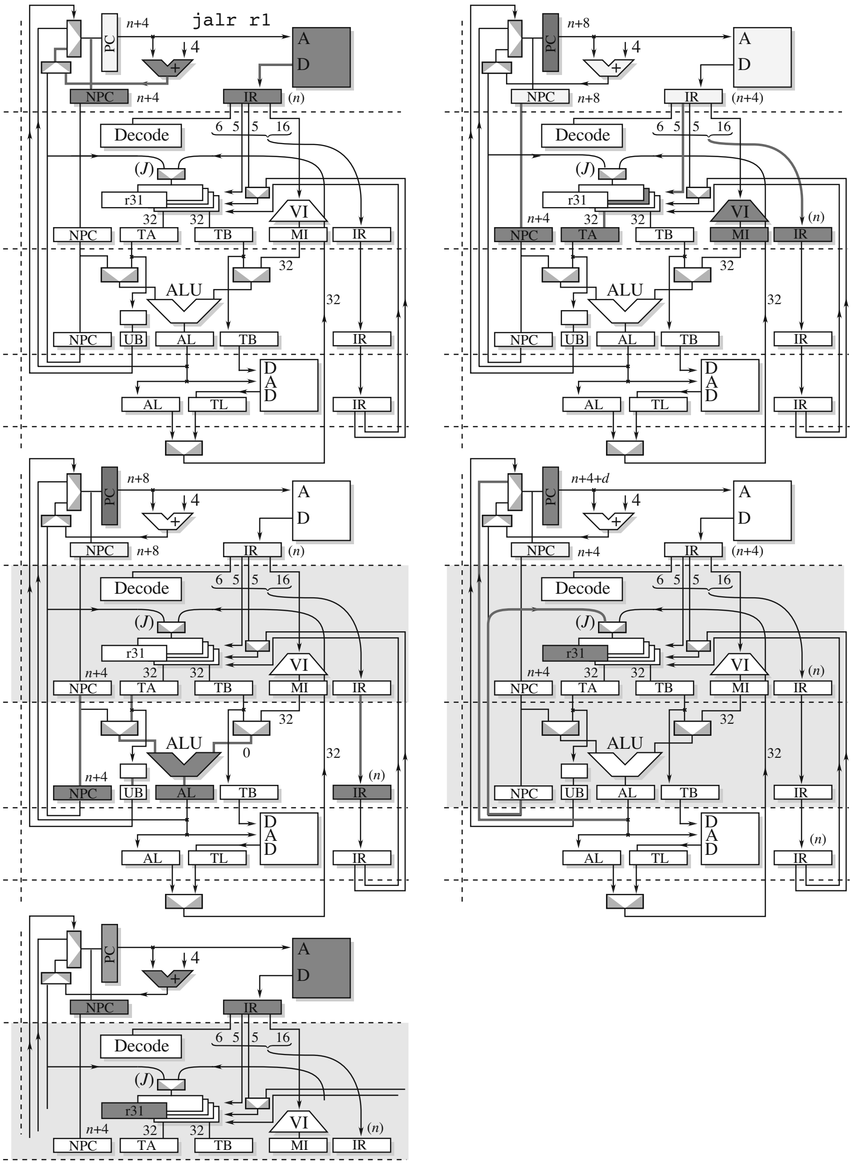


Figure A1.11. *Execution of the* `jalr`

A1.6. Caches in a multiprocessor environment

Exercise 12.1. (MSI protocol) (page 284)

Running the program leads to the exchanges shown in Figure A1.12.

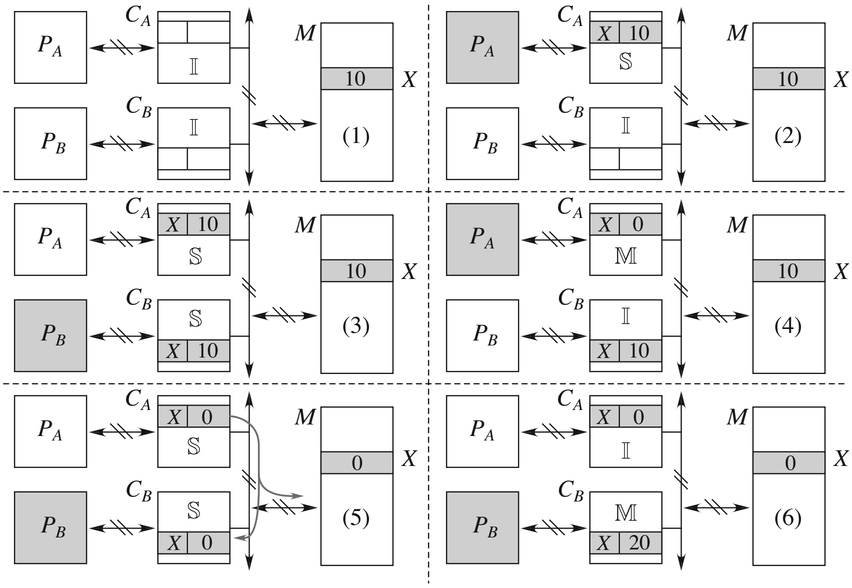


Figure A1.12. *Description of the MSI protocol operation*

REMARK A1.1.–

1) Initial situation.

2) Variable X is read by P_A: cache C_A is updated with the value 10. The line switches to the 𝕊 state in C_A.

3) Variable X is read by P_B: cache C_B is updated with the value 10. The line switches to the 𝕊 state in C_B.

4) P_A writes 0 in its cache. The line switches to the 𝕄 state. The lines in the other caches switch to 𝕀.

5) P_B wants to read X. P_A responds to the SR signal by writing the value it is storing in memory and switches to the 𝕊 state. Cache C_B is updated and the line switches to 𝕊.

6) P_B writes 20 in its cache. The line switches to the 𝕄 state. C_A invalidates the line.

Exercise 12.2. (MSI protocol, update sequences) (page 284)

The value of the index is 8 for each access (`[X]`, `[X+2]`, `[X+6]`). The line is therefore "shared". Figure A1.13 describes the execution of the protocol.

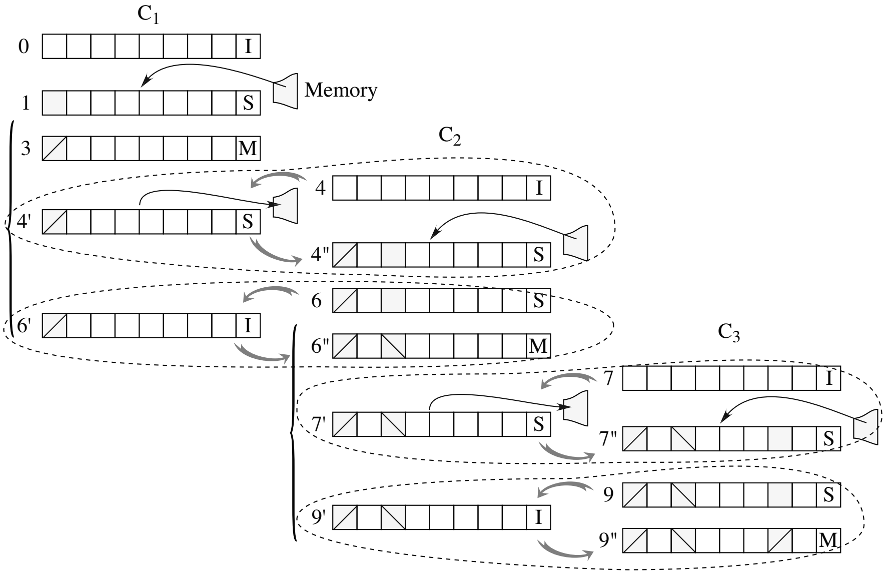


Figure A1.13. *Description of the protocol operation*

In step 4, the read request from P$_2$ triggers a memory update by P$_1$, with a switch to the $\mathbb{S}$ state for the corresponding line ($4'$). The memory transfer to C$_2$ is then performed, with a switch to the $\mathbb{S}$ state ($4''$).

Exercise 12.3. (MESI protocol) (page 285)

n			C_A	C_B	Memory
1	$R_A(X)10$	Miss	10-E		10
2	$R_B(X)10$	Miss	10-S	10-S	10
3	$W_A(X)0$	Invalidation in C_B	0-M	I	10
4	$R_B(X)0$	Coherence	0-S	0-S	0
5	$W_B(X)20$	Invalidation in C_A	I	20-M	0

Exercise 12.4. (Berkeley protocol) (page 285)

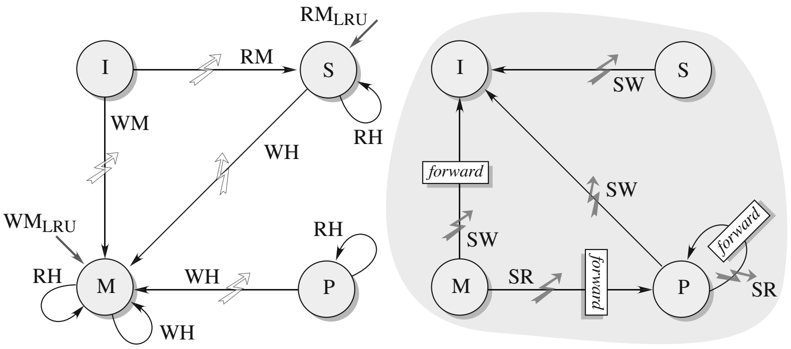


Figure A1.14. *Berkeley protocol: the processors P_0 and P_i*

REMARK A1.2.–

– Read Hit (RH): there is no communication on the bus. Accesses are performed without delays.

– Write Hit (WH): if the line is in the $\mathbb{M}$ state, access is performed with no exchange on the bus. If the line is in the $\mathbb{P}$ or $\mathbb{S}$ state, any possible copies are invalidated and the line switches to the $\mathbb{M}$ state.

– Read Miss (RM) and Write Miss (WM): see explanations on the *Read Miss* LRU and the *Write Miss* LRU.

Exercise 12.5. (Firefly protocol) (page 286)

A description of the protocol is shown in Figure A1.15.

– For an RH, there is no change of state.

– For an RM, the P$_i$ detect and signal possession of the information if applicable (by setting the *SharedLine*). In that case, either they all have the same data, since coherence is maintained in the $\mathbb{E}$ and $\mathbb{S}$ states, or the line is in the $\mathbb{M}$ state and it is first stored into memory. In any case, if the line is held somewhere else, it can be directly loaded from cache to cache instead of being read in memory.

If no other cache contains the data, a read operation is performed in memory and the state switches to $\mathbb{E}$.

– For a WH, if the line is in the $\mathbb{M}$ state, the only write operation is in the cache (*write-back*, *w-b*). If the line is in the $\mathbb{E}$ state, it is updated and switches to the $\mathbb{M}$ state. If the line is in the $\mathbb{S}$ state, write operations are performed in the cache and the

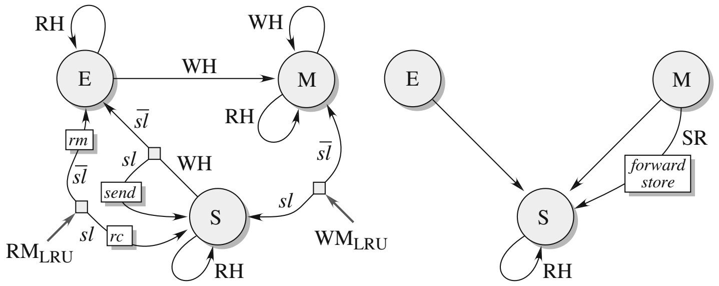


Figure A1.15. *Firefly protocol; broadcasting is ensured by the state of the SharedLine.* rm*,* rc *and* sl *stand for* read memory*,* read cache *and* SharedLine*, respectively*

memory (*write-through*, *w-t*). The other caches that have the line are set to *SharedLine* and update the line (*read*).

– A WM is handled like an RM followed by a WH. If the data are present in P$_i$, it is updated there and the state switches to $\mathbb{S}$. Otherwise, the line in P$_0$ changes to $\mathbb{M}$.

A line in the cache in the $\mathbb{E}$ or $\mathbb{S}$ state can be replaced at any time, since coherence is ensured.

A1.7. Superscalar architectures

Exercise 13.1. (Periodicity of operation) (page 321) The period of operation is equal to 2. Iterations 2 and 4, on the one hand, and 3 and 5, on the other, lead to identical sequences of steps.

	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
ld$_3$	R	R	E_1	E_2	E_2	E_2	W																		
divf$_3$		R	R	R	R	R	R	E	E	E	E	W													
stf$_3$			R	R	R	R	R	R	R	R	R	R	E_1	E_2	W										
sub$_3$				D	E	E	W																		
bnz$_3$					D	D																			
ld$_4$						R	R	E_1	E_2	W															
divf$_4$							R	R	R	R	E	E	E	E	E	W									
stf$_4$								R	R	R	R	R	R	R	R	R	E_1	E_2	W						
sub$_4$									D	E	E	W													
bnz$_4$										D	D														
ld$_5$											R	R	E_1	E_2	E_2	E_2	W								
divf$_5$												R	R	R	R	R	R	E	E	E	E	W			
stf$_5$													R	R	R	R	R	R	R	R	R	R	E_1	E_2	W
sub$_5$														D	E	E	W								
bnz$_5$															D	D									

Exercise 13.2. (Examples of hazards) (page 321)

1) Types of conflicts: $I_1 \prec I_2$, $I_2 \prec I_3$, $I_1 \prec I_3$, $I_3 \prec I_4$.

```
I1: divf f2,f2,f1
I2: stf  f2,r0(r3) ; RAW with I1
I3: mulf f2,f5,f6  ; WAR with I2, WAW with I1
I4: stf  f2,r0(r4) ; RAW with I3
```

2) Comment. – Since `f2` is renamed as `f2'` and `f2"`, where `f2'` and `f2"` are the avatars of `f2`, we can write:

```
divf f2',f2,f1 | F  I  E  E  E  E  E  E  W
stf f2',r0(r3) |    F  I  .  .  .  .  .  I  E  W
mulf f2",f5,f6 |       F  I  E  E  W
stf f2",r0(r4) |          F  I  .  I  E  .  W
```

Phase E of the `mulf` can be initiated before the end of the `stf`, since `mulf` can use `f5` and `f6`. The WAR with `stf` is resolved (avatar) and there is no hold in W. The resolution of the WAW between I_1 and I_3 satisfies the WAR.

3) Execution diagram:

```
divf f2',f2,f1 | F  I  E  E  E  E  E  E  W
stf f2',r0(r3) |    F  I  .  .  .  .  .  I  E  W
mulf f2",f5,f6 |       F  I  E  E  W
stf f2",r0(r3) |          F  I  .  I  E  .  .  .  W
```

The only difference with the previous sequence stems from the writing at the same address, which can only be done *in the order* of the program!

Exercise 13.3. (Execution of a loop) (page 322)

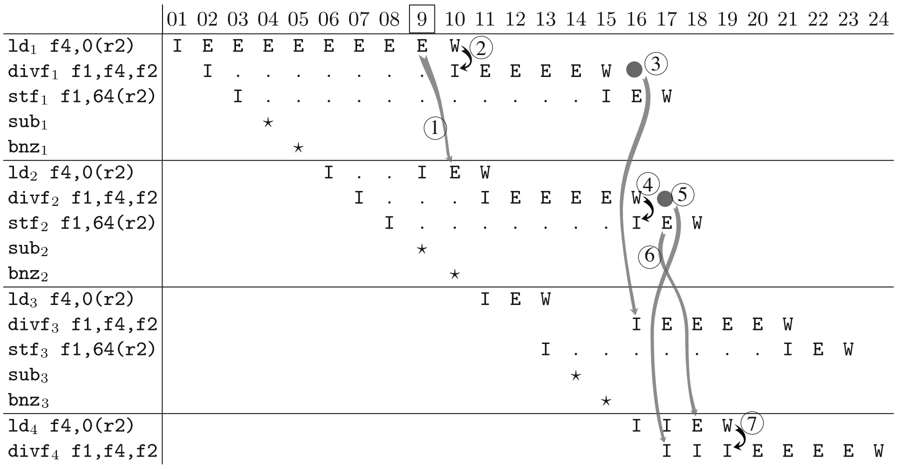
Comments of Figure A1.16:

– Arrows 2, 4 and 7: RAW conflict, the `divf` waits for the `ld` to perform its WB.

– Arrow 1: there is only one stage and one pipeline for the `ld` and the `st`. $\mathtt{ld}_2$ waits for $\mathtt{ld}_1$ to free up the execution stage.

– Arrows 3 and 5: resource problem. There are only two entries d_1 and d_2 in the reservation station. We have to wait for one of the two to become free again.

– Arrow 6: the execution stage of $\mathtt{st}_2$ is busy. $\mathtt{ld}_3$ is delayed.



	01	02	03	04	05	06	07	08	$\boxed{9}$	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
ld$_1$ f4,0(r2)	I	E	E	E	E	E	E	E	E	W														
divf$_1$ f1,f4,f2		I	.	.	.	.	.	.	.	I	E	E	E	E	W	●								
stf$_1$ f1,64(r2)			I	.	.	.	.	.	.	.	.	.	.	.	I	E	W							
sub$_1$				⋆																				
bnz$_1$					⋆																			
ld$_2$ f4,0(r2)						I	.	.	I	E	W													
divf$_2$ f1,f4,f2							I	.	.	.	I	E	E	E	E	W	●							
stf$_2$ f1,64(r2)								I	.	.	.	.	.	.	.	I	E	W						
sub$_2$									⋆															
bnz$_2$										⋆														
ld$_3$ f4,0(r2)											I	E	W											
divf$_3$ f1,f4,f2																I	E	E	E	E	W			
stf$_3$ f1,64(r2)													I	.	.	.	.	.	.	.	I	E	W	
sub$_3$														⋆										
bnz$_3$															⋆									
ld$_4$ f4,0(r2)																I	I	E	W					
divf$_4$ f1,f4,f2																	I	I	I	E	E	E	E	W

Figure A1.16. *Execution of a loop*

Appendix B

Programming Models

A2.1. Instruction coding in the I8086

Abbreviation		Abbreviation	
reg	register	cond	condition code
reg8	8-bit register	sreg	segment register
reg16	register 16 bits	imm	immediate value
mem	memory address		

Abbreviation	
d	Direction (0 = memory to register, 1 = register to memory)
w	Indicator word (1)/byte (0)
s	Sign (indicator set to 1 for extension from 8 to 16 bits)
mod	Mode indicator: − 00 if r/m = 110: direct addressing, otherwise disp = 0 and there is an indirection. The operand addressing mode must be based, indexed or based-indexed. − 01 indirect access with disp in 8 bits − 10 indirect access with disp in 16 bits − 11 two register instruction (reg for the target and r/m for the source)

Abbrev.		Abbrev.		Abbrev.	
reg	Registers – 000 AX AL – 001 CX CL – 010 DX DL – 011 BX BL	reg	Registers – 100 SP AH – 101 BP CH – 110 SI DH – 111 DI BH	sreg	Segments – 00 ES – 01 CS – 10 SS – 11 DS

Abbreviation	
r/m	Register/memory (access mode). If the mod field is set to 11, r/m specifies the source register and reg the target. Otherwise, the addressing mode is coded by r/m as follows: – 000 DS:[BX+SI+disp] – 001 DS:[BX+DI+disp] – 010 SS:[BP+SI+disp] – 011 SS:[BP+DI+disp] – 100 DS:[SI+disp] – 101 DS:[DI+disp] – 110 DS:[BP+disp] – 111 DS:[BX+disp]
disp	Operand offset
data	Constants

EXAMPLE A2.1. (**Instruction coding**) –

Instructions	Coding
`mov al,[bx]`	`8A07`
`add ax,bx`	`01D8`

`mov es:[bx+di+a],ax`	`26`
	`89410A`

Hexa	code	d	w	mod	reg	r/m	disp
`8A07`	100010	1	0	00	000	111	
`01D8`	000000	0	1	11	011	000	
`26`	001001	1	0				
`89410A`	100010	0	1	01	000	001	0000 1010

`26` applies to the following instruction `mov` using `es` instead of `ds`.

A2.2. Instruction set of the DLX architecture

A2.2.1. *Operations on floating-point numbers*

addf, addd subf, subd	Single and double precision add and subtract
multf, multd divf, divd	Same as above for multiply and divide
cvtf2d, cvtf2i cvtd2f, cvtd2i cvti2f, cvti2d	Convert floating-point to double precision or integer Convert double precision to floating-point or integer Convert integer to floating-point or double precision

EXAMPLE A2.2. (**Operations on floating-point numbers)**

```
movi2fp f1,r2   ; Convert f1:=float(r2)
movi2fp f2,r3   ; Convert f2:=float(r3)
divf f3,f1,f2   ; Floating-point division f3:=f1/f2
movfp2i r1,f3   ; Convert r1:=long(f3)
```

A2.2.2. *Move operations*

lb, lbu, sb	Load byte, load byte unsigned, store byte
lh, lhu, sh	Same as above, but for 16-bit words
lw, sw	Load/store 32-bit words
lf, sf, ld, sd	Load/store single and double precision floating-point numbers
movi2s, movs2i	Transfer between general registers and control registers
movf, movd	Transfer between floating-point registers, in single or double precision (register pair)
movfp2i, movi2fp	Transfer between floating-point registers and integer registers (32 bits)
lhi	Load the 16 most significant bits of a register

A2.2.3. *Arithmetic and logic operations*

add, addi, addu, addui	Add implicit, immediate, signed and unsigned
sub, subi, subu, subui	Same as above, but for subtract
mult, multu, div, divu	Multiply and divide signed and unsigned
and, andi	Implicit and immediate AND
or, ori	Implicit and immediate OR
xop, xopi	Implicit and immediate exclusive OR
sll, srl, slli, srli	Shift left/right logical, implicit and immediate
sra, srai	Shift right arithmetic, implicit and immediate
s__, s__i	Sets to 1 or 0 based on lt, gt, le, ge, eq or ne condition

EXAMPLE A2.3.–

```
lhi r1,#-3    ; r1 loaded with FFFD0000(16)
srai r1,#1    ; r1 = FFFE8000(16)
lh r2,(r4)    ; Memory read r2:=MEM(r4)
add r2,r2,#1  ; r2:=r2+1
sw (r4),r2    ; Rewrite into memory
```

A2.2.4. *Branches*

beqz, bnez	Conditional branch
bfpt, bfpf	Test in the status register of the floating-point processor
j, jr	Direct or indirect unconditional branch
jal, jalr	Jump and link: direct or indirect unconditional branch with link
trap	Transfer to the operating system (vectored address)
rfe	Return from exception

EXAMPLE A2.4. (**Conditional branches**)

```
slti r1,r2,#3   ; If r2 < 3 then r1:=1 else r1:=0
beqz r1,next   ; If r1=0 then go to next
```

Direct addressing is written in the format `j tag`, `beqz rk,tag`, etc. The assembler calculates a shift (16 or 26 bits) relative to the following instruction and uses it in the instruction code. In the case of indirect addressing, the content of the register is used as the target address.

Branches with links cause the return address (PC+4) to be stored in register `r31`. The return from the subroutine call is then performed using `jr r31`.

Bibliography

[ABB 04] ABBOTT D., *PCI Bus Demystified*, 2nd ed., Butterworth-Heinemann Ltd, May 2004.

[ADV 87] ADVANCED MICRO DEVICES, *Am29000 Streamlined Instruction Processor*, 1987.

[AND 67] ANDERSON D.W., SPARACIO F.J., TOMASULO R M., "The IBM system/360 Model 91: machine philosophy and instruction-handling", *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 8–24, January 1967.

[ARC 86] ARCHIBALD J., BAER J-L., "Cache coherence protocol: evaluation using a multiprocessor simulation model", *ACM Transaction on Computer Systems*, vol. 4, no. 4, pp. 273–298, 1986.

[BAU 04] BAUKUS K., VAN DER MEYDEN R., "A knowledge based analysis of cache coherence", *Proceedings of 6th International Conference on Formal Engineering Methods*, ICFEM 2004, Seattle, Washington, WA, 8–12 November 2004

[ENC 08] ENCYCLOPÆDIA BRITANNICA, 2008.

[CEN 78] CENSIER M., FEAUTRIER P., "A new solution to coherence problems in multicache systems", *IEEE Transaction on Computers*, vol. C-27, pp. 1112–1118, 1978.

[CLA 85] CLARK D.W., EMER J.S., "Performance of the VAX-11/780 translation buffer: simulation and measurement", *ACM Transactions on Computer Systems*, vol. 1, pp. 31–62, 1985.

[COL 85] COLWELL R. P., HITCHCOCK III C.Y., JENSEN E.D., BRINKLEY SPRUNT H.M., KOLLAR C.P., "Instruction sets and beyond: computers, complexity, and controversy", *Computer*, vol. 18, no. 9, pp. 8–9, 1985.

[DEH 93] DEHNERT J.C., TOWLE R.A., "Compiling for the Cydra 5", *The Journal of Supercomputing*, vol. 7, nos. 1–2, pp. 181–227, 1993.

[DUB 98] DUBOIS M., SCHEURICH C., BRIGGS F.A., "Memory access buffering in multiprocessors", *25 Years of the International Symposia on Computer Architecture (selected papers)*, ISCA '98, ACM, New York, NY, pp. 320–328, June 1998.

[JOS 83] FISHER J.A., "Very long instruction word architectures and the ELI-512", *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 140–150, 1983.

[FRA 84] FRANCK S., INSELBERG A., "Synapse tightly coupled multiprocessor: a new approach to solve old problems", *Proceedings AFIPS '84 National Computer Conference*, pp. 41–50, July 1984.

[GOL 63] GOLDSTINE H.H., VON NEUMANN J., *Planning and Coding of Problems for an Electronic Computing Instrument*, Pergamon Press, 1963.

[GOO 10] GOOSSENS M., *The XeTeX Companion, TeX meets OpenType and Unicode*, CERN, January 2010.

[GOS 80] GOSLING, J. B., *Design of Arithmetic Units for Digital Computers*, Macmillan, 1980.

[HEN 96] HENNESSY J., PATTERSON D.A., *Computer Architecture: A Qualitative Approach*, Morgan Kaufman Publisher, 1996.

[HEN 11] HENNESSY J., PATTERSON D.A., *Computer Architecture: A Qualitative Approach*, 5th ed., Morgan Kaufman Publisher, October 2011.

[HIL 89] HILL M.D., SMITH J., "Evaluating associativity in CPU caches", *IEEE Transactions on Computers*, vol. 38, no. 12, pp.1612–1629, 1989.

[IBM 93] IBM and Motorola, *PowerPC 601 RISC Microprocessor User's Manual*, IBM Motorola, 1993.

[IEE 08] IEEE Computer Society, IEEE Standard for Floating-Point Arithmetic, IEEE Std 754™-2008, Approved 12 June 2008, IEEE-SA Standards Board, June 2008.

[INT 79] Intel Corp., MCS-80/85tm Family User's Manual, 1979.

[INT 85] Intel Corp., High Performance Microprocessor with Integrated Memory Management, 1985.

[INT 97] Intel Corp., Intel Architecture Optimization Manual, 1997.

[INT 09] Intel Corp., Intel Processor Identification and the CPUID Instruction, Application Note 485, August 2009.

[INT 10] Intel Corp., Intel Advanced Encryption Standard Instruction Set, rev. 3.0. White Paper, January 2010.

[JOU 98] JOUPPI, N. P., "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *25 Years ISCA: Retrospectives and Reprints*, pp. 388–397, 1998, available at http://doi.acm.org/10.1145/285930.285998.

[KAT 85] KATZ R.H., EGGERS S.J., WOOD D.A., PERKINGS C.L., AND SHELDON R.J., "Implementing a cache consistency protocol", *SIGARCH Computer Architecture News*, vol. 13, no. 3. pp. 276–283, 1985.

[KIL 62] KILBURN T., EDWARDS D.B.G., LANIGAN M.J., SUMNER F.H., "One level storage system", *IRE Actions on Electronic Computers*, vol. EC-11 vol. 2, pp. 223–235, 1962.

[LAI 11] Laine S., Karras T., "High-performance software rasterization on GPUs", *Proceedings of the ACM SIGGRAPGH Symposium on High Performance Graphics, HPG '11*, ACM, New York, NY, pp. 79–88, 2011. DOI: 10.1145/2018323.2018337.

[LAM 79] Lamport L., "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Transactions Computers*, vol. C-28, no. 9, pp. 690–691, 1979.

[LAU 83] Laurent E., *La puce et les Géants : de la révolution informatique à la guerre du renseignement*, Fayard, 1983.

[LAV 75] Lavington S., *A History of Manchester Computer*, NCC Publications, 1975.

[LEE 84] Lee J.K.F., Smith A. J., "Branch prediction strategies and branch target buffer design", *IEEE Computer*, vol. 17, pp. 6–22, 1984.

[MAU 79] Mauchly J.W., "Amending the ENIAC story", *Datamation*, vol. 25, no. 11, pp. 217–220, October 1979.

[MCM 98] Mcmillan K.L., *Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking*, Springer-Verlag, pp. 110–121, 1998.

[MEL 88] Melvin S.W., Shebanow M.C., Patt Y.N., "Hardware support for large atomic units in dynamically scheduled machines", *Proceedings of the 21st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 60–63, 1988, available at http://doi.acm.org/10.1145/62504.62535.

[MEN 42] Menabrea L.F., "Sketch of the analytical engine invented by Charles Babbage", *Bibliothèque Universelle de Genève*, no. 82, October 1842.

[MIL 96] Milo T., Milutinovic V., "The word-invalidate cache coherence protocol", *Microprocessors and Microsystems*, vol. 20, no. 1, pp. 3–16, 1996.

[MOR 81] Moreau R., *Ainsi naquit l'informatique : les hommes, les matériels à l'origine des concepts de l'informatique d'aujourd'hui*, Dunod, 1981.

[NAT 84] National Semiconductor Corp, NS-32081 Floating-Point Unit, 1984.

[PAT 99] Patel S.J., Trace cache design for wide-issue superscalar processors, PhD Thesis, University of Michigan, 1999.

[PRE 91] Prete C.A., "RST cache memory design for a tightly coupled multiprocessor system", *lEEE Micro*, vol. 11, no. 2, 1991.

[ROS 69] Rosen S., "Electronic computers: a historical survey", *Computer Survey*, vol. 1, pp. 7–36, 1969.

[ROT 96] Rotenberg E., Bennett S., Smith J.E., Rotenberg E., "Trace cache: a low latency approach to high bandwidth instruction fetching", *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, pp. 24–34, December 1996.

[RUD 84] Rudolph L., Segall Z., "Dynamic decentralized cache schemes for MlMD parallel processors", *Proceedings of the 11th ISCA*, pp. 340–347, 1984, available at http://doi.acm.org/10.1145/800015.808203.

[SAI 96] Sailer P.M., Kaeli D.R., *The DLX Instruction Set Architecture Handbook*, Morgan Kaufmann, 1996.

[SCI 96] SCIENCE ET VIE, Qui a inventé l'ordinateur ? 1996.

[SIM 88] SIMMONS W.W., ELSBERRY R.B., *Inside IBM: The Watson Years: A Personal Memoir*, Dorrance, 1988.

[SMI 81] SMITH J.E., "A study of branch prediction strategies", *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA'8, IEEE Computer Society Press, Los Alamitos, CA, pp. 135–148, 1981, available at http://dl.acm.org/citation.cfm?id=800052.801871.

[STO 90] STONE H.H., *High-Performance Computer Architecture*, 2nd ed., Addison Wesley Longman Publishing Co., Inc., Boston, MA, 1990.

[SUM 62] SUMNER F.H., HALEY G., CHEN E.C.Y., "The central control unit of the 'Atlas' computer", *Proceedings of IFIP Congress*, 1962.

[TAN 76] TANG C.K., "Cache design in the tightly coupled multiprocessor systems", *AFIPS Conference Proceedings, National Computer Conference*, ACM, New York, NY, pp. 749–753, June 1976. DOI: 10.1145/1499799.1499901.

[TEX 89] TEXAS INSTRUMENTS, Third-Generation TMS 320 User's Guide, 1989.

[TEX 10] TEXAS INSTRUMENTS, TMS320C62x DSP CPU and Instruction Set Reference Guide, May 2010.

[THA 87] THACKER C.P., STAWART L.C., "Firefly: A multiprocessor workstation", *Proceedings of the 2nd International Conference Architectural Support for Programming Languages and Operating Systems*, pp. 164–172, 1987, available at http://doi.acm.org/10.1145/36206.36199.

[THA 88] THACKER C.P., STEWART L.C., SATTERTHWAITE E.H., "Firefly: a multiprocessor workstation", *IEEE Transactions on Computer*s, vol. 37, no. 8, pp. 909–920, 1988.

[THO 63] THORNTON J.E., *Considerations in Computer Design – Leading up to the Control Data*, Control Data Corporation, Chippewa, IL, 6600, 1963.

[THO 70] THORNTON J.E., *Design of a Computer – The Control Data 6600*, Scott, Foresman and Company, 1970.

[VON 45] VON NEUMANN J., "First draft of a report on the EDVAC", PhD Thesis, Moore School of Electrical Engineering, Pensylvania, June 1945.

[WIL 51] WILKES M.V., "The best way to design an automatic computing machine", *Manchester University Computer, Inaugural Conference*, pp. 16–18, July 1951. Reprinted in *Annals of the History of Computing*, vol. 8, pp. 118–121, 1986, and in *The Early British Computer Conferences*, WILLIAMS M.R., CAMPBELL-KELLY M. (eds), Charles Babbage Institute Reprint Series for the History of Computing, vol. 14, MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, pp. 182–184, 1989.

[WIL 65] WILKES M.V., "Slave memories and dynamic storage allocation", *IEEE Transactions on Electronic Computers*, vol. EC-14, pp 270–271, April 1965.

[WIL 53] WILKES M.V., STRINGER J.B., "Microprogramming and the design of the control circuits in electronic digital computers", *Proceedings of Cambridge Philosophical Society*, 1953.

[YEH 91] YEH T.Y., PATT Y.N., "Two-level adaptive training branch prediction", *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp 51–61, 1991.

Index