Course Notes

High Performance Computing

Marco Vanneschi

Department of Computer Science, University of Pisa

Background

Structured Computer Architecture

The goal of this Part is to study the main concepts and techniques in computing architecture according to a uniform and structured approach.

A "uniform" approach is necessary for at least two reasons: *i)* in order to clearly dominate the interrelations between the vast amount of concepts and technologies, and *ii)* to put the students, having different and/or incomplete background, in the same condition to attend and to study this course.

The "structured" approach means that all the techniques and technologies of computer architecture and high performance computing are studied in the context of a sound conceptual framework, able to cover both current and future systems and technologies.

The student must be aware of the way in which this Part has to be utilized. Because we cannot replicate an entire Computer Architecture course at the Bachelor level, during the initial lectures we'll review only some major concepts and techniques, which are fundamental to fully understand the issues studied in Part 1 and in Part 2 about high performance programs and high performance architectures, respectively. A detailed treatment will not be provided during the initial lectures. The student is invited to use the Sections of the current Part in order to be (to become) able of applying the concepts and techniques needed in Part 1 and Part 2, and to fill any possible gap whenever it is necessary or for personal culture.

Contents

1. Structuring by levels and processing modules 4

1.1 Vertical structuring by interpretation levels 4

1.1.1 Interpretation and compilation 6

1.1.2 Typical levels 7

1.2 Process level 8

1.2.1 Run-time support of processes 9

1.2.2 Compilation of applications by collection of processes 10

1.3 Assembler level 11

1.3.1 A first example 12

1.3.2 RISC *vs* CISC: a first view 13

1.3.3 Hierarchical levels and "flattening" of executable code 14

1.4 Firmware level 14

1.4.1 A firmware system as a collection of processing units 15

1.4.2 Firmware architecture of general-purpose computers 17

1.4.3 On the existence of the assembler machine 18

1.5 Horizontal structuring by processing modules 19

1.6 Performance evaluation: abstract machines and cost models. 21

2. The firmware level 24
2.1 Hardware level: logic components for PC and PO implementation 25
2.1.1 Combinatorial components 25
2.1.2 Registers and memory components 27
2.2 Microprograms and clock cycle 29
2.2.1 Microinstructions 30
2.2.2 Clock cycle 31
2.3 Processing unit design method through an example 33
2.3.1 Specification and microprogram 33
2.3.2 Operating Part and Control Part 34
2.3.3 Clock cycle 35
2.3.4 Performance evaluation 36
2.4 Communication at the firmware level 37
2.4.1 Communication channels and links 37
2.4.2 Asynchronous communication on dedicated links 39
2.4.3 Other communication forms on dedicated links 43
2.4.4 Synchronous communication on buses 45
2.4.5 Interconnection solutions: dedicated *vs* shared links 46
2.5 Modular memory organization 48
2.6 Exercises 49

3. The assembler machine and its basic interpreter 50
3.1 Logical address space and virtual memory 50
3.2 The assembler machine 53
3.2.1 Instructions and variables 53
3.2.2 Addressing modes 54
3.3 D-RISC: a Didactic RISC assembler machine 56
3.3.1 Instruction set 56
3.3.2 Compilation rules 59
3.3.3 Procedures 61
3.4 An elementary processor for D-RISC 63
3.5 Performance evaluation 67
3.6 Input/output 69
3.6.1 Data transfers 70
3.6.2 Synchronization and events: interrupts 71
3.7 Interrupt and exception handling 72
3.8 Exercises 75

4. Processes and virtual memory 76
4.1 Data structures for process run-time support 76
4.2 From process creation to process execution 77
4.3 Memory hierarchies and virtual memory 79
4.3.1 Introduction to memory hierarchies 80
4.3.2 Virtual memory: address translation 81
4.3.3 Memory Management Unit 82
4.3.4 Page fault exception 84
4.4 Virtual memory and shared objects 84

4.5 Shared pointers 85

5. Memory hierarchies and cache architecture 89

5.1 Memory hierarchies 89
5.1.1 Locality and reuse 89
5.1.2 Optimal page size 91
5.1.3 Working set 92
5.1.4 Paging on-demand *vs* prefetching 93
5.2 Cache memory 93
5.2.1 Treatment of writing operations 94
5.2.2 Address translation methods 96
5.2.3 Cache optimizations and annotations at assembler level 98
5.2.4 Performance evaluation 99
5.2.5 Block transfer and memory hierarchy organization 101
5.2.6 Write-Through evaluation 103
5.3 Exercises 104

6. Interprocess communication and its run-time support 105

6.1 Concurrent language definition 105
6.1.1 Structure of parallel programs 105
6.1.2 Typed communication channels 106
6.1.3 Communication forms 107
6.1.4 Non-determinism control in communications 108
6.2 Interprocess communication run-time support 110
6.2.1 Process support and shared data structures 110
6.2.2 “Generic” implementation 111
6.2.3 Direct copy into the target variable in asynchronous communications with suspended receiver 112
6.2.4 Synchronous communication: “peer” implementation 113
6.2.5 “Zero-copy” communication 114
6.3 Communication latency 116
6.4 Design issues for interprocess communication run-time support 117

1. Structuring by levels and processing modules

To describe a computing system we are used to distinguish implementation *levels*, and at each level we recognize the functional structure of the system as composed of processing *modules*. The very rough distinction between "hardware" and "software" is misleading and quite inadequate to understand the real features needed by a system designer or by an application designer, or even by an advanced user.

The intuitive meaning of level structuring is to distinguish between system views which are relatively "abstract", i.e. closer to the application level, or relatively "concrete", i.e. closer to the physical architecture including the operating system. However, even this approach is inadequate, because it hides the distinction between those levels that can be formally recognized and those levels which are just a matter of convenience for the designer or the user.

For example, assume that we are developing a complex application using the C language. Often it is useful to adopt a modular development methodology, by implementing a first set S1 of components which, in turn, are used as building blocks, with well-defined interfaces, to implement another set S2 of components. The designer could think that his/her application is structured in two levels, where S1 is the lowest and S2 the highest one. In this case we can speak about *functional levels*, however they do not correspond to real system levels. In fact, both S1 and S2, which are written in the same language, belong to the same system level. Structuring by functional levels, though important for modularity reasons, is just a matter of convenience for the application development or even for documentation. The same is true by iterating the previous reasoning, i.e. by implementing, at the same system level, further functional levels on top of S1and S2.

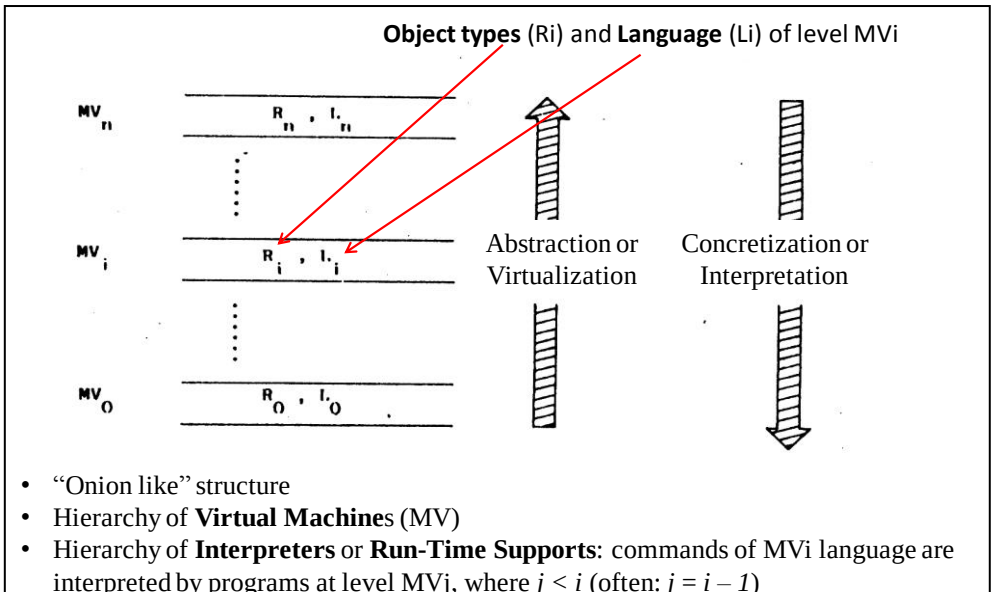
As another example, consider an application that has been described and tested in C. Traditionally, the final product, that is rendered available to the users, is a piece of code (source code and/or binary executable code) with suitable external interfaces. An alternative version is a "hardware" unit, or a set of units, whose behavior is equivalent to the "software" version. Conceptually, in the "hardware" version the C description has been translated (automatically or manually) into a proper formalism, which is a binary executable code though different from the "software" version. In this example the distinction between a "software" implementation and a "hardware" one is quite inessential: we have two different implementations which are at the same level, but use a *different support* (hopefully, with different performance values).

From both the previous examples, we recognize the need for a formal definition of the following concepts:

1. *vertical structuring* through system levels, which will be called **interpretation levels**,
2. *horizontal structuring*: parts that compose a system at a given level, which will be called **processing modules** at that level, as well as their *interaction* and *cooperation*.

1.1 Vertical structuring by interpretation levels

The functionalities of a system can be organized into a *hierarchy* of *interpretation levels*, or *virtual machines*, as shown in the following figure:



- "Onion like" structure
- Hierarchy of **Virtual Machine**s (MV)
- Hierarchy of **Interpreters** or **Run-Time Supports**: commands of MVi language are interpreted by programs at level MVj, where $j < i$ (often: $j = i - 1$)

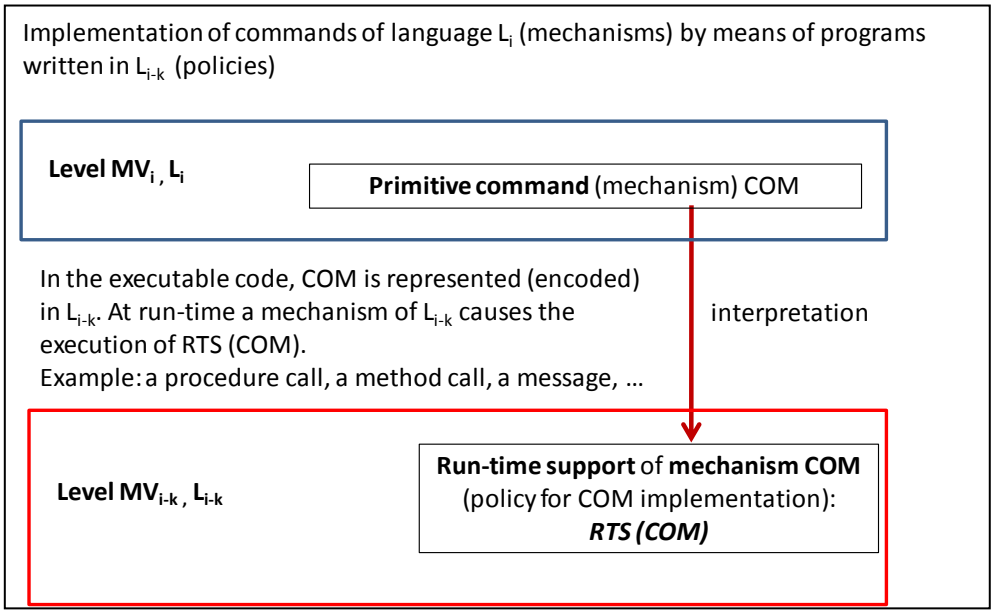
The **hierarchical structure** (like an onion) is such that at each level there is no visibility of the innermost (i.e. lower) levels.

Each level is characterized by its own view of the system through a *programming language*, thus through a proper definition of the *object types* that can be manipulated by such language. Distinct levels have distinct languages. It is important to realize that this **language-based approach** is valid at every level, not only at the highest ones (traditionally associated to the applications), but also at the lowest ones (traditionally associated to the computer architecture).

The hierarchical structure and the language-based approach lead to the following fundamental property of the level structuring:

- the commands (or *mechanisms*) of level MV_i are *interpreted* by programs (or *policies*) written at level MV_j, with $j < i$. Often $j = i - 1$, but this is not mandatory.

If COM is a command of language L_i of MV_i, the interpreter, or **run-time support**, of *COM* will be denoted by *RTS(COM)*.



The meaning of interpretation, or run-time support, is the classical one in programming language fundamentals: the execution of command *COM* consists in the execution of *RTS(COM)*. Better, as

illustrated in the figure, if *RTS(COM)* is at level MV_{i-k}, at run-time a mechanism of L_{i-k} causes the execution of *RTS(COM)*. This **linking** mechanism depends on the characteristics of L_{i-k}, for example it could be a procedure call, or a message, or even a code expansion (*RTS(COM)* is replicated in every place in which *COM* has to be executed).

1.1.1 Interpretation and compilation

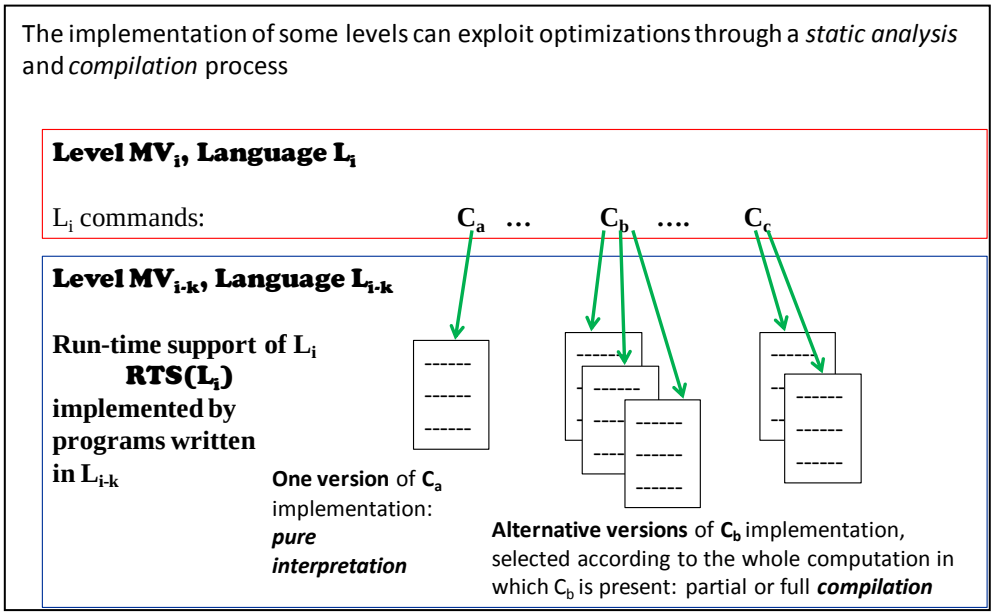
The definition of level structuring in terms of the interpretation concept does not exclude that **compilation** is adopted as a language translation technique. In fact, *compilation is a central issue in structured computer architecture*, as far as *code optimizations* can be recognized by a *static analysis*.

It should be clear that interpretation is always applied. We are interested in recognizing where, when and how compilation can be applied too.

It is useful to point out that:

- in *pure interpretation* exactly the same code *RTS(COM)* is executed each time *COM* is encountered;
- in a *compiled approach* the whole computation, or a suitable part of it, is analyzed statically in order to detect optimization depending on the specific utilization of the commands. Thus, *more than one version of RTS(COM)* exists for the same command COM, and the compiler selects one of them according to the specific utilization of COM in the computation.

The following figure explains typical situations in the implementation of a language:

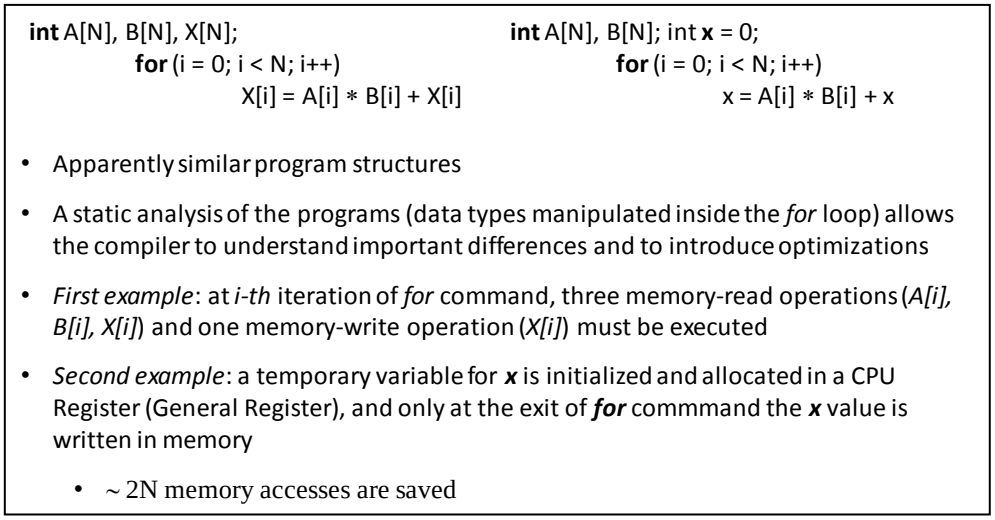


There are commands (C_a in the figure) in which only one version of the run-time support exists, thus the compiler inserts this version (a link to this version) in the executable code each time C_a is met: it is a case of *pure interpretation*. For other commands (C_b, C_c in the figure) more versions of their run-time support are available to the compiler, which select one of them according to the way the command is used in the whole computation (*partial or full compilation*).

For C_a it is possible that, according to the way the command is used (e.g. according to the values of some data) the execution of *RTS(C_a)* is far from the optimal behaviour, *unless* the interpreter is designed in such a way that proper distinctions can be performed at run-time. Instead, for C_b the compiler is able to statically recognize different utilizations and, for each utilization, to select "the

best" *RTS(C_b)* version in order to optimize the execution, e.g. according to the values of some data if statically foreseen, or according to the form of the piece of code containing C_b.

The following figure shows a very simple, yet meaningful, example of alternative versions of run-time support that are recognized by a compilation-oriented language (e.g., a C-like language). In this case, the optimization concerns the processing time according to the memory hierarchy exploitation, i.e. trying to minimize the number of accesses to the main memory.

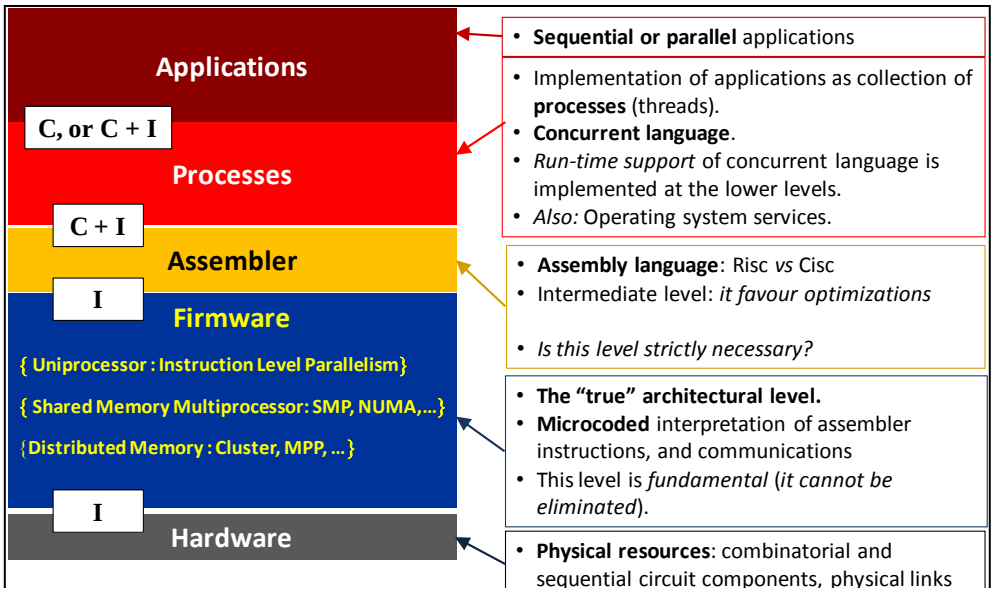


In the previous distinction between compilation-based and interpretation-based approaches, we stressed the case in which code optimizations, that depend on the way *COM* is used in the context of the whole computation, are statically recognized in the compiled approach. However, though more difficult, *optimizations are possible in the pure interpretation approach too*: this means that (as said before for C_a) the code of *RTS(COM)* contains several alternative branches according to specific conditions that are *tested or monitored at run-time, when/where possible*.

This distinction between ***static* vs *dynamic optimizations*** is a central issue in computer architecture. A notable example is represented by optimizations in pipelined scalar /superscalar/ multithreaded CPUs. In order to eliminate delays and performance bottlenecks during the program execution, in some machines the binary assembler code is properly restructured at compile-time according to specific features of the firmware architecture, while in other machines restructuring is directly delegated to the firmware interpreter with no/few actions at compile-time. The latter case is characterized by *out-of-order* execution of instructions, while in the former case *in-order* execution is basically adopted. Out-of-ordering implies a much more complex CPU structure in terms of hardware components, firmware algorithms, and chip area. On the other hand, if optimizations are delegated to the firmware interpreter, then binary compatibility can be achieved between computers with the same assembler machine but different firmware architectures (e.g. two different Intel/AMD processors with the same x86 assembler machine).

1.1.2 Typical levels

The following figure shows five typical virtual machines (some of them could also be futher decomposed) in a general-purpose system. Symbols "C" and "I" at the interfaces between levels denotes "Compilation" and "Interpretation" respectively:



At the highest level we have languages to develop sequential applications, like C, C++and Java. We are interested also in high-level formalisms to develop *parallel applications*, as it will be studied in Part 1. Applications are compiled into, and/or interpreted by, the lowest levels according to the features of specific application languages.

Sections 1.2, 1.3, 1.4 contain an overview of the main characteristics of the other levels and their relationships, which are quite fundamental issues for understanding principles and technologies of structured computer architecture. A detailed treatment will be the subject of Sections 2, 3, 4, 5, 6.

1.2 Process level

This level is quite fundamental for any system. Here we express computation by means of **processes**, i.e. *entities that are able to be executed concurrently and to cooperate*.

Concurrency is a general term, denoting that in a computation we can establish a partial order of execution between activities (called processes or threads): at any time instant some activities are independent and ready for execution, while others are waiting for events generated by the currently ready activities. In a parallel architecture, concurrency can be exploited by *real parallelism* of execution, where the maximum number of ready activities in the execution phase (running phase) is equal to the number of processing nodes (processors). In a uniprocessor architecture, parallelism is simulated, i.e. at most one of the ready activities is in the execution phase.

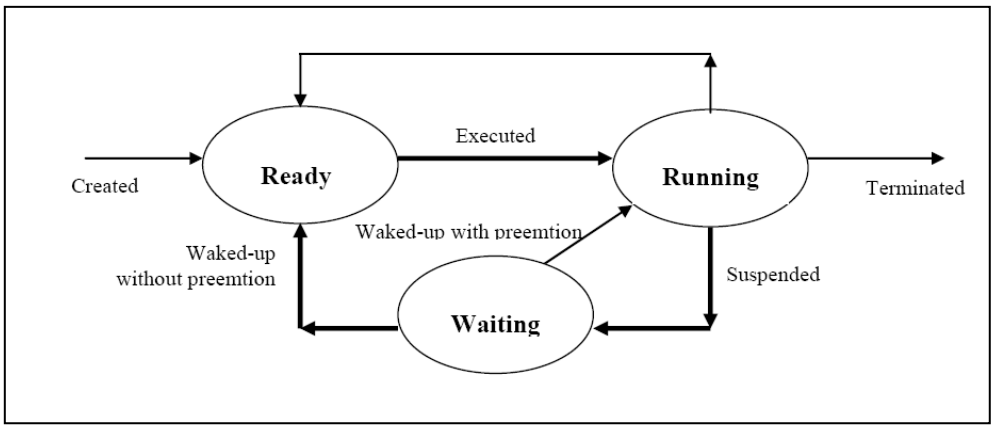
At the process level we must have *a concurrent programming language* in order to express concurrent/parallel computations. This issue will be studied in Section 6. In some cases a real, new language is defined, whose commands provide the concepts of process, parallelism, ordering, scheduling, and cooperation between processes. In many other cases, computations at the process level are expressed through sequential languages (e.g., C) "instrumented" by a collection of *libraries* to express parallelism and cooperation concepts. For example, MPI is a library to express cooperation via message-passing, while in OpenMP cooperation is allowed through shared variables.

1.2.1 Run-time support of processes

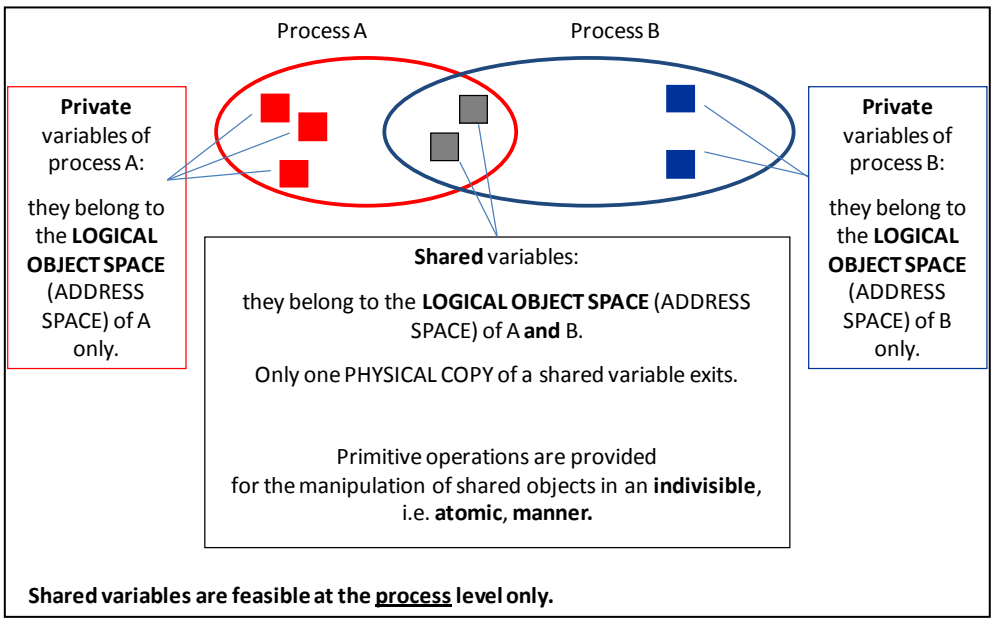
The run-time support of the concurrent language, i.e. the run-time support of processes, includes the implementation of:

- low-level scheduling, i.e. process phases and contexts, according to the specific assembler-firmware architecture,
- cooperation mechanisms (based on message-passing, or shared variable, or both)

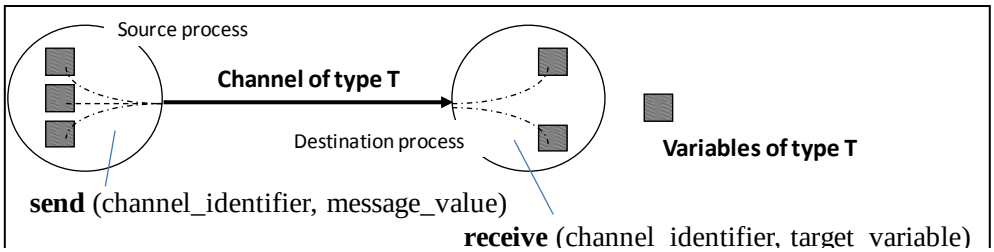
In the following figure, the typical *low-level scheduling* of process phases is shown in the form of a graph of states and state transitions:



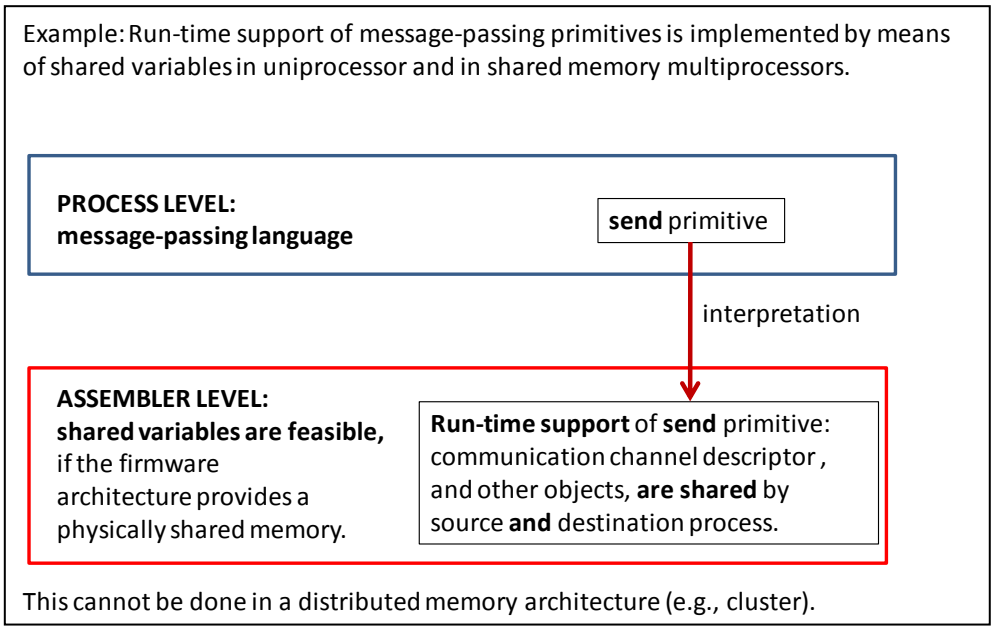
The following figure illustrates the concept of shared objects, which can exists at the run-time support level for a uniprocessor or for a shared-memory multiprocessor, or that can be defined at the concurrent language level itself:



Conceptually, processes cooperating by message-passing are characterized by communication mechanisms with the following features:



As shown in the figure, two processes cooperating by message-passing do not share any variable: they operate on local variables only, and their cooperation is based upon the explicit exchange of values. The *run-time support* of message-passing mechanisms can be based on shared variables if the underlying architecture provides the shared variable concept in a primitive manner, as in a uniprocessor or in a shared memory multiprocessor:



This example is meaningful to exemplify how the vertical structuring is characterized by the ***separation of concerns*** principle: in this case, the concurrent language at the process level can be defined with local variables only (as in a message-passing language), while its run-time support can be implemented by shared-variable mechanisms if this is provided, and it is more efficient, at the level at which such implementation is done.

1.2.2 Compilation of applications by collection of processes

As any other level, the process level can be used *per se*, that is a user can be interested in designing computations using the concurrent language directly.

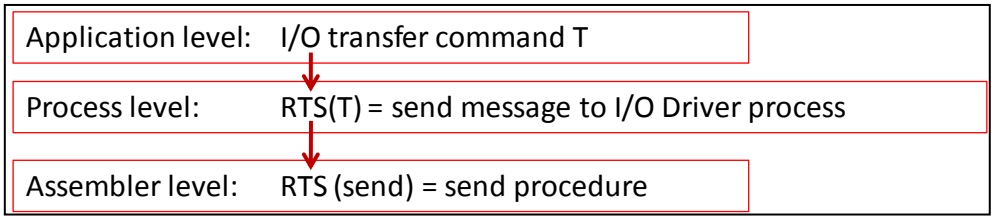
In any case, in a hierarchical structuring, the process level is used to implement *the run-time support of programs at the Application level*, both for sequential and for high-level parallel programs. In other terms, any program at the Application level is expressed as a *collection of cooperating processes*, some of which correspond to the functionalities the programmer wanted to express, while others could be invisible at the programmer but equally exist in order to provide the run-time support of the application program.

For example, consider a sequential application APPL. It is compiled into a process, let us call it APPL_P. APPL_P is not merely a translation of APPL into an executable language, instead it

contains policies that render it possible the cooperation with others processes implementing *services* required by the application explicitly or implicitly.

This linking and cooperation with services belongs to the run-time support of APPL. Its goal is to exploit the whole set of system resources in a seamless and efficient manner, notably: exception handling, memory management, file management, I/O transfers, networking, and others. For this purpose, the compilation of APPL into APPL_P adds to the programmer-visible functionalities of APPL some mechanisms providing the linking and the cooperation with the service processes.

For example, if APPL contains a high-level input/output transfer command, APPL_P could contain a sequence of send-message and receive-message commands of the concurrent language in order to provide the cooperation with the I/O Driver service process, which is in charge of managing the I/O transfer by proper interactions with other service processes and the I/O units:



As we can see, in general the transformation from the application level into the process level is a *combination of compilation and interpretation*. In the example, compilation consists in recognizing an I/O command and in replacing it with one or more message-passing primitives with proper parameters. The interpretation prevails over compilation if compilation is limited to a command substitution with the same library for each occurrence of the same command, i.e. if there is no difference between the run-time support of this instantiation of the message-passing primitives wrt other utilizations of the same primitives. However, better optimizations could be introduced: for example, different versions of the message-passing libraries exist, and the compiler can select "the best one" according to the kind of I/O transfer and to the implementation of the Driver process.

As a simple example, assume that two versions of the send-message primitive exist: one imposes a fixed message length L, while in the other any message length can be specified as a parameter. If messages have length L (i.e. L words have to be transferred through the I/O subsystem), the implementation of the first version is more efficient, since some controls are avoided in the run-time support code. However, if the actual message length is greater than L, just one primitive of the second version is probably more efficient than a sequence of send primitives of the first type. As another example, if the compilation consists in a request-reply interaction with the Driver, i.e. APPL_P waits for the reply after the request, then a request-reply primitive could be more efficient than a pair send-receive.

Several differences and optimizations in the message-passing run-time support will be studied during the course, e.g. depending on the architecture class and/or on the application class.

1.3 Assembler level

An application, in the form of a process containing proper linking and cooperation mechanisms, is translated into an assembler language code. The final *executable version* of the application is the *binary* representation of this code. As said, the implementation of processes run-time support by the assembler machine may be a pure compilation or, at least partially, a mix of compilation and interpretation.

Almost every sequential or parallel architecture is realized according to the so called *Von Neumann* model of stored-program computer, according to which instructions and data are allocated in the

same memory support and the assembler language has an *imperative* semantics (in practice, variables and sequence controls are first-class citizens). The very relevant pros and cons of this choice will be discussed in successive parts of the course from the conceptual and from the technological point of view.

1.3.1 A first example

The assembler machine is characterized by a rather low-level language, if compared with the application languages we are used to. Typically, it contains instructions operating on memory locations and/or processor registers to perform arithmetic-logic operations, or data transfer, or sequencing control actions (branches, jumps), plus other mechanisms which are specific of a certain architecture (e.g. interrupt control, context-switching support, cache management, and others). For example, the piece of code discussed previously:

```
int A[N], B[N]; int x = 0;
        for (i = 0; i < N; i++)
                x = A[i]*B[i] + x
```

could be compiled into an assembler code of this kind (for the moment being, an informal syntax is sufficient):

```
// a processor general register Rx is used as a temporary variable for x, and it is initialized to zero //
// two general registers RA, RB are initialized at the base addresses of arrays A and B respectively //
// a general register Ri contains the value of index i, and it is initialized to zero //
// a general register RN contains the value of N //
LOOP:  LOAD   RA, Ri, Ra      // copy the content of the memory location, whose address is given by RA
                              + Ri (more precisely: the sum of the contents of registers RA and Ri), into
                              the temporary register Ra //
       LOAD   RB, Ri, Rb      //copy the content of the memory location, whose address is given by RB +
                              Ri, into the temporary register Rb //
       MUL    Ra, Rb, Ra     // execute the integer multiplication of Ra and Rb contents, and store the
                              result into Ra //
       ADD    Ra, Rx, Rx     // add the contents of Ra and Rx and store the result into Rx //
       INCR   Ri             // increment the content of Ri //
       IF <   Ri, RN, LOOP   // if the content of Ri is less than the content of RN, then go to instruction
                              with label LOOP, otherwise continue in sequence //
```

Let us assume that the adopted assembler machine has an instruction set of 256 distinct instructions and 64 general registers. Thus, each instruction is encoded into a 32-bit word, with the following fields: a 8-bit operation code, one or more 6-bit register addresses, constants represented by 12 bits (example, the LOOP label in the conditional branch instruction IF) or more, as well as fields with special meaning.

This is just an example. Several classes of assembler machines exist in general and will be discussed in Section 3.

As indicated in the general figure of the systems levels, *in principle the assembler machine is not strictly necessary*, and the execution of processes could be implemented at the firmware level directly. However, in practice the existence of the assembler machine is very useful in order to

provide an efficient representation of processes. In other words, assembler can *an efficient intermediate language* for the process compilation.

This issue will be discussed in Section 1.4.3. For sake of the present Section, it is important to understand that the main purpose of the assembler machine is to allow the compiler to represent programs in a way which is suitable

i. for the underlying architecture, and

ii. for introducing optimizations.

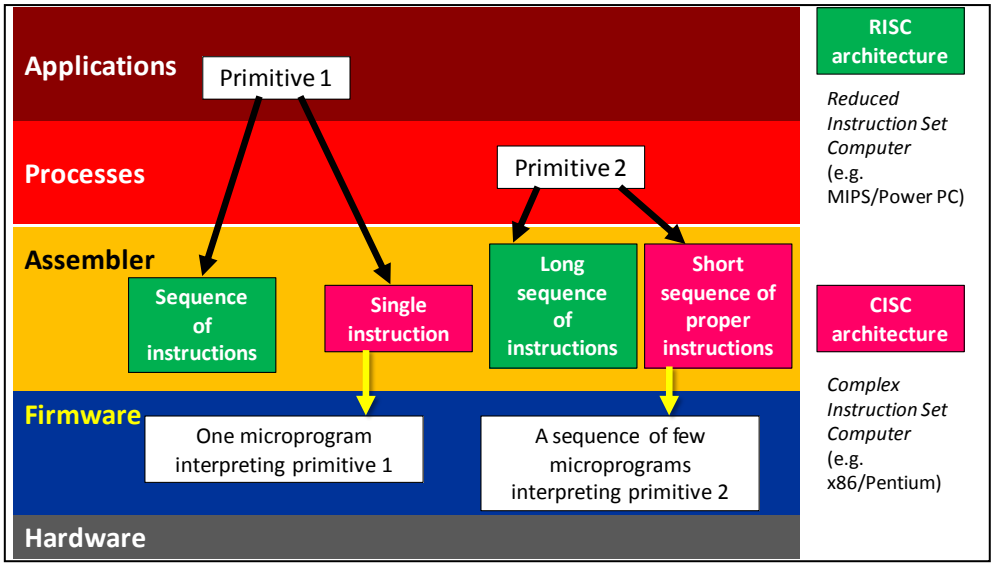
In fact the feasibility of introducing optimizations, according to the specific architecture, is the main focus of the assembler machine definition.

1.3.2 RISC *vs* CISC: a first view

In a *Reduced Instruction Set Computer* (RISC) emphasis is placed onto simplicity of operations, primitive data types, and addressing modes. Such instructions are much simpler than, and "quite far from", the commands of the application languages. Instructions are of fixed length (one word) as in the previous example, rendering easier their decoding and execution. The goal is to reduce the complexity and the clock cycle duration of the processor units and, at the same time, to simplify the design of intensive compile-time optimizations.

In a *Complex Instruction Set Computer* (CISC) the goal is "to reduce the distance" between the application languages and the assembler language. In fact, there are some high-level language constructs that correspond to very short sequences of assembler instructions, or even to a single instruction. The higher complexity concerns not only the arithmetic operations and sequencing control instructions, but also the primitive data types and the memory addressing modes. Usually, instructions are of variable length. In principle, all these features contribute to reduce the compiler work, and at the same time to increase the complexity of the processor units.

The following figure illustrates some basic features of RISC *vs* CISC:

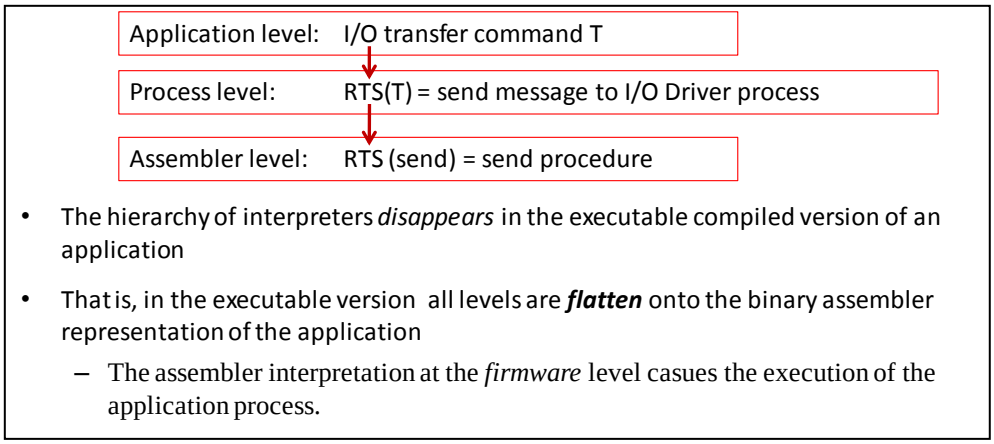


At the first sight, RISC is more compilation-oriented, while CISC is more interpretation-oriented. Thus, in RISC complexity is concentrated in the compiler, while in CISC it is concentrated in the firmware architecture. Not surprisingly, quite all CISC machines adopt out-of-ordering, while some RISC machines, especially of the most recent multi-core generations, adopt an in-order firmware architecture. However, things are not so simple, and the accurate analysis of application

optimizations and of computer architecture, studied in the various parts of the course, will reveal a much more complex reality.

1.3.3 Hierarchical levels and "flattening" of executable code

Let as consider again the example of the run-time support for linking a service into an application:



As indicated in the figure, the hierarchy of interpreters *per se* does not introduce some *overhead* in the process execution. The execution mechanism is not a sequence of calls from one level to the other, instead all levels are *flatten* into the final binary version of the executable code.

In other terms, the compilation of the application language replaces the application level command (in the example, T) with the executable version of the send-message library. The sequence of translations (from the application language into the concurrent language, then from the concurrent language into the assembler language, then from the assembler language into the binary assembler version) exists in principle: in fact some systems adopts a similar strategy, while others merge the translation steps, however this is not essential for our purposes.

It is important to realize that the flattening into the final executable version of the application is not in contrast with the concept of hierarchical levels, instead it is exactly consistent with this concept and with the way in which levels are implemented.

The student is invited to understand this issue very clearly.

Of course, these considerations are relative to systems designed according to well-structured methodologies and by means of proper tools. Not necessarily this is true in the commercial world.

Moreover, the student is invited to reflect upon the difference between the interpretation level structuring and apparently similar approaches to system structuring. A notable example is represented by the ISO/OSI levels of networking protocols: this is not an interpretation level structuring as defined in this Section, and this is not without consequences. For example, why there is so high overheads in TCP/IP communications compared to the communication latency of the "bare" network?

1.4 Firmware level

In the typical vertical structuring, the firmware level provides the *interpretation* (pure interpretation) of the executable version of process instructions, which in a general purpose computer are usually instructions of the assembler machine. From this point of view, the firmware

language (*microlanguage*) is the real machine language. It is at the firmware level that we have the typical vision of a computer architecture, for example:

- a uniprocessor computer composed of a CPU, a main memory, and I/O subsystem,
- a shared memory multiprocessor, including several CPUs able to address the same physical main memory,
- a distributed memory multicomputer, for example a cluster of networked PCs or workstations,
- and so on.

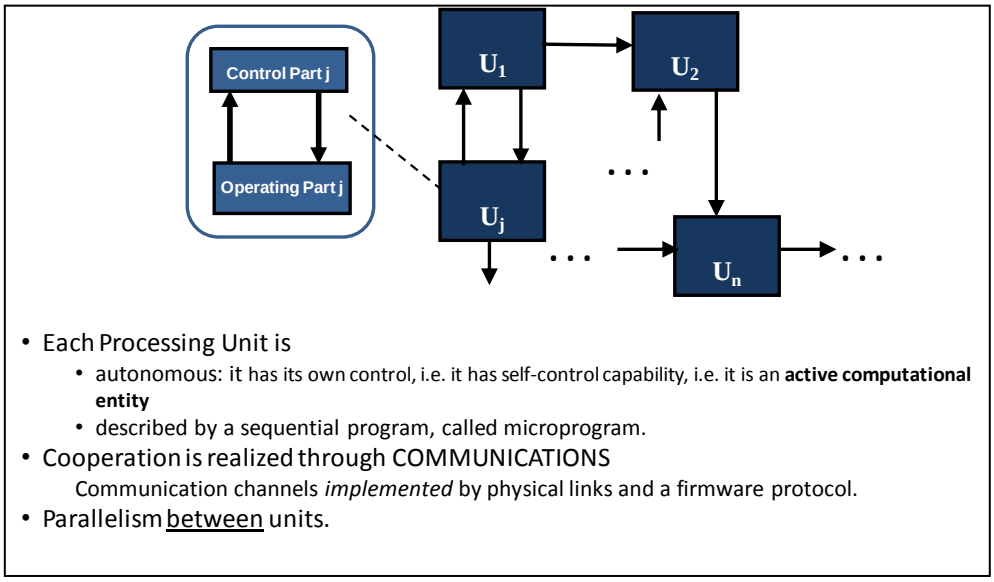
Moreover, it is at the firmware level that we have a detailed view of the main features of a certain computer architecture. For example,

- the existence of a memory hierarchy (cache levels),
- the parallel-pipelined execution of assembler instructions,
- the interconnection structures between CPU(s), memory levels, I/O units,
- and so on.

1.4.1 A firmware system as a collection of processing units

A formal theory exists, according to which we are able to study any system at the firmware level in a systematic manner.

The system is viewed as a collection of cooperating **processing units**:



A processing unit (simply, a unit) is defined as a particular case of a *processing module*, which is:

a) an autonomous *active* entity, i.e. with its own *self-control* capability. This means that the decisions taken by a module, during any step of its behavior, cannot be forced by any other module, instead decisions are taken autonomously according to the module internal state and to the values of the input information;

b) internally described by a *sequential* program. This program describes the sequence of steps of the module autonomous behavior, operating on the internal state and the input information, and in general producing output information;

c) executable in parallel with other modules and cooperating with them through proper concurrency mechanisms.

Because of property *a)*, the module is fully and autonomously responsible about if and when to accept input information generated by other modules (on the contrary, a class instance in an object-oriented language is not a module according to our definition).

Specific features of the firmware level are the following:

1) in turn, the processing unit computation is *interpreted* at the hardware level,

2) the *hardware level* consists of *logic circuits*: combinatorial circuits (e.g., an adder), sequential circuits (e.g. a string recognizer), memory elements (e.g. clocked registers), and links. Through a proper formal definition of the unit behavior, the unit can be viewed as a pair of interconnected sequential circuits, called the ***Operation Part*** and the ***Control Part***. The Control Part is the concrete implementation of the self-control capability concept: at each step, it orders the actions to be executed to the Operation Part, where all the necessary computing resources are properly organized;

3) the internal sequential behavior is described by a ***microprogram***, whose commands (*microinstructions*) consist of microinstruction label, logical conditions, branches, and register assignment statements. For example, a microinstruction could be:

current_microinstruction. *if* (sign (A) = 0)

then {A – B → A, C+ 1 → C, goto next_microinstruction_1}

else {A + C → B, 0 → D, goto next_microinstruction_2}

where:

- A, B, C, D are names of registers in the Operation Part, which are the *variable* containers on which the computation operates;
- the assignment is denoted by the → operator;
- the "," operator means "in parallel during the same clock cycle".

The semantics is the following: the current internal state of the Control Part corresponds to the *current_microinstruction* label; being in this internal state. the Control Part observes the input *sign(A)* which is an output of the Operation Part (output of the most significant bit of register A); if this input is equal to zero, then the Control Part generates proper output values which make it possible the execution of the assignments A – B → A and C+ 1 → C in the Operation Part, and the Control Part passes into the next state corresponding to the microinstruction label *next_microinstruction_1*; else …;

4) both the Operation Part and the Control Part are ***synchronous*** sequential circuits *with the same clock signal*. Let *f* be the clock frequency (e.g. *f* = 2 GHz). Then each microinstruction is executed in a fixed time interval $\tau = 1/f$, called the ***clock cycle*** of the processing unit (e.g. τ = 0.5 nsec): this means that both the Operation Part and the Control Part stabilize their internal state within the clock cycle duration;

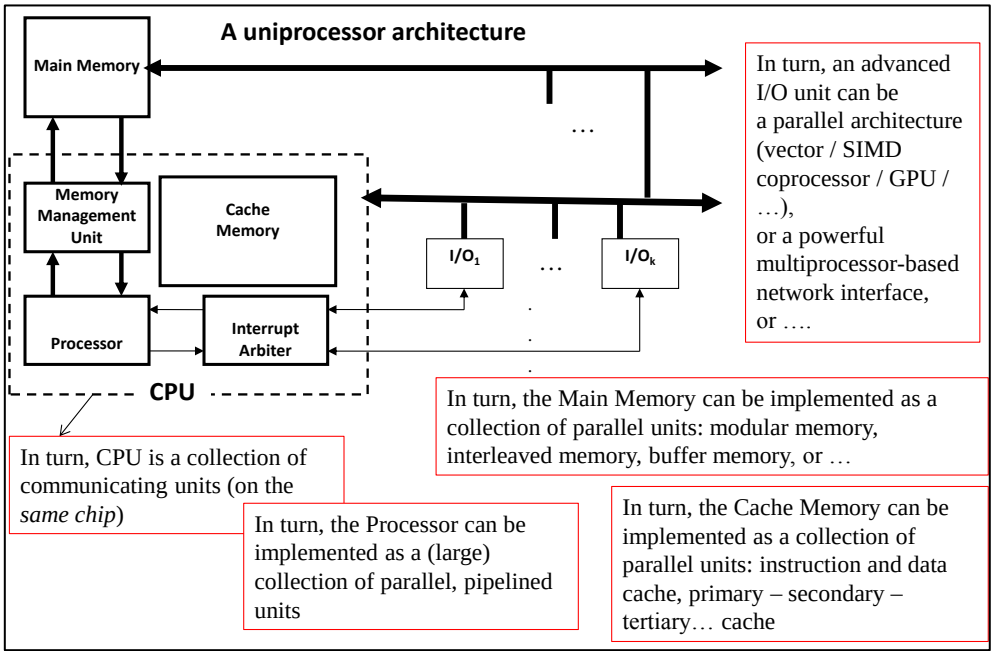
5) the *synchronous model of execution* makes it possible to efficiently exploit parallelism between register assignment operations. For example, the sequential computation {A + B → B; A + 1 → A} is equivalent to the following parallel one: {A + B → B, A + 1 → A}, because the contents of A is stable during the clock cycle, and it will be incremented only at the beginning of the next clock cycle;

6) at the firmware level, only cooperation via explicit *message-passing* is possible (i.e., no shared variable between units can exist). For the communication with the other units, the

decision of "if and when, and with which unit, to communicate" is explicitly represented in the microprogram. Proper communication protocols and interconnection structures at the firmware level will be studied in Section 2.

1.4.2 Firmware architecture of general-purpose computers

The following figure shows a possible firmware architecture of a uniprocessor general purpose computer:



In modern computers, each block in this figure is composed by multiple processing units, *each one* with its own Operation Part and its own Control Part. For example, a pipelined CPU is a large collection of processing units including:

- Primary instruction cache
- Primary data cache
- Memory management units
- Instruction issuing unit and Instruction preparation unit
- Register renaming unit
- Branch Unit
- Master Execution Unit
- Functional units for arithmetic operations, each one composed of several pipelined stages,
- Interrupt unit
- Secondary Cache
- External memory interface unit
- etc.

Notice that *there is no centralized "locus" of control*: even in a uniprocessor architecture, the overall behavior of the system at the firmware level is realized by the "peer-to-peer" cooperation of the various units, each one managed by its own Control Part.

As indicated in the figure, also the external memory and the I/O units are realized by (large) collections of processing units, and proper *interconnection structures* between all the units of the system have to be realized in order to achieve satisfactory values of bandwidth and latency. All such issues will be studied in Section 2.

Of course, the same principle holds for parallel architectures (multiprocessors, multicomputers), where the parallelism degree is much higher, however without the need to introduce centralized "loci" of control.

As said, the assembler language is entirely interpreted by the underlying firmware level. Informally, the firmware interpreter of an *elementary* sequential processor has a microcode structure of this kind:

```
while (true) do
{ read the instruction, whose address is the current content of the program counter, from the
  memory; // this implies proper communication between the processor and one or more
  memory hierarchy levels //
  decode the instruction according to the operation code;
  if needed, read the operand values from the memory hierarchy  or from registers;
  execute the instruction;
  if needed, store the results into the memory hierarchy or into registers;
  modify the program counter with the address of the next instruction to be executed;
  if an exception occurred or interrupt events are signaled, then call the proper handler procedure
}
```

As we'll see, this is a very simplified and unrealistic view of the firmware interpreter: not only because it is informally and generically expressed, but mainly because it has a sequential behavior. Today, no processor is so elementary! This description is just to show a sequential algorithm for the firmware interpreter, which will be properly restructured and parallelized in order to efficiently exploit

a) the memory hierarchy,

b) the pipelined behavior.

Topics *a)* and *b)* correspond to quite fundamental issues in computer architecture, both from a conceptual and from a technological point of view.

1.4.3 On the existence of the assembler machine

Finally, we come back to the question posed in Section 1.3.1 about the real need of the assembler level.

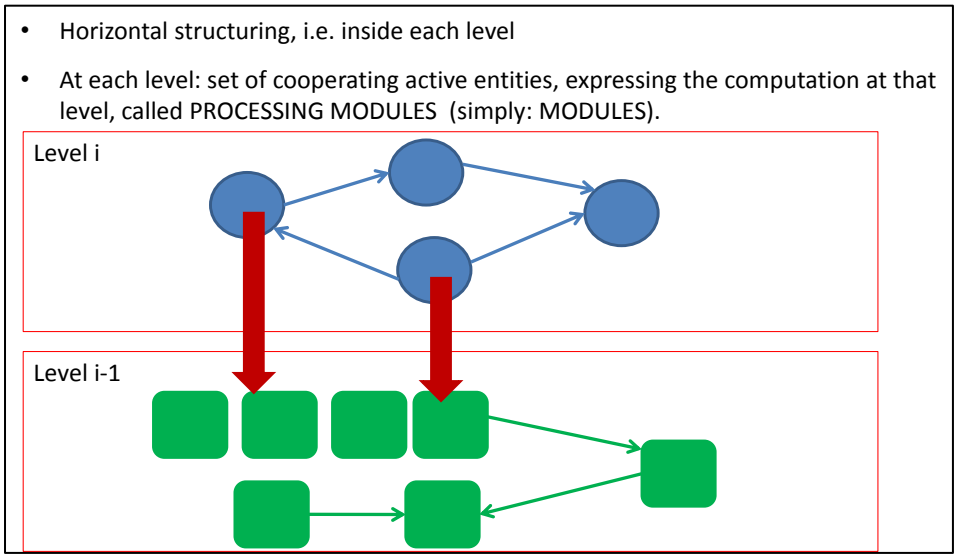
During the process execution, each instruction is interpreted by the firmware machine, thus the final effect is that each process command is executed through a (long) sequence of microinstructions. This sequence *per se* does not contain optimizations, which however has been introduced in the assembler code. If we attempt to eliminate the assembler level and, at the same time, if we wish to maintain optimizations, in principle we could *compile* the process into a microcode directly. In fact, this solution has been adopted in same computers of several years ago, when the realization of microprogrammable processors was technologically feasible (i.e. loading the processor microcode in the same way of the assembly code loading). With the current technology, this solution is no more effective, for the following reasons:

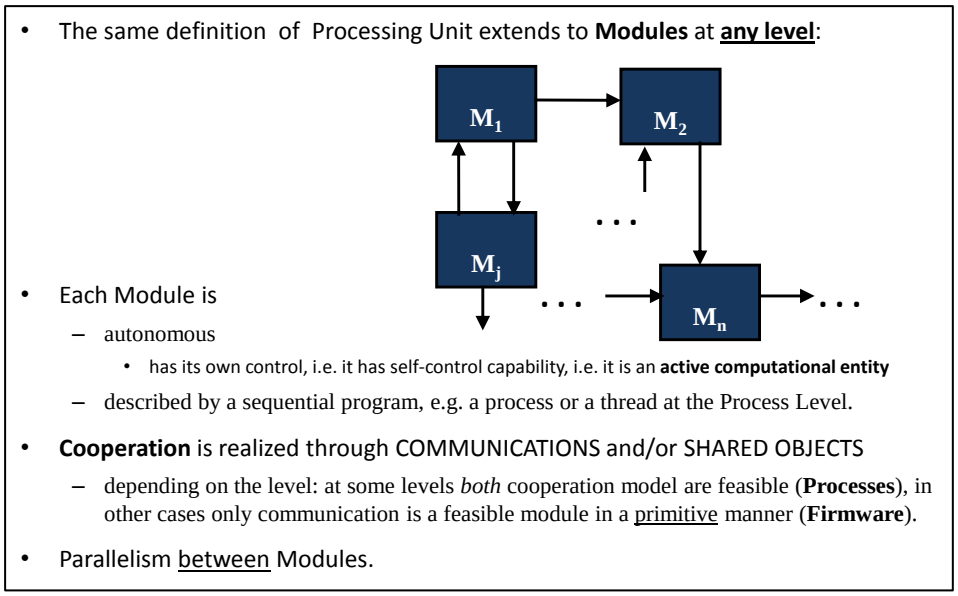
- because of the VLSI on-chip integrated realization of CPU and of the memory hierarchy, the dynamic loading of microcode - each time a process becomes running - is very difficult and expensive;
- in principle, the performance of a microcoded compilation could be better than an assembly code compilation. However, this difference tends to vanish owing to the parallel-pipelined CPU realization, in which the ideal processor bandwidth is equal to one instruction per clock cycle (scalar CPU) or even equal to more than one instruction per clock cycle (superscalar CPU).

In conclusion, the existence of the assembler level is fully justified by higher flexibility and better technology exploitation without sacrificing performance.

1.5 Horizontal structuring by processing modules

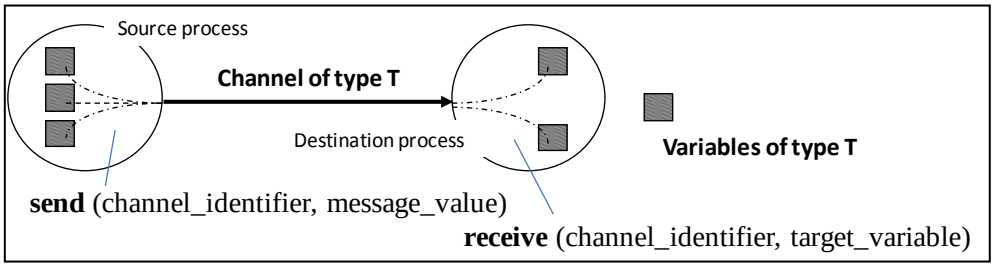
The system structure at the firmware level is a good example of the horizontal structure. Inside *any* level, the system is a collection of processing modules having the features already defined:



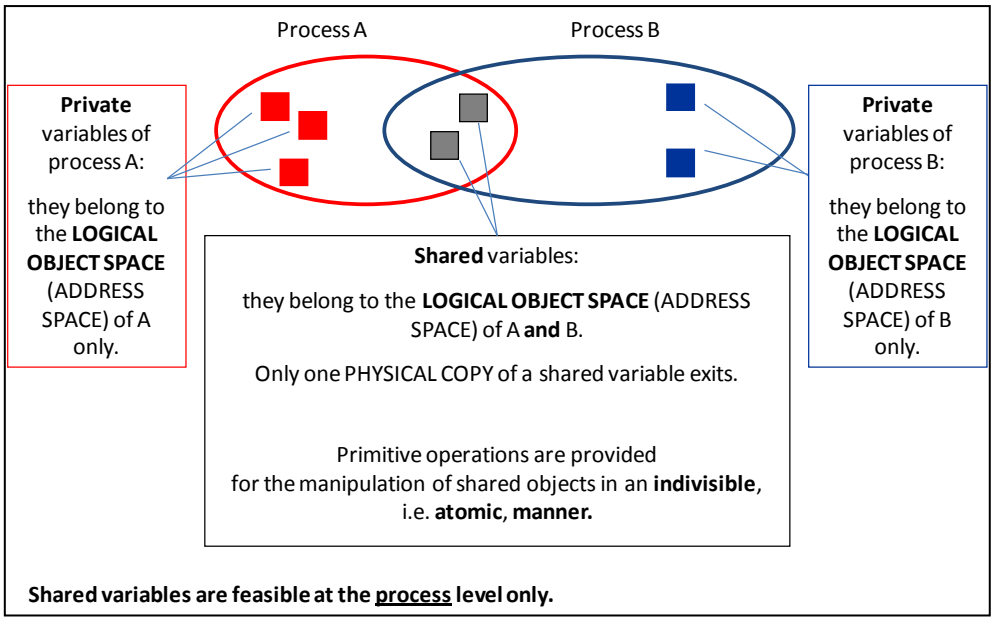


In fact, the *firmware level* and the *process level* are the virtual machines of main interest for our purposes. At the process level, modules are *processes* or *threads* (the terminology will be clarified in successive parts of the course. *Threads* are merely a technological variant of the process concept. Thus, for the moment being, it is sufficient to use the term "process").

Processes can cooperate by *message-passing* (like at the firmware level): proper inter-process communication mechanisms will be studied in Section 6. Using a figure of Section 1.2:



Moreover, cooperation mechanisms based on *shared variables* can be used at the processes level. Using a figure of Section 1.2:



Typical mechanisms based on shared variables, available in many languages and libraries, include:

- Semaphores and locking mechanisms,
- Condition variables,
- Critical sections,
- Monitors

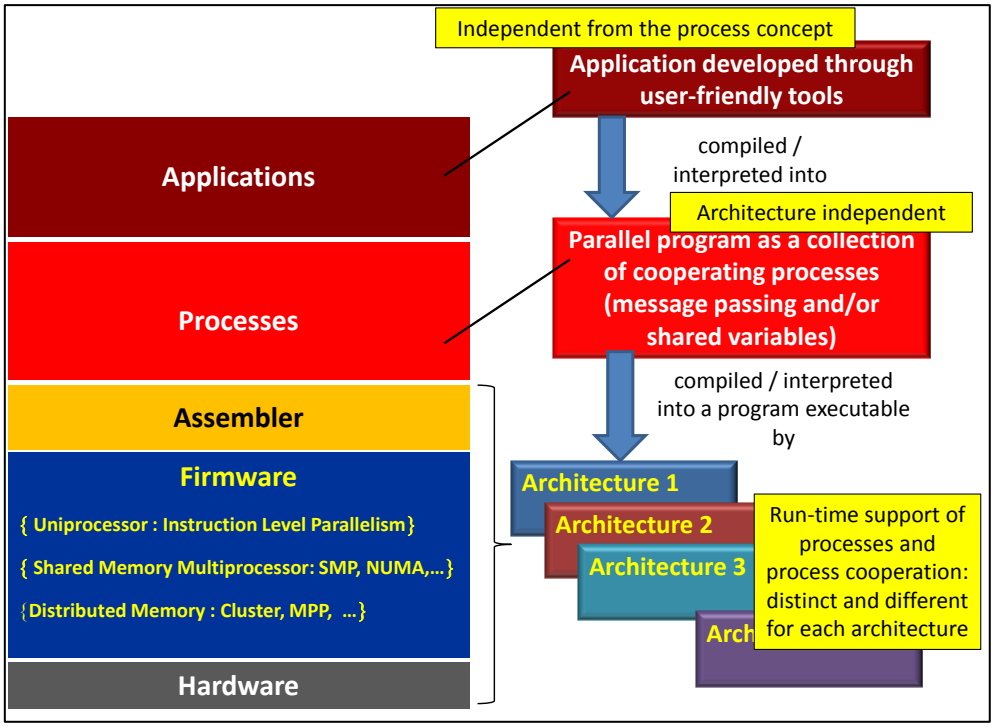
with an increasing abstraction view of concurrency and cooperation.

In our course, we are especially interested in the message-passing paradigm for inter-process cooperation, while the mechanisms based on shared variables will be studied for defining the process run-time support in shared memory multiprocessor architectures.

1.6 Performance evaluation: abstract machines and cost models.

To conclude Section 1, it is worth to introduce the issues of performance evaluation which, in our course, will be studied in strict relation with the system structure by levels and by modules.

The following figure is a synthetic view of the course approach:



The system structuring by levels is shown at the left, while the rigth part illustrates the conceptual and technological scheme for high-performance application development:

- applications should be designed at the highest level by means of formalisms and tools that are *fully independent from the machine architecture and from the mechanisms at the process level*;
- as said, a *concurrent language* at the process level is the intermediate language into which the abstract parallel applications are compiled and/or interpreted. At this level, several optimizations are applied according to the ***cost model*** that characterizes the high-level formalism and methodology, in order to evaluate and to predict performance parameters, like processing bandwidth, completion time, latency, response time, efficiency, scalability, and so on.

Currently, the most common situation consists in developing parallel applications directly at the process level by means of sequential languages plus message-passing libraries (e.g. MPI) or shared-objects libraries (e.g. OpenMP). This approach is not suitable for a high-level, portable and "productive" methodology, and there is a clear trend to overcome it. However, even when the only available tools are process level libraries, it is important that the *methodology* that we study for the application level is equally applied, i.e. the programmer can adopt the high-level methodology to define parallel applications, and can provide a sort of "compilation by hand" into the process version;

- the *run-time support of the process level* (concurrent language or libraries) is *different for each distinct parallel architecture at the assembler-firmware level*. For example, we saw that the run-time support for multiprocessors exploits the physically shared memory intensively, while in a multicomputer proper mechanisms are adopted to establish an

efficient cooperation among distributed memory nodes. For each specific parallel architecture, a proper set of run-time libraries and optimization techniques is used by the compiler of the concurrent language.

A key concept is the following:

- *the cost model for parallel applications depends in large part on the paradigms* adopted for structuring the parallel computation, that are architecture independent,
- while the impact of the underlying architecture can be focused on some parameters expressing the cost of process cooperation mechanisms.

For example, the *interprocess communication latency* L_{com} is a key parameter in our cost models. It measures the latency for completing an interprocess communication, i.e. to copy the message into the target varibale and to perform all the needed synchronization and low-level scheduling actions. We will see that L_{com} captures a large number of characteristics of the underlying architecture, like

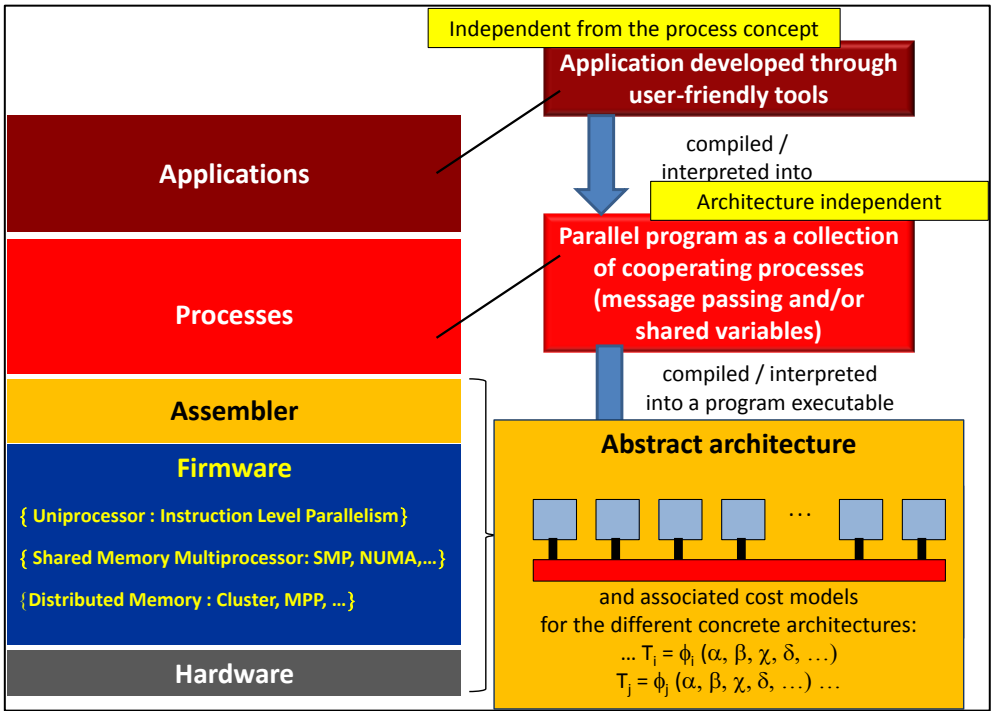
- shared *vs* distributed memory,
- interconnection structures,
- process scheduling,
- memory hierachies,
- features of CPUs, coprocessors, and other processing units,

and so on.

Other parameters of the cost model, notably the *calculation time*, depend on the above characteristics.

In order to take into account all the concepts and features discussed above, we utilize a typical concept in Computer Science: ***abstract architecture***.

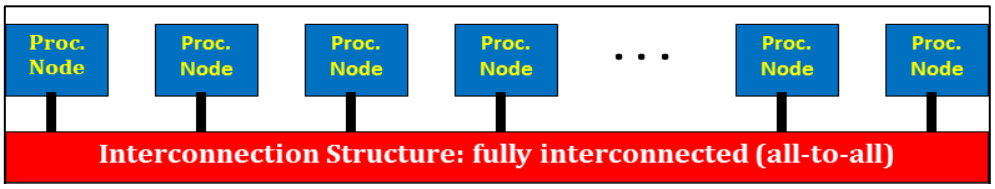
An abstract architecture is able to capture the essential characteristics and features of several, different physical parallel architectures:



The specificity of each individual physical architecture is expressed by the value of some parameters of the cost model (communication latency, calculation time, etc.).

As it is typical of the abstract architecture concept, the compiler "sees" the abstract architecture in order to transform and to optimize the parallel applications, by applying the cost model associated to the abstract architecture itself.

If the interprocess cooperation model is the message-passing one, a reasonable definition of the abstract architecture, both for shared and for distributed memory parallel machines, is the following:



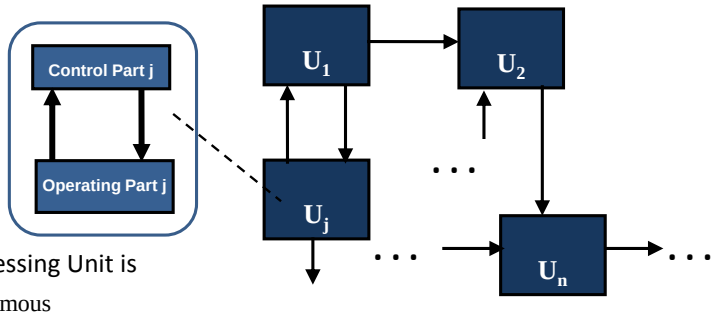
1. a distributed memory architecture, consisting of as many processing nodes as the processes of the parallel program are. That is, each process of the parallel program is allocated onto an independent processing node;
2. every node has the same characteristics of the (abstract) uniprocessor architecture of the processing nodes;
3. nodes interact through a *fully interconnected network*, in which each node is connected by a dedicated link to any other node. An interprocess communication channel is represented by the physical channel connecting the corresponding processing nodes;
4. if the concrete architecture is able to support *calculation-communication overlapping*, then it is supported by the abstract architecture too.

2. The firmware level

Starting from the introduction of Section 1.4, in this Section we study the main concepts and techniques for the realization of any processing unit and for the communication between units.

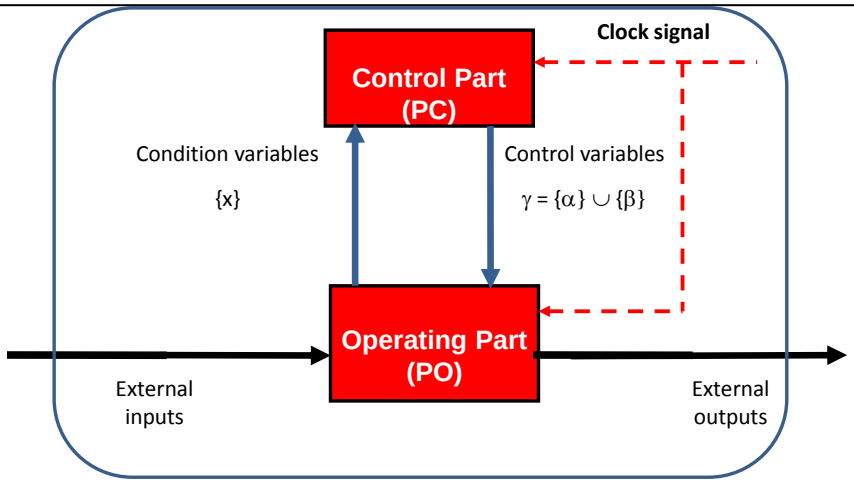
The following figure resumes the main features of the firmware level horizontal structure:

- At this level, the system is viewed as a collection of cooperating modules called PROCESSING UNITS (simply: units).



- Each Processing Unit is
  - autonomous
    - has its own control, i.e. it has self-control capability, i.e. it is an **active computational entity**
  - described by a sequential program, called microprogram.
- Cooperation is realized through COMMUNICATIONS
  - Communication channels *implemented* by physical links and a firmware protocol.
- Parallelism between units.

As introduced in Section 1.4, each unit is composed of two interconnected automata, called ***Control Part*** *(PC)* and ***Operation Part*** *(PO)*, which are realized as *synchronous sequential machines with the same clock*:



PC and PO are ***automata***,
implemented by *synchronous sequential circuits*

Clock cycle: time interval τ for executing a (any) microinstruction.
Synchronous model of computation.

The microprogram, which describes the behavior of the processing unit, is *interpreted at the hardware level* through the implementation of the sequential circuits PC and PO. Thus, in order to understand this model, we need to resume some basic concepts about *logic circuits*, which is a well known part of every course in computer organization/architecture at the Bachelor degree level.

2.1 Hardware level: logic components for PC and PO implementation

2.1.1 Combinatorial components

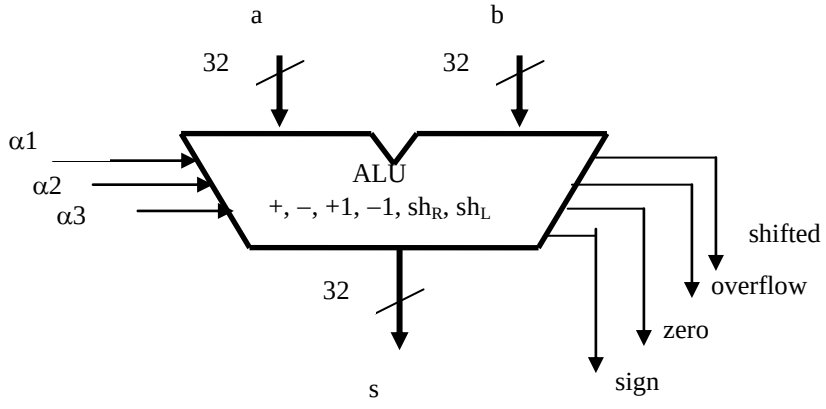
Combinatorial circuits are the hardware implementation of pure *functions* operating on binary variables. In general, the design of a combinatorial circuit implementing a function $y = F(x)$ passes through the formal definition of F and its logical expression in terms of AND-OR-NOT boolean operators (or equivalent boolean operators), possibly exploiting some heuristics in order to face with the exponential complexity of the formal design procedure.

For computer structures, only some special functions, e.g. small PCs, are implemented from scratch using this procedure. For the Operation Part, which has an unacceptably large number of states, a small set of *standard components* is used. The concept is that, starting from the microprogram, any PO can be easily and formally built as a graph whose nodes are instances of such standard components, and whose arcs are physical links connecting the standard components.

A complete set of combinatorial standard components is the following:

1) ALU

A multifunction component able to execute one out of a set of n arithmetic-logic functions, operating on m-bit numbers (typically, m is the *word length*, e.g. 32 or 64 bits). The selection of the desired function is driven by $\lceil lg_2 n \rceil$ boolean control variables. The following figure shows an ALU able to perform six operations (thus, three control variables α_1, α_2, α_3 are needed) on 32-bit input operands *a*, *b*. The primary output result is a 32-bit value *s*. Moreover, four secondary 1-bit outputs ("flags") are available.



A possible definition of the primary output is:

s = *case* α_1 α_2 α_3 *of*

000: a + b;

001: a – b;

010: a + 1

011: a – 1;

10–: sh_R(a) //1-position right shift //

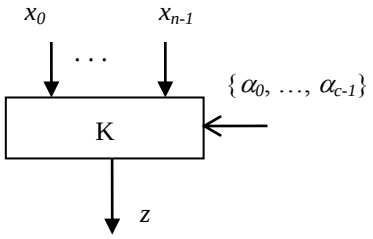
11–: sh_L(a) //1-position left shift //

For each combination, some flags are significant. For example: the "zero" flag assumes the value 1 iff the operation result is equal to zero; the "sign" flag is equal to the most significant bit of the operation result in the adopted signed-numbers representation (e.g. 2's complement).

If t_p is the stabilization delay of an AND/OR gate, a typical stabilization delay of an ALU, including the basic addition operation, is equal to few t_p for m = 32-64 bits, typically from 5 to 10 t_p. As a matter of fact, with the current technology t_p is of the order of magnitude of 10^{-2} nsec.

2) Multiplexer / Demultiplexer

A *multiplexer* K with n primary m-bit inputs $x_0, \ldots, x_{n-1}$, one m-bit output z, and $c = \lceil lg_2 n \rceil$ boolean control variables $\{\alpha_0, \ldots, \alpha_{n-1}\}$, is defined as follows (it is shown the case for $n = 2^c$):



z = *case* $\alpha_0, \alpha_1, \ldots, \alpha_{c-1}$ *of*

0 0 … 0 : x_0

0 0 … 1 : x_1

…

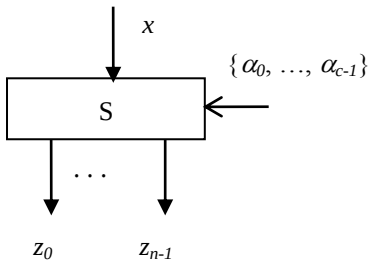
In a PO, a multiplexer is used to route one out of n values onto a specific destination. A more general, very useful interpretation is the following:

- ***array indexing***: given the array X = [x_0 x_{n-1}] and the *index i* equal to the boolean configuration $\alpha_0, \ldots, \alpha_{n-1}$, then $z = X[i]$,

or, equivalently:

- ***memory reading***: given the memory X = [x_0 x_{n-1}] and the *address i* equal to the boolean configuration $\alpha_0, \ldots, \alpha_{n-1}$, then $z = X[i]$ is the result of the reading operation.

A *demultiplexer* S is defined as follows (for $n = 2^c$):



(z_0 z_{n-1}) = *case* $\alpha_0, \alpha_1, \ldots, \alpha_{c-1}$ *of*

0 0 … 0 : (0 0 … 0 x)

0 0 … 1 : (0 0 …x 0)

…

The input value x is routed onto the indexed (addressed) output <u>*and*</u> all the other outputs are forced to zero.

Notice that this function is *not* the multiplexer dual, and it is *not* equivalent to an usual array modification (or to a memory writing) operation. Proof: the array modification / memory writing is defined by "the input value x is routed onto the indexed (addressed) output *and all the other outputs remain unchanged*", which *is not a function*, instead it is an automaton.

For low n, the stabilization delay of a multiplexer and of a demultiplexer is equal to $2t_p$ and to t_p, respectively (K is an AND-OR circuit with n AND gates, S is merely composed of n AND gates).

In general, the stabilization delay of a multiplexer is *logarithmic* in the number of inputs:

$$\left(\lceil lg_g(c+1)\rceil + \lceil lg_g(n)\rceil\right)t_p$$

where g is the *fan-in* of AND-OR gates, i.e. the maximum number of gate inputs for which the maximum stabilization delay is equal to t_p.

The reader is invited to demonstrate this formula, which has very important consequences on technologies.

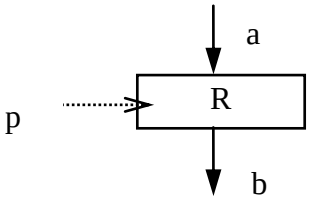
For example, for $g = 8$, the stabilization delay of a multiplexer with $n = 1024$ inputs is equal to $5t_p$ only. We'll see that this fact is the key for effective technologies of large memories.

2.1.2 Registers and memory components

These components are used to stabilize (i.e., to store) the internal state of a sequential circuit. In a PO, registers and register-memories (register-files) represent the physical support to microcode *variables*, respectively of elementary type and of array type.

1) Clocked register

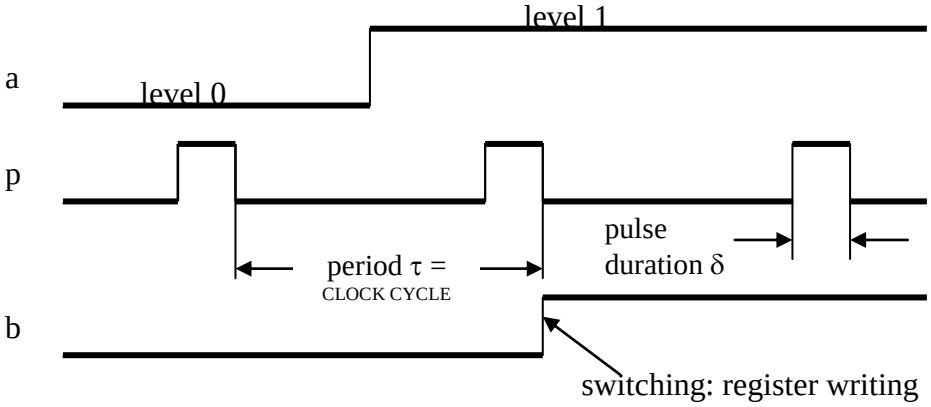
A basic *m*-bit memory element R, with one *m*-bit primary input *a*, one *m*-bit output *b*, and one secondary pulse input *p* (the *clock signal*), is defined as follows:



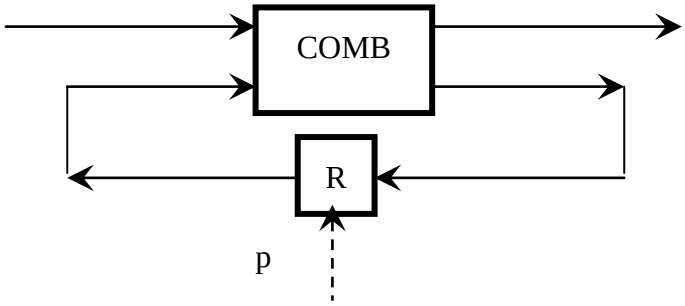
when p do $b = a$

meaning that the output state assumes the value of the input state only when the pulse is present, otherwise the output state is unchanged. The register output coincides with the internal state, thus the *register output* is *stable* during a **clock cycle** (period between two consecutive pulses of the clock signal).

A variation of the input state is registered only at the end of the next pulse, as shown in the following figure:



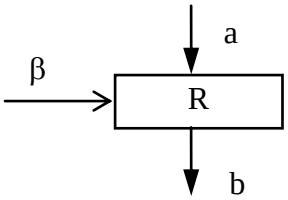
In a **synchronous sequential circuit**, with combinatorial part COMB (implementing the output function and next state transition function of the automata) and internal state implemented with clocked registers:



in order to have a correct behavior the maximum stabilization delay of COMB must be

$$< \tau - \delta$$

In a PO, in general we are interested in enabling register writings explicitly and selectively: this is easily done by AND-combining the pulse signal with an *enabling control variable* β for each register. The conventional symbol and the definition of a register become (the AND gate with inputs *p* and β is not shown in the conventional symbol):

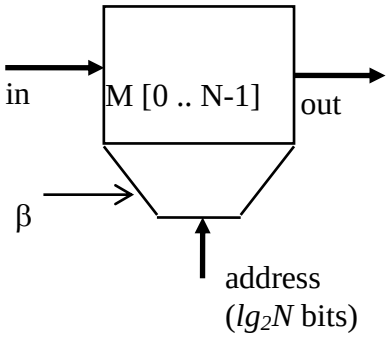


when (*p* **and** $\beta = 1$) **do** $b = a$

2) Memories - RAM

Independently of the specific technology, conceptually a *Random Access Memory* (RAM) is a hardware implementation of an array *M[N]* of *N m*-bit clocked registers, on which the reading and writing operations *by index* are defined. The index is called the ***memory address***. *N* is called the ***capacity*** of M.

The conventional symbol and the semantics of a RAM component are the following:



Reading:

out = M[address]

Writing:

$in_{M[i]}$ = **when** (pulse **and** β = 1 **and** i= address) **do** in

The *reading* operation is always enabled: after the stabilization delay, the memory output *out* is equal to the content of the addressed register. Logically, it is implemented by a *multiplexer* whose primary inputs are the *N* register outputs and whose control variables are the address bits.

The *writing* operation is enabled provided that the β enabling signal is equal to one. If so, the next state of the addressed register is equal to the input value *in*. Logically, it is implemented by a

demultiplexer whose primary input is β, the control variables are the address bits, and the outputs are the *N* register enabling variables $\beta_0, \ldots, \beta_{N-1}$. All the *N* registers primary inputs are equal to *in*.

In the microcode design, memories are used to express array data types or to simulate more complex data structures, exactly in the same way of the high-level programming practice. For example, a microinstruction could contain the following operation:

$$A + B \rightarrow M[J]$$

PO contains the simple registers A, B, J *and* the memory M (for example, M[32K]). In this example, M is addressed by register J, i.e. the J register output is connected to the address input of M.

The logical implementation by multiplexer/demultiplexer demonstrates that the ***memory access time*** (i.e. the maximum time delay for stabilizing the reading/writing operations) depends *logarithmically* on the memory capacity:

$$t_a = \left(\lceil lg_g(lg_2 N + 1) \rceil + \lceil lg_g N \rceil \right) t_p \sim \left(\lceil lg_g N \rceil \right) t_p$$

For example, for N = 4G words (G = Giga = 2^{32}) and $g = 8$, $t_a \sim 11\ t_p$.

For *static RAM* (SRAM), t_p is of the same order of magnitude of low-latency AND-OR gates adopted for CPUs, i.e. for current SRAM $t_p << 1$ nsec. In the previous example, t_a is equal to few nsec or less. SRAM is the memory technology for processor register files and cache memories.

For *dynamic RAM* (DRAM), t_p is at least two order of magnitude greater than for SRAM, because of the refresh overhead. In the previous example, t_a varies from some hundreds of nsec to one or few μsec. DRAM are much slower, however they are much cheaper and much more dense, than SRAM.

For *synchronous DRAM* (SDRAM), t_p is about three-four times lower than for DRAM. SDRAM is an effective current technology for large main memories.

Note: in all this course, we'll assume that the memories are *word-addressable*, instead of byte-addressable, in order to simplify the reasoning and the performance evaluations, without incurring in significant discrepancies wrt the commercial practice.

2.2 Microprograms and clock cycle

The concept of clock cycle, which has been introduced in the previous section, is a very fundamental one to study computer architecture. Often, when describing or designing a system, we must be able to answer questions such as:

- Which operations are feasible in a single clock cycle?
- How many operations can be executed in parallel during the same clock cycle?
- How many clock cycles are needed to execute a given algorithm at the firmware level?
- Is a certain value of the clock cycle compatible with the current technology?
- How to evaluate interesting metrics for system performance?
- Which impact has the communication latency on the whole performance?

Let's start by fixing some concepts through microprogram examples and their implementation.

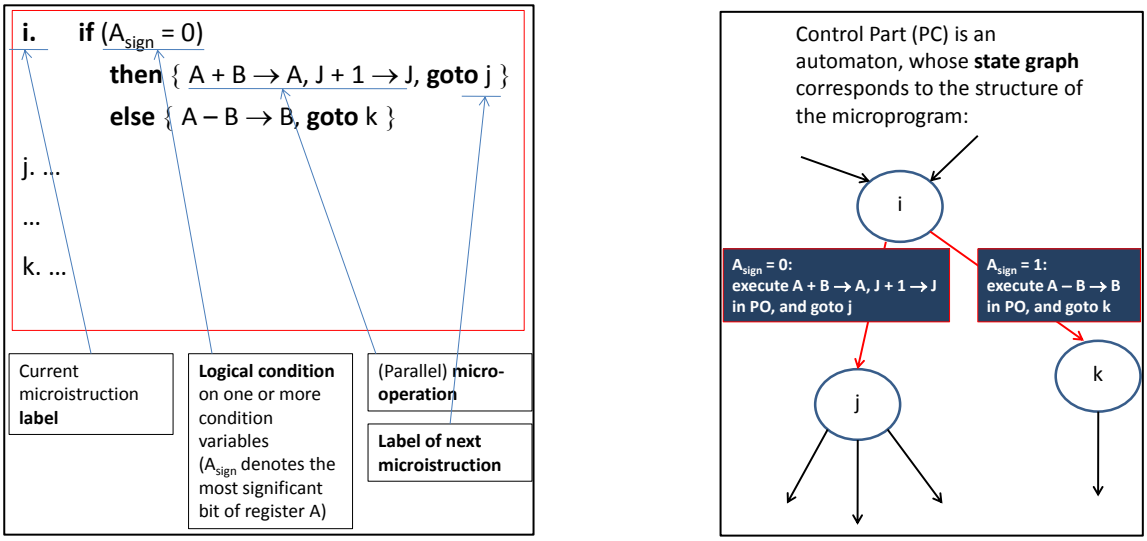
2.2.1 Microinstructions

As introduced in Section 1.4, microprograms are written using a formalism which is suitable for formally deriving the definition and implementation of the PC and PO synchronous sequential circuits.

A *microistruction* is a command that is *executed exactly in one clock cycle*. It contains the following elements:

1. *Current label*, denoting the current state of PC,
2. *Logical conditions*: predicates applied to boolean variables stable in PO during the clock cycle in which the microinstruction is executed (***condition variables***)
3. *Micro-operation*, in general containing more operation executable by PO in parallel during the same clock cycle,
4. *Next microinstruction labels*, i.e. the next internal state of PC.

An example of microinstruction is shown at the left of the following figure:



Let assume that PO contains registers A, B, and J, and that it is able to execute functions as addition, increment by one, and subtraction. The microinstruction semantics is: being PC in state *i*, during one clock cycle

- the value of a condition variable (A_{sign}: the output if the most significant bit of register A) is tested,
- if $A_{sign} = 0$, the assignments A + B → A and J + 1 → J are executed by PO in parallel, and PC passes into the next state *j*,
- otherwise, the assignment A – B → B is executed by PO, and PC passes into the next state *k*.

At the right of the figure a fragment of PC behavior (state graph) is illustrated: being in current state *i*, and according to the value of the input variable A_{sign}, PC causes the execution in PO of A + B → A and J + 1 → J, or of A – B → B. This is done by generating proper values of the ***control variables*** that select the operations in ALUs, select the data routes in multiplexers, and enables the writing operation in the destination registers. Moreover, according to the value of A_{sign}, PC passes into the next state (*j* or *k*).

According to the micro-operations of all the microinstructions, we can build the PO structure as a proper *graph of ALU, multiplexers and registers* that are consistent with the specification represented by the microprogram itself. This is done by a low-complexity procedure.

For simplicity, let us assume that the microprogram consists of microinstruction *i* only. We need:

- one ALU to execute the addition *or* the subtraction of A and B: they are never executed in parallel, thus only one ALU is needed for both. This ALU has a control variable α to distinguish between the execution of the addition or the subtraction. The output of this ALU is connected to the input of A *and* of B;
- a distinct ALU for incrementing J, in order that the parallelism of the micro-operation is respected. This ALU has no control variables. The output of this ALU is connected to the input of J.

Moreover, A, B and J have distinct control variables, β_A, β_B, β_J, for enabling the writing operations. For the first micro-operation (when $A_{sign} = 0$) PC generates the following control variable values: $\alpha = 0$, $\beta_A = 1$, $\beta_B = 0$, $\beta_J = 1$. Otherwise, for the other micro-operation: $\alpha = 1$, $\beta_A = 0$, $\beta_B = 1$, $\beta_J = 0$.

The drawing of this graph of PO is left as an exercise.

Notice that the output of the first ALU is connected unconditionally to the inputs of both A *and* B. It is according to the βs configuration that the ALU result is written into A ($\beta_A = 1$, $\beta_B = 0$) or into B ($\beta_A = 0$, $\beta_B = 1$). In a different example, if needed it could also be possible to write the same value into A and into B in parallel: $\beta_A = 1$, $\beta_B = 1$. In a different example, no modification of A and B is expressed by $\beta_A = 0$, $\beta_B = 0$.

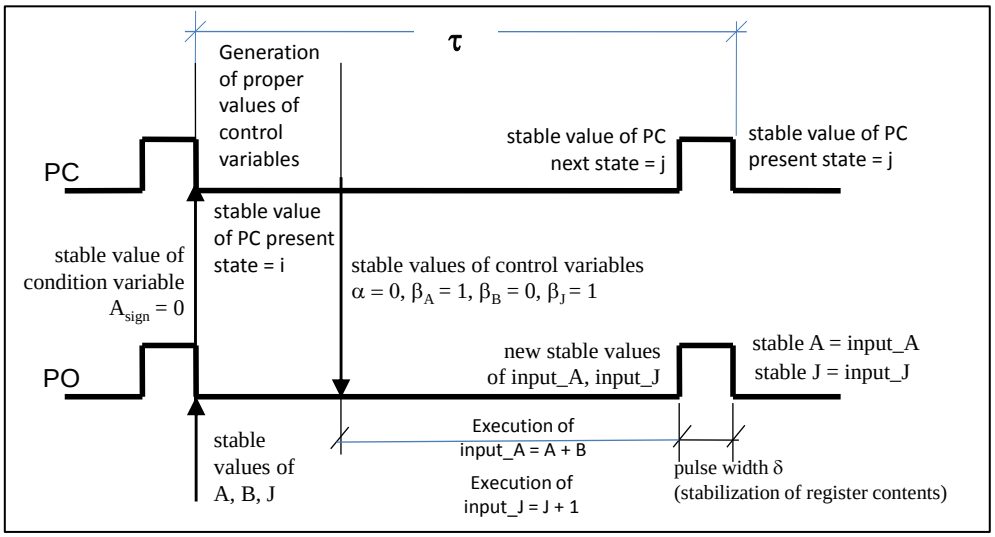
2.2.2 Clock cycle

Referring again to the example of microinstruction

i. **if** ($A_{sign} = 0$) **then** { $A + B \rightarrow A$, $J + 1 \rightarrow J$, **goto** j } **else** { $A - B \rightarrow B$, **goto** k }

let us look at the events that occur during the clock cycle in which the microinstruction is executed.

Let us assume that, with the PC in state *i*, PO generates $A_{sign} = 0$. The following situation occurs:



At the beginning of the clock cycle, A, B and J are stable in PO, while PC is stable in state *i*. Being $A_{sign} = 0$ stable, the PC output function has all the inputs stable (internal state *i*, input state $A_{sign} = 0$) so, after its stabilization delay, the control variables are generated in stable form ($\alpha = 0$, $\beta_A = 1$, $\beta_B =$

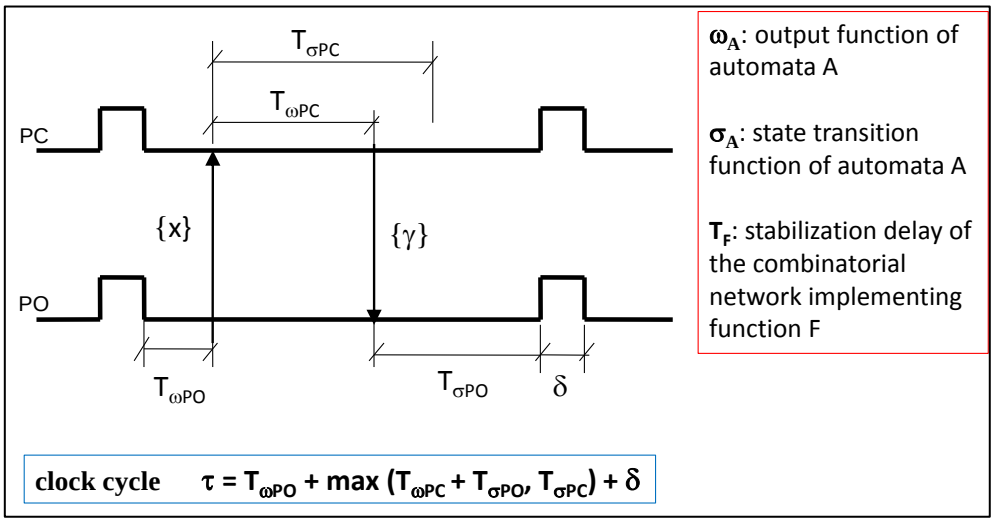
0, $\beta_J = 1$). At this point, all the inputs of the PO next state transition function (consisting in the calculation of A + B and J + 1) are stable (internal state A, B and J, input state $\alpha = 0$, $\beta_A = 1$, $\beta_B = 0$, $\beta_J = 1$) so, after its stabilization delay, the input values of A and J become stable: they represent the PO *next* internal state, which will become the PO *present* internal state after the clock pulse. In the meantime, PC have executed the next state transition function, passing into the state corresponding to microinstruction *j*, which will become the PC present internal state after the clock pulse.

The stabilization delays of the PC output function ($T\omega\text{-}_{PC}$) and of the PO next state transition function ($T\sigma\text{-}_{PO}$) can be easily evaluated as multiples of t_p. Thus the minimum value of the clock cycle is

$$\tau \geq T_{\omega\text{-}PC} + T_{\sigma\text{-}PO} + \delta$$

For example, let $T_{\omega\text{-}PC} = 2t_p$, $T_{\sigma\text{-}PO} = 5t_p$, $\delta = t_p$. With $t_p = 0.05$ nsec, we have $\tau \geq 0.4$ nsec, $f = 1/\tau \leq$ 2.5 GHz. The best standard pulse generator, compatible with this relation, is chosen: for example, f = 2.3 GHz, corresponding to $\tau = 0.44$ nsec.

A more general formula, using the automata theory applied to the automata system PC-PO, exists for evaluating the clock cycle upper bound. It is shown in the following figure, taking into account also the stabilization delays of the PO output function ($T_{\omega\text{-}PO}$) and of the PC next state transition function ($T_{\sigma\text{-}PC}$):



[In terms of formal automata theory, in our model PC is a *Mealy* automaton (during any clock cycle, its output function depends both on the input state and on the current internal state), and PO is a *Moore* automaton (during any clock cycle, its output function depends on the current internal state only). This formal treatment of microprogramming theory is out of scope of the present document: the interested student can ask for additional information and material]

In the previous example $T_{\omega\text{-}PO} = 0$. In different examples, we could have $T_{\omega\text{-}PO} > 0$ if a condition variable is the output of a combinatorial circuit with non-null delay. For example consider a microinstruction which tests the predicate *zero (A – B)* as an output ALU flag.

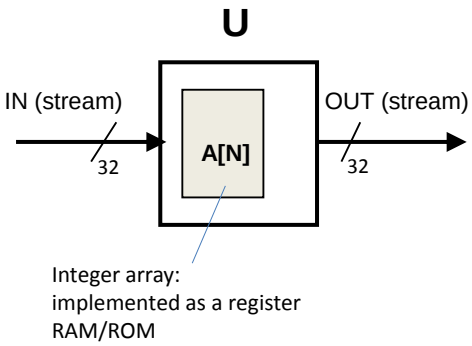
In general, $T_{\sigma\text{-}PC}$ is fully overlapped to $T_{\omega\text{-}PC} + T_{\sigma\text{-}PO}$.

2.3 Processing unit design method through an example

The formal procedure to design a processing unit is called ***Gerace's method***, in memory of the research pioneering work of Giovan Battista Gerace, who was also one of the founders of the first Computer Science Department in Italy at the University of Pisa.

2.3.1 Specification and microprogram

Let us consider the following specification of a processing unit U:



Specification $\forall IN$: *OUT = number_of_occurrences (IN, A)*

Pseudo-code of interpreter:

```
int A[N]; int IN, OUT, B, C;
while (true) do
        { B = IN; C = 0;
          for (J = 0; J < N; J ++)
                  if ( A[ J ] == B) then C = C + 1;
          OUT = C
        }
```

As it occurs usually, the processing unit works ***on streams***, i.e. it receives a possibly infinite sequence of input values *IN* (32-bit integer in the example) and, for each input value, applies a certain computation (in the example: number of occurrences of *IN* in a 32-bit integer array *A[N]*) and produces a corresponding output value *OUT* (32-bit integer in the example).

For the moment being, we neglect the synchronization with the source and the destination of the streams. That is, we assume that, at each new iteration of the external *while*, a new meaningful value of *IN* is received. This assumption is not realistic in practice. The synchronization issue for correct communication will be studied in Section 2.4.

The problem specification (for each *IN*, compute the number of occurrences of *IN* in *A*) has been expressed by a high-level pseudo-code, as we are used in the program design at any level.

We assume that the array A is implemented by a RAM memory component which can be contained in the unit PO.

At this point, we describe the algorithm by the microlanguage having the characteristics explained in Section 2.2.1:

Microprogram (= interpreter implementation)

0. IN $\rightarrow$ B, 0 $\rightarrow$ C, 0 $\rightarrow$ J, **goto** 1

1. **if** (J_{most} = 0) **then** {*zero* (A[J_{module}] – B) $\rightarrow$ Z, **goto** 2}
   else {C $\rightarrow$ OUT, **goto** 0}

2. **if** (Z = 0) **then** {J + 1 $\rightarrow$ J, **goto** 1}
   else {J + 1 $\rightarrow$ J, C + 1 $\rightarrow$ C, **goto** 1}

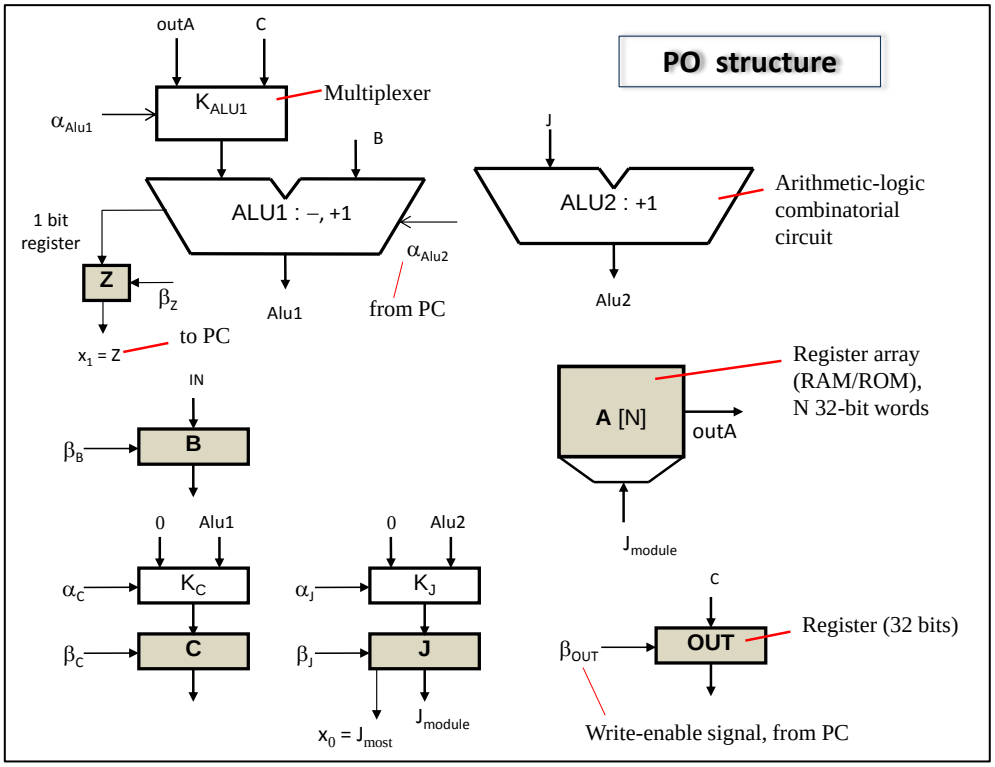
// to be completed with synchronized communications //

Let us assume that N = 32K. For the loop control we use a 16-bit register J, initialized to zero and incremented at each iteration; when the most significant bit J_{most} becomes equal to 1, we have executed all the loop iterations. The 15 least significant bits of J, denoted by J_{module}, are used to address the 32K memory.

The condition *if (A[J] == B)* has been implemented by computing *zero(A[Jmodule] – B)* $\rightarrow Z$ (ALU flag) and by testing Z in the next microinstruction. Alternatively, we could have tested *zero(A[Jmodule] – B)* as a condition variable in microinstruction 1, thus eliminating microinstruction 2. The effect would have been a sharp reduction of the number of clock cycles (one half) at the expense of a greater clock cycle size. Apart from didactic reasons, the choice of maintaining microinstruction 2 could have been motivated by additional design specifications for this particular unit (e.g., the clock cycle minimization).

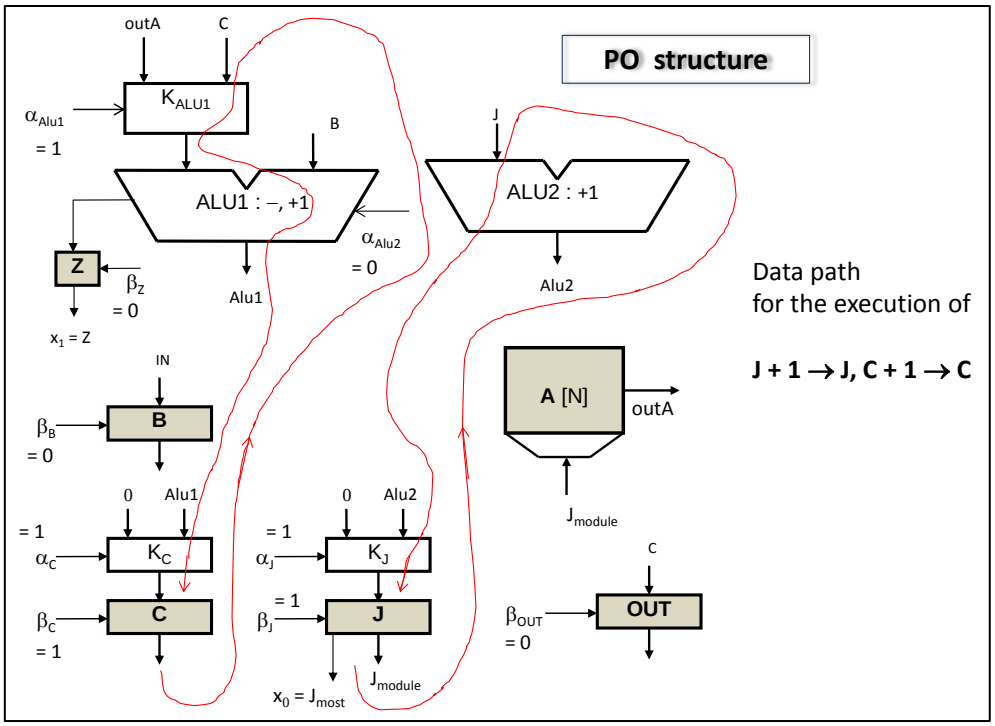
2.3.2 Operating Part and Control Part

The Operating Part, *formally* derived from the microprogram, is shown in the following figure, where all the links with the same symbols (e.g. Alu1 and C. which denotes the C register output) must be mutually connected to form the real graph.



The successive figure illustrates how the PO supports the execution of a certain micro-operation.

The PC design consists of two simple combinatorial circuits (implementing the output function ω_{PC} and the next state transition function σ_{PC}) connected to a 2-bit state register. Also such functions are derived *formally* from the microprogram structure, once the PO has been realized.



2.3.3 Clock cycle

Let $T_{ALU} = 5t_p$, $T_A = 5t_p$ (memory A access time), $\delta = t_p$.

We evaluate the clock cycle upper bound according to the formula:

$$\tau = T_{\omega PO} + max\,(T_{\omega PC} + T_{\sigma PO}, T_{\sigma PC}) + \delta$$

Both PC functions are implemented by 2-level AND-OR combinatorial circuits, thus

$$T_{\omega PC} = T_{\sigma PC} = 2t_p$$

Owing to the choice of the condition variables:

$$T_{\omega P0} = 0$$

$T_{\sigma Po}$ is equal to the maximum stabilization delay of the register assignment operations. Since the PO multiplexers are 2-level AND-OR circuits, $T_K = 2t_p$. The delays are the following:

- IN → B: *0*
- 0 → C: $T_K = 2t_p$
- 0 → I: $T_K = 2t_p$
- zero(A[J_{module}] – B) → Z: $T_A + T_K + T_{ALU} = 12t_p$
- C → OUT: *0*
- J + 1 → J: $T_K + T_{ALU} = 7t_p$
- C + 1 → C: $2T_K + T_{ALU} = 9t_p$

Therefore:

$$T_{\sigma P0} = 12t_p$$

$$\tau = T_{\omega PO} + T_{\omega PC} + T_{\sigma PO} + \delta = 12t_p$$

For $t_p = 0.05$ *nsec*, we have τ = 0.6 nsec, f = 1.66 GHz.

2.3.4 Performance evaluation

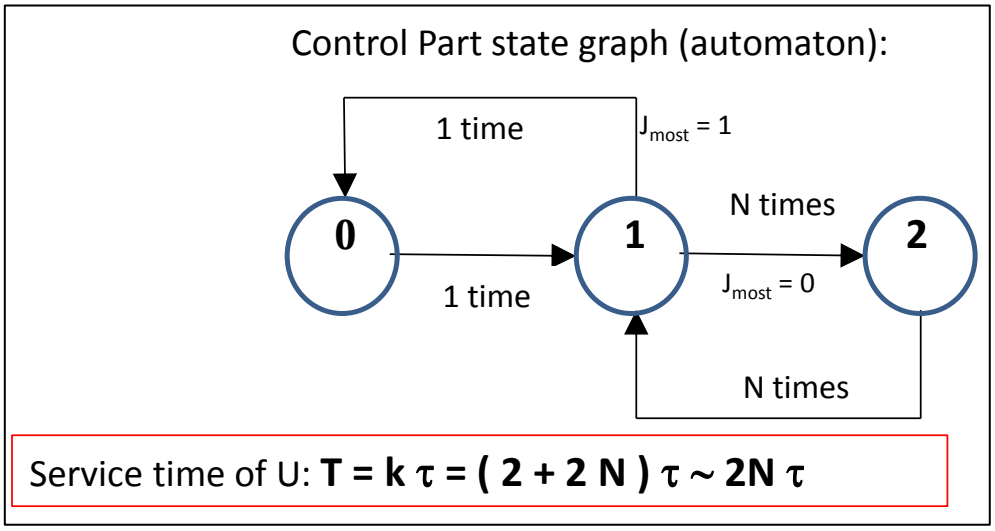
The performance metrics of interest are:

- Service time
- Processing bandwidth
- Latency
- Completion time

The ***service time*** (T) is defined as the average time interval between the beginning of two consecutive stream elements processing.

For a sequential module, this definition coincides with the ***latency*** (L), i.e. the *average time interval to complete the processing of one element*. However, in Part 1 we'll see that this coincidence is no more true for parallel computations.

The following PC state graph illustrates *how many clock cycles* are spent for service time *T*. Each arc is marked with the number of times it is traversed for each stream element processing:



For example, for N = 32K:

$$T \sim 39.3\ \mu\text{sec}$$

The **processing bandwidth** is defined as the average number of tasks (stream elements) processed per second, and it is evaluated as the inverse of the service time:

$$\boldsymbol{B = \frac{1}{T}\ tasks/sec}$$

In our example:

$$B \sim 25432\ \text{tasks/sec}$$

The **completion time** is defined as the total average time to process a stream of finite length m:

$$\boldsymbol{T_c = m\,T}$$

In our case, for $m = 10^6$, we have

$$T_c \sim 39.3\ \text{sec}$$

2.4 Communication at the firmware level

Communications play a central role in the design and implementation of systems at any level, in particular at the firmware level. In this Section we define some basic techniques for the efficient support of firmware communications, to be used in the design of realistic processing units. Such techniques will be used for the treatment of all the architectures studied during the course.

The communication protocols at the firmware level must have the following features:

i. no message is lost,

ii. the order of messages from the same source to the same destination is preserved.

Moreover, our protocols will be ***time-independent***, meaning that their correctness does not depend on the relative service times of the sender and the receiver unit, nor on the relationship between their clock cycles.

2.4.1 Communication channels and links

As at any level, several *forms of communication* can be distinguished. They are based on the way the ***communication channel*** is characterized, where the communication channel is the medium through which the communication is performed. A communication channel provides the resources for both

- the *transmission* of messages,
- the *synchronization* of partner modules (sources/senders, destinations/receivers).

Shortly, "communication = transmission + synchronization".

The communication forms are defined by the combination of two characteristics:

1. Symmetric *vs* asymmetric channels,

2. Asynchronous *vs* synchronous channels.

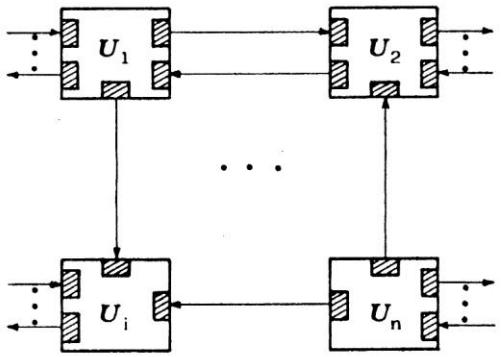
1. Symmetric vs asymmetric channels

In a *symmetric* communication we have one source module (the sender of messages) and one destination module (the receiver of messages). The term *one-to-one* communication is synonymous. Usually, symmetric channels are unidirectional.

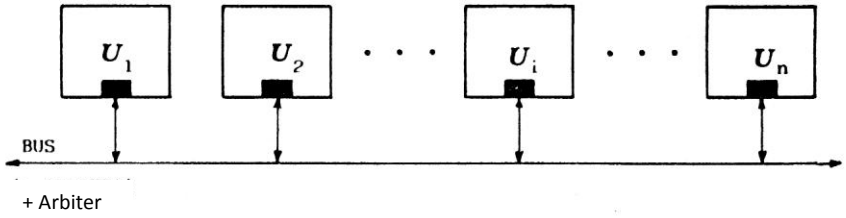
In an *asymmetric* communication we have more than one sender and/or more than one receiver, thus *many-to-one*, *one-to-many*, and in general *many-to-many* channels belong to the class of asymmetric communications.

The following figure shows the possible kinds of ***links*** used for the communication channels:

- the one-to-one unidirectional communication is realized through a ***dedicated*** link,
- a ***shared*** link, notably a *bus*, supports all the possible forms of asymmetric communications through one bidirectional cable connected to multiple units. One-to-one communication is supported by a bus too.



Dedicated links (one-to-one, unidirectional)



Shared link: bus (many-to-many, bidirectional)
It can support one-to-one, one-to-many, and many-to-one communications.

A message sent onto a shared link contains an explicit field with the *destination unit identifier* (i.e. name), which is explicitly tested by the destination. An *Arbiter* mechanism is associated to a shared link, in order to avoid that more than one message is simultaneously written. In case of simultaneous writing of more than one message, the effective value on the bus is the *bit-by-bit OR* of all the sent messages (a facility that is effectively exploited only in very particular cases).

Partner units are connected to the links through proper ***interfaces***, which contain the hardware resources necessary to support both transmission and synchronization. A distinct pair of interfaces is used for each dedicated link, while only one bidirectional interface (one input interface and one output interface) is used by each unit for a shared link.

In all the computer architectures of interest, links are *parallel*, i.e. each link consists of several wires onto which the bits of one or more words are transmitted simultaneously. Serial links are used mainly in local or geographical computer networks.

Pros and cons of dedicated *vs* shared links will be thoroughly discussed in Section 2.4.5.

2. Synchronous vs asynchronous channels.

The **asynchrony degree** of a channel is the maximum number of messages that the source can send without waiting for the first sent message being received.

Though not strictly necessary, it could help to think that the channel implementation adopts a FIFO queue for storing messages that have been sent but not yet received.

Asynchronous communications are characterized by asynchrony degree greater than one.

If the asynchrony degree is equal to *zero*, we speak about *synchronous* communications: in this case, a sort of "rendez-vous" is established between the sender and the receiver, meaning that the first partner ready to communicate must wait for the other partner being ready to communicate too.

In any (blocking) communication, the receiver must wait that at least one message has been sent. In a communication with asynchrony degree $k \geq 0$ the sender is blocked if, having already sent k messages and no one of them yet received, wishes to send a *(k+1)-th* message.

By asynchronous communication, the sender is able to free itself ("to decouple") from the receiver behavior, without being affected too much by the fluctuations in the service and latency times.

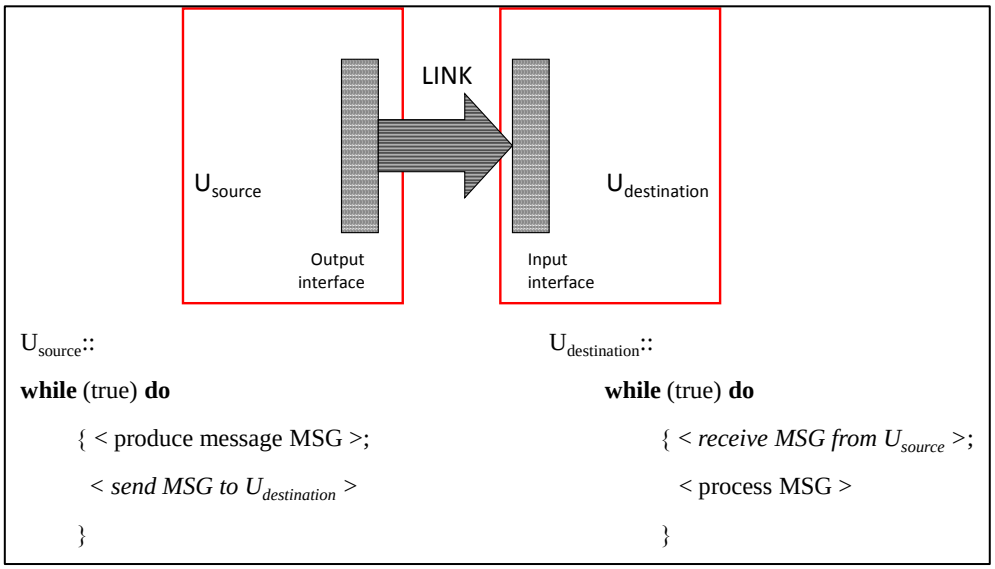
Of notable importance at the firmware level are the *asynchronous communications with k = 1*, since they do not require any special resource for buffering the sent message. On the other hand, at any level, such asynchrony degree is sufficient to substantially decouple the sender from the receiver behavior, especially when their average processing load is *balanced*, i.e. when their mean service times are equal or comparable: as it will be thoroughly studied in Part 1, load balancing is a characteristic that indicates a good quality of design.

2.4.2 Asynchronous communication on dedicated links

Let us study a firmware communication channel for symmetric communication through a dedicated link.

The next figure shows the basic behavior of both the partner units. The microprograms are infinite loops; the source calculates a value and send it to the destination, the destination receives a value and processes it. The nature of calculations is not of interest in studying the communication issues.

The interfaces of source and destination unit contain an *output register* OUT and an *input register* IN, respectively, connected by the unidirectional link. This is sufficient for implementing the message *transmission*: if and when the sender unit writes a value into OUT then, after the *transmission latency* time of the link, the value is stable at the input of IN, into which it will be written at the next clock pulse of the destination unit. Because we wish a time-independent communication, the writing into IN is enabled at *any* clock cycle (no explicit enabling control variable is used for IN).



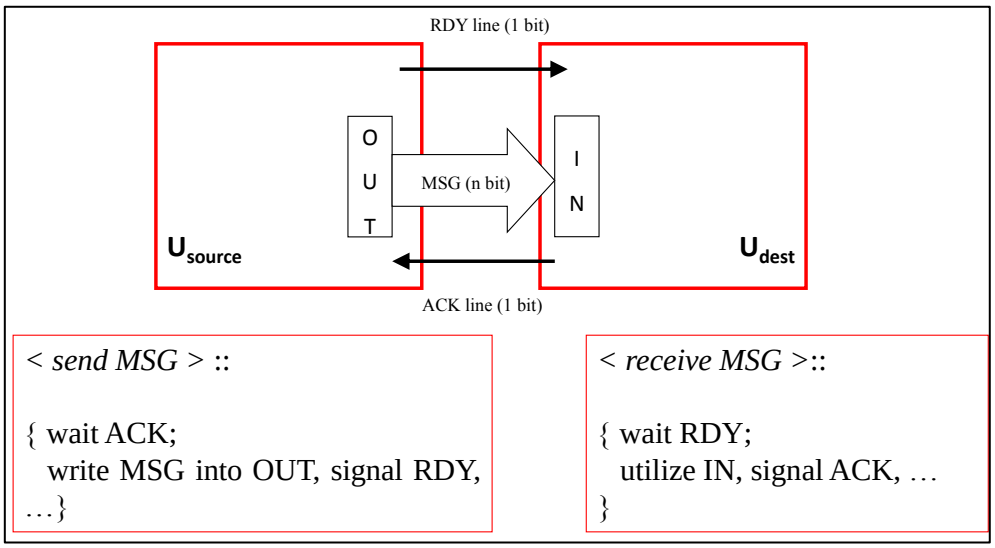
As said, transmission is not sufficient for implementing a communication: we need also a *synchronization* mechanism.

We implement an ***asynchronous channel with asynchrony degree k = 1***. Thus, the synchronization mechanism must be such that:

a) when the destination unit wishes to receive a message, it waits until a new value has been sent by the source unit;

b) after sending a message msg_i, the source unit continues its internal processing and, when it wishes to send a new message msg_{i+1}, it waits until the previous message msg_i has been received by the destination unit.

This synchronization requirements can be met by adding two 1-bit lines to the link: a *ready* line (RDY) from the source to the destination and an *acknowledgement* line (ACK) from the destination to the source:



The figure shows the algorithms of the source and destination units for implementing the protocol:

- in order to send a new message, the source waits the ACK signal, then writes the value into OUT, signals the "new_message" event through the RDY line, and goes on;
- *dually*, in order to receive a new message, the destination waits the RDY signal, then it can utilize the IN content, signals the "received_message" event through the ACK line, and goes on.

An inevitable timing assumption is necessary: if the RDY signal is produced in the same clock cycle in which the message is transmitted (as we'll usually do), then the latency of the RDY line must be strictly greater than the latency of the message lines. This condition, which is deterministic and easy to realize, does not contradict the time independent nature of our protocols.

For a correct asynchronous communication with $k = 1$, the RDY-ACK signaling must satisfy the following condition:

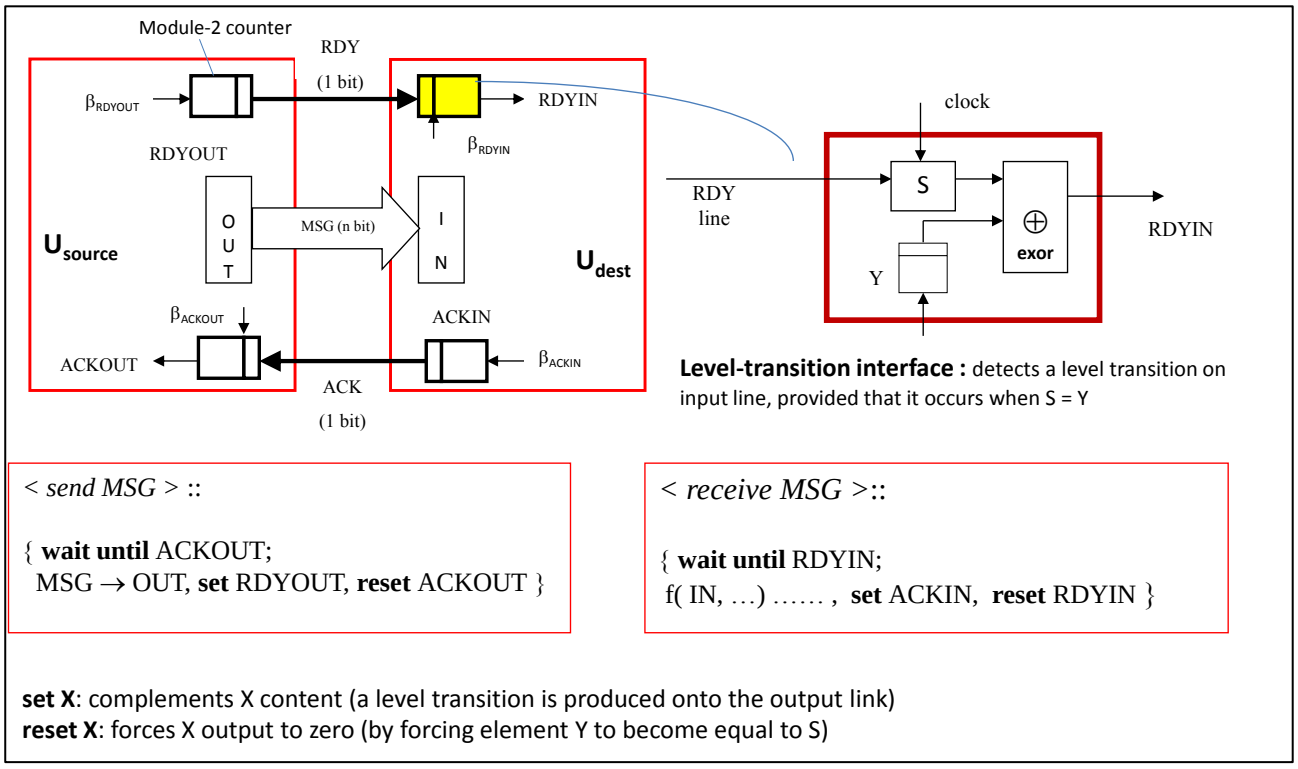
- *signaling correctness condition*: two consecutive RDY values must signal the presences of two distinct messages, two consecutive ACK values must signal the receptions of two distinct messages.

For example, the following intuitive solution does not satisfy the correctness condition: RDY = 1 (RDY = 0) means presence of a new message (absence of message), ACK = 1 (ACK = 0) means message received (no message received). To demonstrate that this is not a correct implementation, it

is sufficient to remember that a binary variable (like RDY and ACK) maintains its value until it is not explicitly modified by the producer of the value itself (*level* signals).

The correctness condition is satisfied by the so called ***level-transition interface***, in which any *transition* of the RDY or of the ACK value (from 0 to 1 *or* from 1 to 0) corresponds to a meaningful event. That is, if RDY = 1 signals a new message, then RDY = 0 signals the next new message. Analogously for ACK. Therefore, the *event generation* is simply realized by complementing (logical negation) the previous value, while for the *event detection* we need to implement a simple automaton which acts as a sequence recognizer (1 0 1 0 1 …).

The interface and its behavior are shown in the following figure:



Both the RDY and the ACK line is interfaced through a pair of *indicators*. In the figure they are denoted by RDYOUT for source and RDYIN for destination, and by ACKIN for destination and ACKOUT for source, respectively. They are not normal registers, instead,

- RDYOUT and ACKIN (event generation) are *modulo-2 counters*: each time the control variable β is = 1, the output is complemented;
- RDYIN and ACKOUT (event detection) are 4-state sequential circuits shown in the figure. For example, RDYIN contains
    - a normal register S to synchronize the input value from the RDY line with the clock cycle,
    - a modulo-2 counter Y,
    - a combinatorial circuit realizing the comparison (*exclusive-OR* ⊕) of S and Y. The output of the comparison is the RDYIN condition variable tested by the destination unit.

The *correct initialization* of the mechanism is: *RDYOUT = Y*. Starting from this situation, the presence of a new message will be correctly detected. In fact, with *RDYOUT = Y* we have RDYIN = 0, indicating that no new message is present. When RDYOUT will be complemented, S will

assume the RDYOUT value (after the transmission latency time). Now S ≠ Y, thus RDYIN becomes = 1, detecting the level transition: this indicates that a new message is present.

As soon as the destination unit has received the message, not only it signals the ACK event (by complementing ACKIN), but it also complements Y (by putting its enabling signal to 1). At this point, we are again in the initial condition, with RDYOUT = Y, thus RDYIN is = 0, denoting that no new message is present. According to this, the IN content is no more meaningful in the clock cycle immediately after the reception, since from now on a new message could be received (overwriting IN) and no assumption is done about the relative speeds of source and destination. Thus, if the IN value has to be used for more than one clock cycle, then IN must be saved into another internal register.

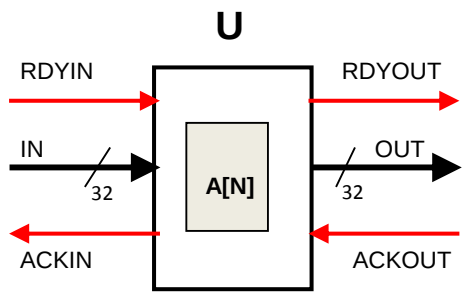
The figure shows the effective microprograms of the source and destination unit (f(IN, …) means any function utilizing the IN value): this is the real in/out protocol to be used for the unit design. The operation *set* and *reset*, used in the microprogram, are defined in the figure: respectively, they correspond to complementing an output indicator (RDYOUT or ACKIN) and to complementing the Y modulo-2 counter (in RDYIN or in ACKOUT).

Notice that all the protocol operations are executed in parallel in the same clock cycle (e.g. MSG → OUT, *set* RDYOUT, *reset* ACKOUT). Thus, an important result is achieved:

- *no overhead, i.e. no additional clock cycle, is paid because of the synchronization protocol*. That is, all the tests and synchronization operations are done in parallel with the tests and micro-operations corresponding to the computation of the processing unit.

Example

Let us complete the processing unit design of Section 2.3 by adding the synchronization for the input message and for the output message:



Specification ∀ IN: OUT = number_of_occurrences (IN, A)

```
int A[N]; int IN, OUT, B, C;
while (true) do
        { receive IN into B;
          C = 0;
          for (J = 0; J < N; J ++)
                  if ( A[ J ] == B) then C = C + 1;
          send C into OUT
        }
```

Microprogram

0. **if** (RDYIN = 0) **then** {*nop*, **goto** 0}
   else {reset RDYIN, set ACKIN, IN → B, 0 → C, 0 → J, **goto** 1 }

1. **case** (J_{most} ACKOUT = 0 –) **do** {*zero* (A[J_{module}] – B) → Z, **goto** 2}
   (J_{most} ACKOUT = 1 0) **do** {*nop*, **goto** 1}
   (J_{most} ACKOUT = 1 1) **do** {reset ACKOUT, set RDYOUT, C → OUT, **goto** 0}

2. **if** (Z = 0) **then** {J + 1 → J, **goto** 1}
   else {J + 1 → J, C + 1 → C, **goto** 1}

The *wait* condition of the protocol is simply realized by

{*nop, goto* current_microinstruction}

where *nop* denotes "no operation", i.e. no PO register is modified.

The *no-overhead* result can be verified: the receive protocol is "incarnated" in microinstruction 0, and the send protocol in microinstruction 1. No clock cycle is wasted wrt the original microprogram of Section 2.3.1. Since the protocol has no implication on clock cycle size (no delays are introduced by the protocol operation), the service time is identical to the version without synchronization.

More precisely, we point out that the ***ideal service time*** is not affected by the synchronization, where the ideal service time is defined as the service time of the processing unit considered *in isolation*.

Communications may affect the ***effective service time***, i.e the service time of the processing unit considered in the real context of the whole system. For example, U could be delayed if the destination unit (the unit to which the OUT result is sent) is a bottleneck. However, in no way this is due to the protocol.

These kind of evaluations about ideal *vs* effective service time will be done thoroughly in Part 1, where cost models for parallel computations at any level will be studied taking into account the communication impact too.

2.4.3 Other communication forms on dedicated links

The following communications forms are particular cases or variants of the basic level-transition method of the previous Section.

1) Synchronous communication

In some cases, the synchronous communication form can be more expressive than the asynchronous one, i.e. when some strong synchronization between the partners is needed.

A simple modification to the sender protocol is sufficient to transform an asynchronous communication into a synchronous one: the ACK is waited immediately after the message transmission, instead of before.

2***) Asynchronous communications with asynchrony degree k > 1***

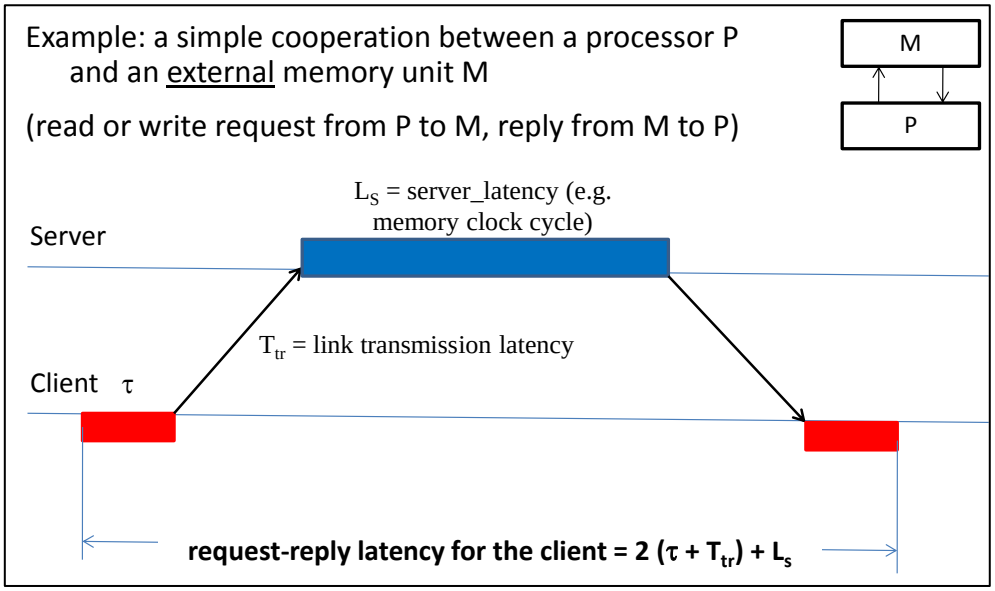
The level-transition interface is not able, in primitive form, to support an asynchrony degree greater than one.

The most general solution to this communication form at the firmware level is solved by emulation, that is by inserting, between the source and the destination unit, *a unit emulating a FIFO queue* of *k* positions. The interfaces between the source and the queue units and between the queue and the destination units are normal level-transition ones.

Alternatively, more direct solutions, without an additional queue unit, are possible in particular cases. They will be studied during the course when needed.

3) Request-reply behavior and link transmission latency

Let us consider a client-server configuration, where the client unit sends a request to a server unit and waits the reply without executing other actions during the service. The following timing behavior occurs:



Because in this simple structure there is no parallelism between client and server, the client latency for utilizing the server result is equal to

$$2\,(\tau + T_{tr}) + L_s$$

where L_s is the server latency and T_{tr} the link transmission latency.

The figure contains the example of the cooperation between a processor and an external memory: the client latency is the *effective memory access time* "as seen" by the processor.

The request-reply protocol is simplified wrt the general one, because we can eliminate the ACKs. For example, the processor microprogram fragment for a memory reading request is the following (the register names are self explanatory):

i. … memory_address → ADDR_INTERFACE, 'read' → OP_INTERFACE, set RDYOUT_MEM, **goto** i+1

i + 1. if (RDYIN_MEM = 0) **then** {*nop*, **goto** i+1} **else** {reset RDYIN_MEM, utilize (IN_DATA_INTERFACE, …}

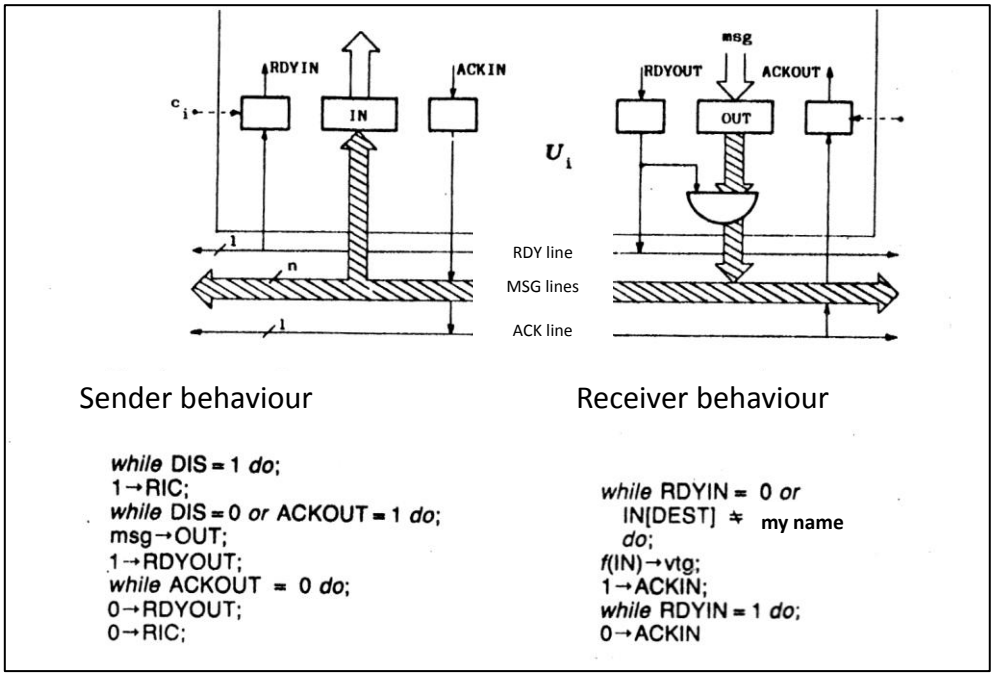
The value of the link transmission latency (T_{tr}) is of the order of some clock cycles for processing units on distinct chips. For example, for $T_{tr} = 5\tau$ and memory clock cycle $\tau_M = 100\tau$, the effective memory access time "as seen" by the processor is equal to 112τ.

As studied thoroughly in successive course parts, this discrepancy between processor and memory performances (and this range spreads year after year), also called *the Von Neumann bottleneck*, is one of the main motivations (probably *the* main motivation) for conceiving sophisticated architectures based on memory hierarchy (caching) and Instruction Level Parallelism.

Moreover, for units realized ***on the same chip*** (e.g. in a CPU: pipelined processor units, MMU, primary and secondary cache, interrupt unit, etc), we can assume that $\boldsymbol{T_{tr} = 0}$. This is another feature that will be exploited intensively in the definition of parallel CPUs.

2.4.4 Synchronous communication on buses

The following figure shows the interface and the send/receive protocols for a generic unit connected to a shared bus.



The bidirectional bus cable includes the message lines, one RDY line and one ACK line, shared by all the processing units. Bidirectionality is implemented by using distinct interfaces for sending and for receiving.

The protocol includes the interaction of the send operation with the Arbiter: RIC is the line for the request to the Arbiter, and DIS is the reply line from the Arbiter. The source unit *acquires* the bus by submitting the request (1 → RIC) to the Arbiter and waiting until the Arbiter has not given the authorization (DIS = 1). When the destination receives the message, the source unit *releases* the bus by informing the Arbiter (0 → RIC), and the Arbiter re-initializes the authorization line (DIS = 0).

The bus interfaces are *not* of the level-transition kind: it is easy to show that the *signaling correctness condition* cannot be satisfied (except in particular cases) with a shared link. The bus protocol is based on *values* (instead of value transitions) to encode the send-receive events. Thus, *normal* registers are used. However, the *re-initialization* of the synchronization mechanism must be done be explicit additional communications: when the source units detects ACKOUT =1, it puts RDYOUT = 0; the destination unit waits for RDYIN = 0 in order to put ACKIN = 0, after that the re-inizialization is completed.

In other words, only one half of the values transmitted onto the RDY and ACK lines correspond to "new message" and "received message" events: the remaining half values are used just for re-initializing the interfaces.

Because of this protocol, only the ***synchronous*** communication form can be supported by a bus. On the other hand, the source unit must wait the reception of the message in order to release the bus, thus asynchronous communications are conceptually impossible in primitive form.

All these consideration show that shared link prootocols are affected by a relevant *overhead*, differently from dedicated links.

Various kinds of arbiters exists. They are not of immediate utility for our purposes, and their description will be done during the course only when needed.

2.4.5 Interconnection solutions: dedicated *vs* shared links

High parallelism in processing and communication is allowed by dedicated links, since more links can be used simultaneously by different pair of units during the computation. Instead, a shared link could become a bottleneck since *at most one* communication at the time is possible. In addition to the limited parallelism, we have seen that the protocol and the arbitration mechanism are source of performance *overhead*. Moreover, *latency* is another advantage of dedicated link solutions. On the other hand, the advantage of shared links is the *cost*.

Let us discuss these issues in detail, since they are of special importance for the structured computer architecture principles and technology.

Let us consider two extreme solutions for the interconnection in a system composed of N processing units, namely the ***single bus*** interconnect and the ***crossbar***, i.e. *all-to-all*, interconnect using dedicated links only.

The crossbar solution is characterized as follows:

- *Number of links*: $O(N^2)$, since there is a dedicated link for each pair of communicating units;
- *Pin-count* per processing unit, i.e. the number of interface bit connections (called *pins*) on the processing unit perimeter: $O(N)$, since each unit has a dedicated interface for each in/out link;
- *Latency* per message, i.e. the time delay of a single communication: constant $O(1)$, since any message travels just one link (there is no message routing);
- *Maximum communication bandwidth*, i.e the maximum number of messages that can be communicated simultaneously in the system: $O(N^2)$, since any message is transmitted over a dedicated channel and there is no contention between units.

The single bus solution is characterized as follows:

- *Number of links*: constant $O(1)$, by definition;
- *Pin-count* per processing unit: constant $O(1)$, since each unit has only a bidirectional interface;
- *Latency* per message: $O(N)$, since the cable transmission delay is proportional to its physical length, thus proportional to the number of connected units. Moreover, the synchronous protocol and the arbitration contribute to increase the latency;
- *Maximum communication bandwidth*: constant $O(1)$, since at most one message is transmitted over bus.

Let us discuss the performance-cost comparison.

Provided that it is realizable, the crossbar solution has the maximum bandwidth and the minimum latency, while the single bus solution has the minimum bandwidth and the maximum latency. Of course, the *bandwidth* needs depend heavily on the effective *parallelism degree* of the computation implemented by the N-unit system. If such parallelism is low ($<< N$), than the crossbar solution is redundant, while the single bus solution might be sufficient. A notable example is the *I/O interconnect* in uniprocessor computers, which traditionally has been implemented by a single bus (e.g. PCI): this is because the simultaneous communication of CPU/M to/from distinct I/O units is a rare event. However, the bus bandwidth and latency becomes a real problem for *massive* I/O data communication, especially in advanced applications: for example, a highly parallel coprocessor (e.g., a GPU) could be inefficiently exploited when the amount of exchanged data is very large compared to the processing requirements (e.g. in some graphical or mathematic applications with light calculation functions).

On the other hand, the crossbar has the maximum cost and the single bus has the minimum cost.

The cost issue deserves a special attention. At the firmware level, the communication cost depends mainly upon two factors:

a) *Cost of the links per se*: with any kind of technology, links are expensive and waste physical space on cards and boards;

b) *Pin-count*: in an integrated realization of a processing unit, or collection of units (e.g. a CPU), on a single VLSI chip, the number of bit connections for the external interfaces must be bounded in order to minimize the chip area and to adapt it to the functionalities and resources of the unit(s). Notice that a linear increase of the chip perimeter (e.g. adding a further word to an interface) implies a quadratic increase of the chip area. If the area is "too large" compared to the space occupied by the integrated components, than the clock cycle increases rapidly with the component size and the wire length. In our notation the t_p value, which includes the delays of wires connected to a gate, increases. Currently, a good VLSI technology is characterized by a component size of the order of fractions of nanometers: the corresponding t_p value is of the order of 10^{-2} *nsec* and the *wire delays are negligible* compared to the net delay of the gate. However, in order to really exploit the sub-nanometer technology, it is mandatory that the physical components are "well and regularly packed". Typical pin-count values for modern CPUs is of the order of some hundreds / one thousand.

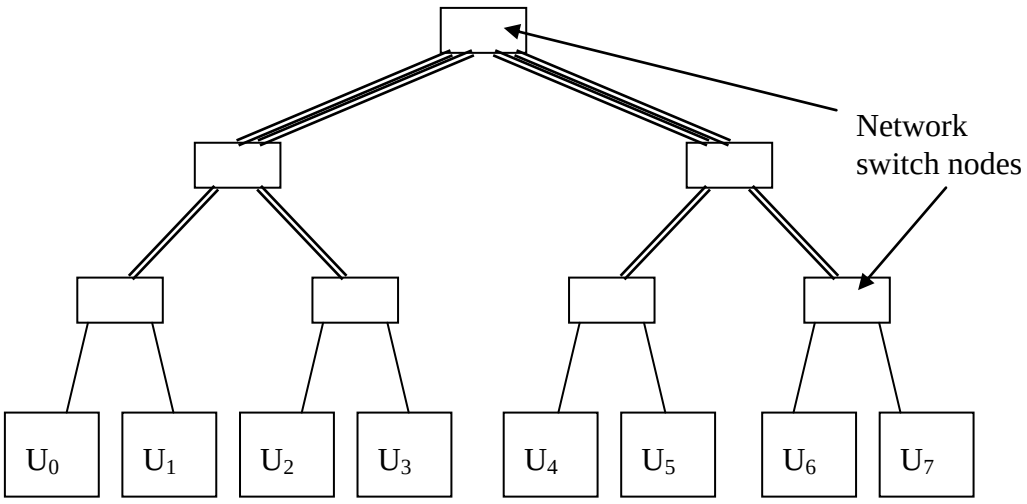
As seen, the single bus has low link cost and limited pin-count independently of *N*. The crossbar solution could be impracticable for high values of *N*, because of the quadratic explosion of the wire amount and of the linear explosion of the pin-count.

In conclusion, *both extreme solutions are suitable only for low degree of parallelism*: the single bus for performance reasons, the crossbar solution for cost reasons.

Limited degree networks

In Part 2 of the course, we'll see that powerful interconnection structures for parallel and distributed architectures avoid buses and rely on *dedicated* links only, provided that they are connected in proper topologies able to minimize the pin-count, to sharply reduce the wire amount, and, at the same time, to achieve high communication bandwidth and low latency. They are called *limited degree interconnection structures*. Notable structures, studied in Part 2, are cubes, butterflies, and fat trees.

Just to give an example, consider *N* units connected as the leaves of a binary *tree*, whose root and intermediate nodes act as *switching units (routers)*, as shown in the next figure:



A message from U_i to U_j travels through the subtree whose root is the common ancestor of U_i and U_j. The distance to be traveled is proportional to the subtree depth. Thus, in general, the *latency* is

$$O(\log N)$$

which is a very satisfactory results for high N (perhaps the best if the constant latency is impossible to be achieved).

The structure is composed of dedicated links only.

The number of links is $O(N)$ and the pin-count per node is very limited and constant ($O(1)$).

In theory the bandwidth is very high. In practice, according to the communication patterns to be supported for the parallel computation, the bandwidth might be severely limited by the conflicts in the network nodes and links.

A better solution, with a bandwidth close to the ideal one, is the so called *fat tree*, where the link capacity increases passing from a tree level to the next one. This is the solution shown in the figure, by anticipating that proper implementations exist that are able to double the link capacity at each level without jeopardizing the pin count.

All the most recent commercial interconnection networks are based on the fat tree principle (e.g. *Infiniband*) or on other limited degree networks.

Moreover, the trend is to adopt limited degree networks for the uniprocessor *I/O interconnect* too, replacing traditional buses.

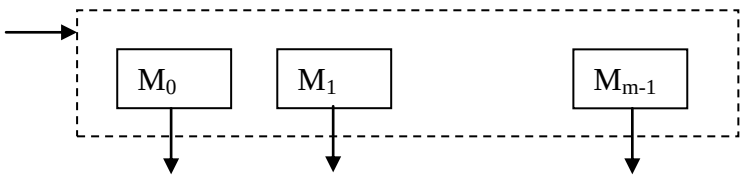
Finally, in Part 2 we'll see that the *multicore* technology tends to exploit powerful *on-chip* limited degree networks.

2.5 Modular memory organization

In many parts of this course we'll need to utilize a *high-bandwidth memory*.

The memory bandwidth is the (average) number of words read/written during the access time latency. A traditional memory, realized as a single processing unit, has relative bandwidth equal to one.

Higher bandwidth are achieved introducing the concept of *modular memory*. More than one identical memory units (m) are provided, able to operate in parallel:



In principle, the ideal bandwidth is equal to m.

Modular memories can be organized in different manners, depending on the way the *addresses are distributed among the modules*.

In a ***modular sequential memory***, addresses are distributed sequentially in the same module: if γ denotes the capacity of a single module, than address i belongs to module j with

$$j = i \ \mathbf{div} \ \gamma$$

The address configuration is the usual one for block-structured information:

module	address inside the module

For example, if the whole capacity is 4G and $m = 8$, then $\gamma = 512M$ per module. Thus the 3 most significant bits identify the module and the remaining 29 bits represent the address inside the module.

In a ***modular interleaved memory***, addresses are distributed sequentially from one module to the other in a circular way: address *i* belongs to module *j* with

$$j = i \ \textbf{mod} \ m$$

The address configuration is now:

address inside the module	module

For example, if the whole capacity is 4G and $m = 8$, then the 3 least significant bits identify the module and the remaining 29 bits represent the address inside the module.

The sequential organization is used mainly for *expansion* reasons. Only under particular conditions we can exploit parallelism among the memory modules (e.g. we need to operate simultaneously on information of address $i, i + \gamma, i + 2\gamma, \ldots$). Often, the memory bandwidth remains one or low.

The interleaved organization is very interesting from the point of view of parallelism. The best, and very important indeed, case is the *by-block access*: if *m* consecutive words are to be read or written then they can be accessed in parallel during the same access time. In this case, the bandwidth is equal to the ideal one, i.e. *m*.

In other cases, for a sequence of requests in which addresses are not consecutive, the bandwidth is $< m$ because some request could refer the same module, thus they must be served sequentially.

Interleaved memory will be intensively used for the cache architecture (Section 5) and for shared memory multiprocessors (Part 2).

2.6 Exercises

1. Prove the following assertion in general and by examples: the formula of Section 2.3.3 for clock cycle evaluation gives an upper bound (i.e. specific microprograms exist that can be executed correctly with a clock cycle lower than the value computed by the formula).
   What is the advantage to use an upper bound formula, instead of evaluating the clock cycle more exactly?

2. Design a processing unit U so defined: contains a memory component M of 4K 32-bit words; receives a binary operation code OP, an address IND of M, a 2-bit value J, and a 32-bit value X; if OP = 0 sends to a 8-bit output OUT1 the value of the J-th byte of M[IND]; if OP = 1 sends to a 13-bit output OUT2 the result of the binary search of X in M: the address of M location equal to X or the value -1 if X is not present in M.

   Use transition level interfaces, assuming that the input message (OP, IND, J, X) is received by unit U1 and the results OUT1, OUT2 are sent to unit U2.

   Use transition level interfaces, assuming that the input message (OP) is received by unit U11, the input message (IND, J) is received by unit U12, the input message (X) is received by unit U13, the results OUT1 is sent to unit U21 and the result OUT2 is sent to unit U22.

3. The assembler machine and its basic interpreter

This Section studies the issues of the assembler machine and of the associated firmware architecture. The assembler level will be exemplified using *a Didactic RISC (D-RISC)* which captures the main features of current RISC machines (e.g., MIPS, PowerPC).

As far as the CPU is concerned, the firmware architecture studied here will be a very simplified one (no caching, no instruction level parallelism), just in order to show the basic concepts and to have an idea of the performance metrics. Section 5 and Part 1 will develop the advanced CPU issues in detail, while other issues – I/O, interrupts and exceptions – are studied in a definitive form in the present Section.

We start this Section with a necessary definition of logical address space and virtual memory, since this issue is fundamental to introduce the assembler machine and its interpreter. The virtual memory techniques and implementation will be studied thoroughly in Section 4.

3.1 Logical address space and virtual memory

In Section 1.2 we saw that an application is compiled into a collection of one or more processes. During the compilation of a process P, the ***virtual memory*** of P (VM_P) is built: VM_P is an abstraction of the memory of P, containing all the objects (instructions and data) referred by this process. The set of addresses of the virtual memory is called the ***logical address space***. In a linear virtual memory, this space start at the logical address zero.

For example, consider the simple process of Section 1.3.1:

```
int A[N], B[N]; int x = 0;
        for (i = 0; i < N; i++)
                x = A[i]*B[i] + x
```

The compiler allocates the process VM by choosing the base addresses and taking into account of the address range for every object.

The VM of this process is shown schematically in the next page.

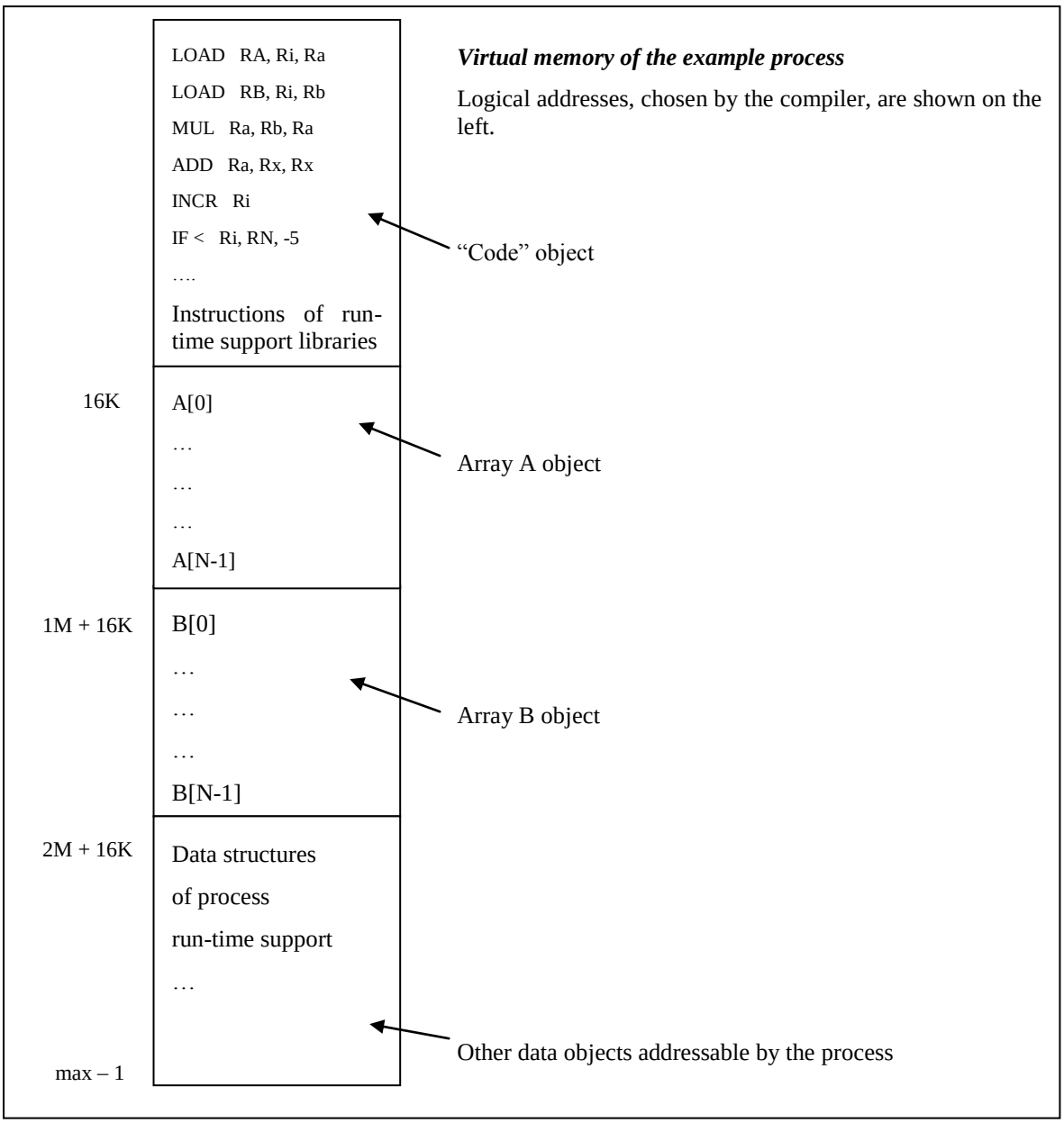
In this example, the "code" object (instructions) consists of the first six words of VM starting at the logical address zero (the assembler instructions introduced informally in Section 1.3.1 have been used; starting from Section 3.2 the assembler machine will be defined thoroughly). Let N = 1 Mega. Because an integer is usually represented by a word (e.g. 32 bit), two subspaces of 1 Mega addresses are allocated to A and to B, for example starting at addresses 16K and (1M + 16 K) respectively. Integer variables *x* and *i* are not allocated in VM if the compiler chooses to implement them by general registers.

As shown in the figure, according to Section 1.2 (many) other objects are *linked* to the process VM at compile time, i.e. all the instructions and data belonging to the run-time support of the process. For the moment being, it is sufficient to reason about the programmer-visible objects.

At run-time, when the instructions are effectively executed, *the processor generates logical addresses*:

- every assembler instruction is uniquely identified by a logical address (from zero to five in the example), contained in a register called the *program counter* (or instruction counter, IC), properly updated in the microprogram interpreting the instruction,

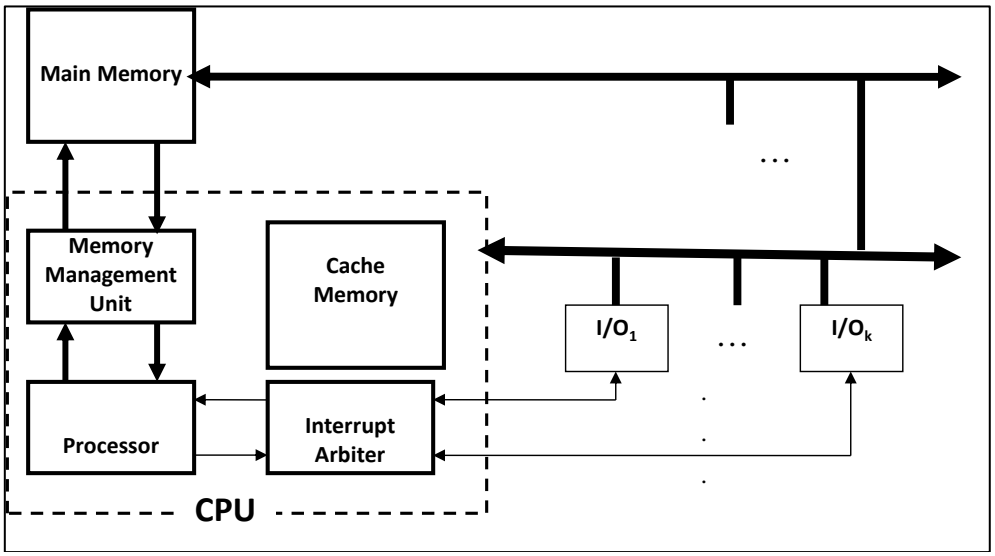
- the addresses computed by the two LOAD instructions are logical addresses too, belonging to the range of A and B respectively.



Notice that some locations of VM are *initialized* at compile time. In our example, all the instructions (the binary representations of the instruction encodings) are initialized in VM. Arrays A and B could be initialized or acquired at run-time by some external medium: in the latter case, the 2-Mega logical address space for A and B, though it is unspecified ("don't care" values) at compile time, *exists all the same,* because the addresses in this range are just the addresses that will be generated at run-time, thus must be different from the addresses of other objects.

In order to be executed, the process must have been loaded into the ***physical memory***, i.e. the main memory M, during a previous phase of the process life cycle. We will discuss these phases in detail in Section 4. The correspondence between logical addresses and physical addresses is established by the ***process relocation***, or ***address translation*, *function***. This function is applied at run-time for every memory access.

Often, the firmware architecture contains a processing unit, called *MMU* (Memory Management Unit) dedicated to the efficient execution of the address translation function (only one clock cycle latency):



The processor is not aware of the distinction between logical and physical addresses. It has the illusion to be connected to the virtual memory. In fact, each memory access request, containing a logical address, is intercepted by MMU which generates the physical address (if defined) and forwards the request, modified with the physical address in place of the logical one, to the main memory. For modularity and control reasons, in general the memory reply is intercepted by MMU and forwarded to the processor.

If the dynamic allocation of the main memory (studied in detail in Section 4) is adopted, the address translation function may be undefined for some logical addresses, meaning that the referred information are not currently allocated in main memory. In this case, MMU returns a *memory fault exception* to the processor. An ***exception*** is an event, *synchronous* with the current behavior, that requires a proper handing to reach again a consistent state of the computation, if possible. Other exceptions signaled by MMU are *protection violation* (protection checks are done concurrently with address translation), and various *faults/errors of memory or links*. Exceptions can occur during arithmetic operations, e.g. overflow. In all the mentioned cases exceptions are generated at the firmware level. Exceptions at the assembler level, or at the application level, are equally possible (some high-level languages provide proper constructs to detect and to handle them).

There is no a-priori relation between the virtual memory size and the main memory capacity. The maximum VM size of a process is established by the logical address length: for example, if the assembler machine provides 32-bit logical addresses, then the maximum size of a process cannot exceed 4G words; for larger processes a compile-time error is generated.

The (maximum) capacity of M could be less or equal to the maximum VM size or, better and more usually, it is greater or even much greater, in order to support the simultaneous allocation of several processes in main memory. For example, the firmware architecture of M and of MMU could provide 40-bit physical addresses, meaning that the main memory capacity is expandable within 1T words (T = Tera = 2^{40}).

Though defined as an abstraction, the process virtual memory VM_P is a concrete entity in the system. VM_P, considered as the final result of the compilation, is a ***file*** (the so-called executable or

object file of the program) which, as any other file, is stored into a permanent memory medium (a disk). When the process will be created, this file will loaded (at least in part) into the main memory.

Process life cycle and virtual memory will be studied thoroughly in Section 4.

3.2 The assembler machine

3.2.1 Instructions and variables

As introduced in Section 1.3, in a Von Neumann stored-program computer the assembler level is characterized by an imperative language, thus based upon the concepts of

1. Variables and assignment,
2. *Explicit sequencing* of instructions.

Instruction sequencing is implemented by a special variable, called ***program counter*** or instruction counter (IC), which contains the current instruction address and is properly updated by every instruction according to the instruction semantics. In some machines IC is explicitly visible at the assembler level (it can be manipulated as any other variable), in others it is visible at the firmware level only.

Variables of a process can be allocated in:

a. the ***virtual memory*** of the process, thus accessed in main memory during the process execution,
b. general registers.

Variables in memory are characterized by a *logical address*. Various *addressing modes* exist for referring a variable. They will be reviewed in the next section.

General registers can be thought of as extensions of the process memory. They are characterized by absolute physical addresses. Some machines have very few general registers, others have some tens – some hundreds, organized as linear arrays. For example:

RG[64]

denotes an array of 64 general registers. "General" means that such registers are used not only for operands of arithmetic operations or predicates, but they are also used for other instruction features, notably for *computing logical addresses* of objects.

General registers constitute a very efficient resource for program optimization. Accessing a general register has no overhead: general registers are visible at the firmware level too, thus register access is merely part of microinstructions. Therefore, the compiler tries to statically or dynamically load program data in general registers in such a way that they remain allocated there as long as possible, On the other hand, register addressing is very cheap because of the low size of RG array: in the previous example, a register address is 6-bit wide. In some programs, it is possible that RG size is insufficient: in this cases, the code generated by the compiler contains *explicit* data transfers to/from proper memory areas, according to some global optimization strategies.

A variable stored in a *word*, usually 32-bit or 64-bit wide, correspond to the basic *integer* type. Other elementary types are represented by parts of a word or by more words, e.g., a character as 8-bit byte, a short integer as half word, a long integer as double word, a real as double word or by quad-word, and so on.

By convention, the word length is just assumed to be the number of bits to represent the basic integer type. There is no direct relation between the word length and the instruction length or the

logical address, though in RISC machines instructions and logical addresses are of fixed length and usually one-word wide, while in CISC machines often instructions and logical addresses are of variable length (one or more words).

An instruction is represented by a binary ***instruction format***, in which the component elements are encoded by proper fields:

- operation code,
- logical addresses or, more often, information to compute logical addresses, of data and of next instruction,
- possibly, immediate operand values,
- other information and annotations.

According to such elements, the basic semantics of an instruction can be described as follows:

{ execute the actions corresponding to operation code and other fields;

update IC to the logical address of the next instruction;

activate other actions (procedures) for event handling, both synchronous events (*exceptions*) and asynchronous events (*interrupts*) }.

Operations associated to instructions are of various kinds. As introduced in Section 1.3.2, usually, in a RISC machine they are basic arithmetic-logic operations and basic predicates, while in a CISC machine more complex operations are provided: some of them are complex arithmetic-logic (e.g. trigonometric, square roots, logarithm, graphics, array operations, etc.) operations and predicates, as well as instructions corresponding to higher-level commands or their parts (e.g. synchronization primitives, data transfers, etc). This issue will be developed during various parts of the course.

3.2.2 Addressing modes

In a limited number of cases, the values of (short) instruction operands are contained directly in the instruction format. We speak about ***immediate*** operands.

Except this case, operands are referred to by their ***addresses***: a logical address for a variable in memory, an absolute physical address for a variable in a general register.

a) Absolute addressing

The address is a *constant* contained in the instruction format and fixed by the compiler.

Absolute addressing is always used for *general registers*. For example,

ADD R3, R18, R7

Has the following semantics for the operation actions:

$$RG[3] + RG[18] \rightarrow RG[7]$$

For variables in memory, the absolute addressing mode not only wastes space in the instruction format, but also it is inflexible, since in many cases addresses have to be computed by program.

b) Indirect addressing via general registers

General registers are a very efficient resource to contain addresses to variables in memory, or to instruction to which the control has to be transferred. Some instructions contain fields with the absolute addresses of one or more registers, for example:

ADD_MEM R3, R18, R7

means:

$$VM[RG[3]] + VM[RG[18]] \rightarrow VM[RG[7]]$$

where *VM[x]* means the content of the virtual memory location whose address is the value *x*.

Addresses are initialized in general registers or, more in general, addresses are computed by instructions sequences and written into general registers. Thus, we achieve full flexibility at the very low cost of instruction encoding (e.g. instructions ADD and ADD_MEM occupy the same space, notably one word).

c) Indirect addressing via memory locations

Pointers are a typical program structure that requires that memory locations contain addresses to other locations where the desired information is stored. For example, while the instruction with indirect addressing via registers:

LOAD R9, R31

means

$$VM[RG[9]] \rightarrow RG[31]$$

i.e., the content of virtual memory location, whose address is contained in RG[9], is written into RG[31],

the instruction

LOAD_INDIRECT R9, R31

has the following effect:

$$VM[VM[RG[9]]] \rightarrow RG[31]$$

i.e., the virtual memory location, whose address is contained in RG[9], contains the address of another location whose content is written into RG[31].

Indirection may be iterated several times, e.g. VM[VM[VM …] …], as required by the implementation of some complex linked data structures (*more indirection levels*).

Indirect addressing via memory – especially with more indirection levels – is primitive in several CISC machines, while usually RISC machines *emulate* indirect addressing via memory with a sequence of simpler instructions without indirection.

d) Base plus index addressing

This addressing mode is particularly suitable for the manipulation of arrays. Using indirect addressing via registers, two registers are used: one containing the *base* address of the array, the other for the *index*. The address is given by the sum of the content of the base and of the index register. For example:

LOAD R10, R15, R50

means:

$$VM[RG[10] + RG[15]] \rightarrow RG[50]$$

An alternative is the refer the base and the index through memory locations, whose address can be obtained in one of the other addressing modes: in the simplest case, base and index registers refer two memory locations whose sum is the address of the element.

e***) Relative addressing***

Apparently similar to base plus index, the relative addressing mode is used, for any type, when we wish to compute an address relatively to another one. A notable case, in branch or jump instructions, is *addressing of target instructions relatively to IC*. For example:

GOTO offset

where offset is a *signed* constant, has the following effect:

$$IC + offset \rightarrow IC$$

The offset value can be shorter than the word, so that the instruction can be encoded in a single word.

f) Stack addressing

When stacks are used to organize data, e.g. in nested function calls, the instruction operands can be found implicitly at the top and top-1 positions of the stack. Instruction Push and Pop are convenient to manipulate the top position.

3.3 D-RISC: a Didactic RISC assembler machine

3.3.1 Instruction set

The main characteristics of D-RISC are:

- 32-bit word: this is also the size of all instructions and of logical addresses;
- 64 general registers RG[64] visible at the assembler level, while the program counter IC is directly manipulated at the firmware level according to the semantics of the various instructions;
- 256 distinct operation codes. The 8 most significant bits of any instruction represent the operation code;
- virtual memory addressing modes are *base plus index via registers* (for data), *relative to IC* (for target instructions), and *indirect via registers* (procedure call and some jumps) Some special instruction use different modes, however always based on simple indirect addressing via register. All the other addressing modes must be emulated by program.

The classes of instructions are:

1. Arithmetic-logic operations, with the operand values in general registers contents only,
2. *Data transfers* between virtual memory and general registers (LOAD, STORE),
3. *Branch* (IF) and *jumps* (GOTO, CALL),

4. *Special instructions*, corresponding to specific requirements of the process run-time supports and/or to optimizations for various classes of architectures (cache utilizations, pipelining, multiprocessor).

From now on, general registers will be referred by the *key-word R* denoting "register address", e.g. R5 means the general register of address 5. Moreover, for better readability of programs, register addresses will be identified by intuitive *symbols* corresponding to their contents and semantics, e.g. Rbase_A could be used to denote the general register containing the base address of array A. In fact, Rbase_A will be a specific register (e.g. R5) allocated by the compiler.

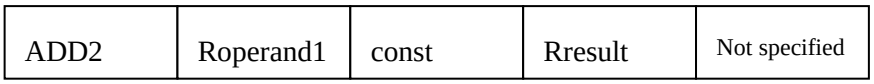
Arithmetic-logic instructions include all the most common operations on integers (ADD, SUB, MUL, DIV, MOD, etc) and booleans (AND, OR, NOT, EXOR, etc). The instruction format is

- ADD Roperand1, Roperand2, Rresult

Monadic operations are provided for convenience: INCR Rdata, DECR Rdata, CLEAR Rdata (putting zero into a register), as well as *data transfers between registers* (MOV Rsource, Rdestination)

Immediate operands are allowed, and recognized by the absence of the key word R, provided that they are represented by 6 bits, e.g.

- ADD Roperand1, const, Rresult



Notice that this instruction has a different binary opcode wrt the normal ADD.

Load and Store instructions are used, respectively, to copy the contents of a virtual memory location into a register, or to transfer a register content into a virtual memory location:

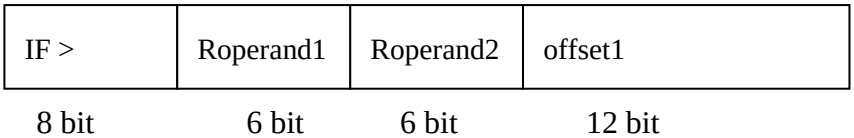
- LOAD Rbase, Rindex, Rdestination
- STORE Rbase, Rindex, Rsource

They use *base plus index addressing via registers*. The instruction format has the same fields of arithmetic-logic instructions.

Immediate index, represented by 6-bit signed integer, is allowed.

Conditional branch instructions use *IC-relative addressing* with constant offset. *Predicates operate on general register contents only*. The most common predicates (=, ≠, >, <, ≥, ≤) are provided for diadic predicates, as well for monodic predicates wrt zero (= 0, ≠ 0, > 0, < 0, ≥ 0, ≤ 0):

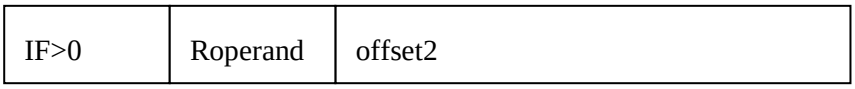
- IF > Roperand1, Roperand2, offset1



where *offset1* is a 12-bit signed integer constant. The semantics (without interrupts and exception treatment) is:

if RG[Roperand1] > RG[Roperand2] *then* IC= IC + offset1 *else* IC = IC + 1

- IF > 0 Roperand, offset2



where *offset2* is a 18-bit signed integer constant. The semantics (without interrupts and exception treatment) is:

if RG[R1] >0 *then* IC=IC+*offset2 else* IC = IC + 1

Unconditional branches (jumps) provide IC-relative addressing or indirect register addressing for the target instruction:

- GOTO offset3 meaning: IC + offset3 → IC

- GOTO Rtarget meaning: RG[Rtarget] → IC

For procedures D-RISC provides *the CALL instruction,* that merely causes the branch to the first instruction of the procedure and saves the return address. All the parameter passing actions are left to the program explicitly.

CALL Rprocedure, Rreturn

meaning:

IC + 1 → RG[Rreturn], RG[Rprocedure] → IC

That is, the procedure returns the control to the main program at the instruction address immediately following the CALL. Usually, the last instruction of the procedure is

GOTO RG[Rreturn]

It is worth to note that the procedure is just a part of the process containing the calling program, thus it is executed *in the same logical addressing space*. This is true for process run-time libraries too (e.g., a send-message procedure or a low-level scheduling procedure).

Special instructions

These instructions are listed here for completeness. However, except some simple cases, they will be motivated and explained in specific parts of the course where they are used.

a) Null Instruction: NOP

b) Process termination: END

c) Last instruction of context-switch: START_PROCESS Rtabril, RIC

d) Explicit saving of IC: COPY_IC Rsave

e) Enabling/disabling interrupts:

- Interrupt masking: MASKINT Rmask
- Interrupt disabling: DI
- Interrupt enabling: EI

f) Annotations for interrupt enabling/disabling in any instruction.

For example:

ADD Ra, Rb, Rc, DI

LOAD RA, Ri, Ra, EI

g) *Busy waiting of interrupts:* WAITINT Ri/o, classe, Ra, Rb

h) Control of indivisible sequences on multiprocessor shared memory:

SET_INDIV Rbase, Rindice

RESET_INDIV Rbase, Rindice

i) Load/Store annotations for indivisible sequences on multiprocessor shared memory.

For example:

LOAD Rsemlock, 0, Rtemp, setindiv

STORE Rsemlock, 0, R0, resetindiv

l) Annotations in branch instructions for Delayed Branch in pipelined CPUs. For example:

IF < Ri, RN, LOOP, delayed_branch

m) Annotations in Load/Store instructions for cache management:

m1) Prefetching a block

m2) Don't deallocate a block

m3) Explicit deallocation of a block

m4) Explicit re-writing of a block into main memory

3.3.2 Compilation rules

In this Section compilation rules for the most common high-level constructs are defined and exemplified in D-RISC. Program compilation is done by *composition* of compilation rules.

Arithmetic expressions

No special rule except the operator precedence ones. In D-RISC expression operands have to be LOADed into temporary general registers, and the final results STOREed into memory, if required.

Conditional and iterative commands

Let us denote by C a predicate (guard), by $\mathcal{B}$ a command or a sequence of commands, by *compile*($\mathcal{B}$) the compilation of $\mathcal{B}$, by IF_C LABEL and IF_N_C LABEL branch instructions with predicate C true and false respectively. For example, if C is the "=" predicate, then an example of IF_C LABEL could be

IF= Ri, Rj, LABEL

and an example of IF_N_C LABEL:

IF≠ Ri, Rj, LABEL

Compilation rules:

compile(**if** C **then** $\mathcal{B}$) ≡

```
                    IF_N_C  CONT
                    compile(B)
            CONT:   …
```

compile(**if** C **then** $\mathcal{B}1$ **else** $\mathcal{B}2$) ≡

```
                    IF_C    THEN
                    compile(B2)
                    GOTO CONT
            THEN:   compile(B1)
            CONT:   …
```

compile(**while** C **do** $\mathcal{B}$) ≡

```
            LOOP:   IF_N_C  CONT
                    compile(B)
                    GOTO LOOP
    CONT:           …
```

compile(**do** $\mathcal{B}$ **while** C) ≡

```
            LOOP:   compile(B)
                    IF_C    LOOP
    //continuation//  …
```

About iterative constructs, when the compiler is sure that at least one iteration will be executed, it replaces the ***while do*** with a ***do while***. The ***for*** construct is very often compiled as ***do while***, as an example when an array is scanned.

The compilation rules represent simple examples of optimizations, along with the proper choice of addressing modes and registers in order to minimize the number of memory accesses (see the example in Section 1.1.1).

Much more architecture-dependent optimizations will be studied in the various parts of the course.

Example

Let us compile the example of Section 1.1.1 into D-RISC according to the compilation rules:

```
/program 1/

int A[N], B[N];
for (i = 0; i < N; i++)
        A[i] = A[i] + B[i];
```

```
/program 2/

int a; int B[N];
for (i = 0; i < N; i++)
        a = a + B[i];
```

/compiled program 1/

```
LOOP:   LOAD   RA, Ri, Ra
        LOAD   RB, Ri, Rb
        ADD    Ra, Rb, Ra
        STORE  RA, Ri, Ra
        INCR   Ri
        IF <   Ri, RN, LOOP
```

/compiled program 2/

```
        LOAD   RA, 0, Ra
LOOP:   LOAD   RB, Ri, Rb
        ADD    Ra, Rb, Ra
        INCR   Ri
        IF <   Ri, RN, LOOP
        STORE  RA, 0, Ra
```

where RA, RB, Ri and RN are initialized at compile-tıme.

3.3.3 Procedures

Procedures are used for code structuring, or even to save memory space (this is not the main utilization of procedures), and for code modularity or portability. For example, some procedures could be available in the form of *libraries*, which are pieces of code that have been already compiled and that can be inserted in any program by respecting the interface conventions. Notable examples are process run-time support libraries, as well as mathematical or graphical or financial functions, and so on.

In D-RISC the input parameter passing is done explicitly before the procedure call and the output parameter passing at the return point in the main.

In general, parameter passing can be done according to the following modes:

1. ***by_value***
   - *1.1. via registers*
   - *1.2. via memory*
2. ***by_reference***
   - *2.1. via registers*
   - *2.2. via memory*

Usually, the by_value mode is used for elementary types, while data structures are passed by_reference, i.e. by passing the base address. When possible or convenient, the value or the address is passed through a general register. However, there are some cases in which we are forced to pass the parameters via memory: for example, when a procedure library is used and its input-output data are variables in memory.

Example

Consider the following generalization of program 2 (previous Section):

```
int A[N], B[N];
        for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                        A[i] = A[i] + B[j]
```

We wish to exploit program 2 as a procedure:

```
int x; int Y[N];
for (k = 0; k < N; k++)
        x = x + Y[i];
```

with input parameters *x* and Y[N], and output parameter *x*. It is reasonable to pass *x* by_value and *Y* by_reference. Let us assume that the procedure definition uses parameter passing *via registers* Rx and RY (in this example, N is constant):

```
PROC:     CLEAR   Rk
LOOP_P:   LOAD    RY, Rk, Ry
          ADD     Rx, Ry, Rx
          INCR    Rk
          IF <    Rk, RN, LOOP_P
          GOTO    Rret
```

The main is:

```
          MOV     RB, RY
LOOP_M:   LOAD    RA, Ri, Ra
          MOV     Ra, Rx
          CALL    Rproc, Rret
          MOV     Rx, Ra
          STORE   RA, Ri, Ra
          INCR    Ri
          IF <    Ri, RN, LOOP_M
```

where the parameter passing interface is represented by are the MOV instructions in bold (MOV Rx, Ra can be avoided, using Rx directly; it has been used for modularity reasons).

Let us now assume that the procedure is an already compiled library, and that its input-output data are variables in memory: x = VM[x_pointer], base_Y = VM[Y_pointer]:

```
PROC:     CLEAR   Rk
          LOAD    Rx_pointer, 0, Rx
          LOAD    RY_pointer, 0, RY
LOOP_P:   LOAD    RY, Rk, Ry
          ADD     Rx, Ry, Rx
          INCR    Rk
          IF <    Rk, RN, LOOP_P
          STORE   Rx_pointer, 0, Rx
          GOTO    Rret
```

The main is the following:

```
          STORE   RY_pointer, 0, RB
LOOP_M:   LOAD    RA, Ri, Ra
          STORE   Rx_pointer, 0, Ra
          CALL    Rproc, Rret
LOAD   Rx_pointer, 0, Ra
          STORE   RA, Ri, Ra
          INCR    Ri
          IF <    Ri, RN, LOOP_M
```

Now the "LOAD Rx_pointer, 0, Ra" is necessary.

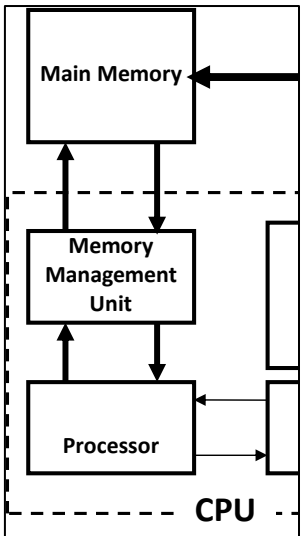
Some overhead is paid for parameter passing in the second version, as a counterpart for modularity and portability to exploit an available library.

Of course, other combinations for parameter passing are possible according to the procedure definition, e.g. x by_value via register and Y by_reference via memory.

3.4 An elementary processor for D-RISC

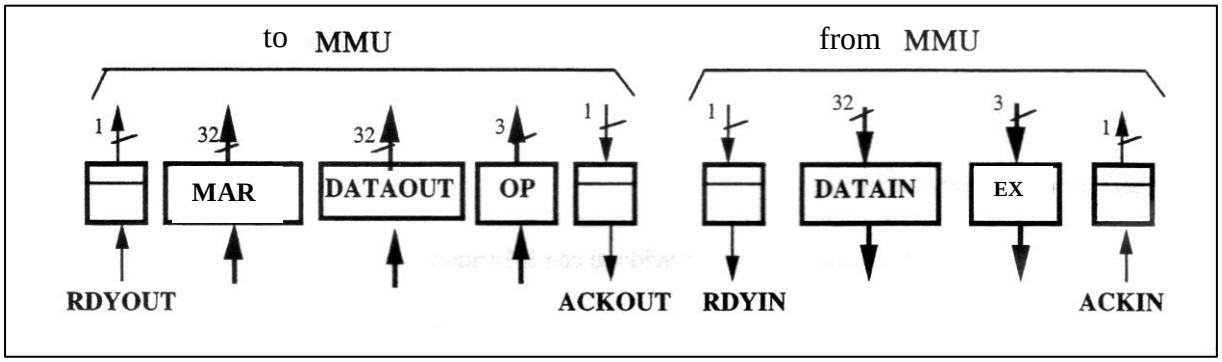
In this Section we describe the firmware interpreter of D_RISC for the processor of a very simplified CPU (no caching, no instruction level parallelism), just in order to show the basic concepts and to have an idea of the performance metrics.

The whole computer architecture has already been discussed as far as virtual memory *vs* main memory and MMU are concerned:



Processor and MMU are realized on the same chip.

The processor interface is the following:



where the interface registers are: MAR the Memory Address Register, DATAOUT and DATAIN the register containing values exchanged with the external word, OP the requested operation encoding (read /write /…). Memory interaction is by request-reply communications (Section 2.4.3), thus indicators ACKOUT and ACKIN will not be used for the moment being (they will be necessary for other asynchronous interactions).

The interface register EX contains a code of the operation outcome: if = 0 the request has been served successfully, otherwise it is the code of an *exception* (undefined address relocation function, protection violation, faults/errors of memory or of links, etc, as we'll seen in Section 4.3).

We assume that an *interrupt signaling* from I/O is detected by an interface indicator INT. The I/O and interrupt handling issues will be studied in Sections 3.6, 3.7. For the moment being, it is sufficient to provide some microprogram labels (*micro_int*, *micro_exc*) corresponding to the microcode segments firing the interrupt handling and the exception handing actions.

As introduced in Section 1.4.2, the general scheme of instruction interpretation is:

while (true) do

{ read the instruction, whose address is the current content of the program counter, from the memory;

decode the instruction according to the operation code;

if needed, read the operand values from the memory hierarchy or from registers;

execute the instruction;

if needed, store the results into the memory hierarchy or into registers;

modify the program counter with the address of the next instruction to be executed;

if an exception occurred or interrupt events are signaled, then call the proper handler procedure

}

The effective microprogram is listed in the following. Symbolic labels are used for the microinstructions, e.g. instruction fetch starts at microinstruction *ch0*. For each D-RISC class, the microprograms of only few representative instructions are shown. IR is the Instruction Register, i.e. a temporary register in which the current instruction word is stored. Other temporary registers (A, B; C, …, M, N, Q …) are used when needed. By definition, temporary registers of the interpreter level (firmware) are not visible at the interpreted level (assembler). The record formalism is used for register fields, e.g. IR.COP is the 8-bit operation code field of IR:

{instruction fetch}

ch0. IC → MAR, 'read' → OP, set RDYOUT, ch1

ch1. (RDYIN, or(EX) = 0-) nop, ch1;

(= 11) reset RDYIN, micro_exc;

(= 10) reset RDYIN, DATAIN → IR, ch2;

{instruction decoding and first microinstruction of every instruction execution phase }

ch2.

{ADD} (IR.COP, INT = 00…0 0) RG[IR.Roperand1] + RG[IR.Roperand2] → RG [IR.Result], IC + 1 → IC, ch0 ;

(IR.COP, INT = 00…0 1) RG[IR.Roperand1] + RG[IR.Roperand2] → RG [IR.Rresult], IC + 1 → IC, micro_int ;

{SUB} (IR.COP, INT = 00…1 0) RG[IR.Roperand1] – RG[IR.Roperand2] → RG [IR. Rresult , IC + 1 → IC, ch0 ;

(IR.COP, INT = 00…1 1) RG[IR.Roperand1] – RG[IR.Roperand2] → RG [IR. Rresult], IC + 1 → IC, micro_int ;

{MUL} (IR.COP, INT = 00…10 -) RG [IR.Roperand1] → M, RG [IR.Roperand2] → Q, 0 → N, 31 → C, IC + 1 → IC, mul0 ;

…

{LOAD} (IR.COP, INT = 001…00 -) RG [IR.Rbase] + RG[IR.Rindex] → MAR, 'read' → OP, set RDYOUT, ld0 ;

{STORE} (IR.COP, INT = 001…01 -) RG [IR.Rbase] + RG[IR.Rindex] → MAR, RG[IR.Rsource] → DATAOUT, 'write' → OP, set RDYOUT, st0 ;

…

{IF=} (IR.COP, INT = 01 …00 -) zero (RG [IR.Roperand1] – RG [IR.Roperand2]) → Z, if0 ;

…

{GOTO}(IR.COP, INT = 01 …01 0) IC + IR.offset → IC, ch0 ;

(IR.COP, INT = 01 …01 1) IC + IR.offset → IC, micro_int ;

…

{CALL} (IR.COP, INT = 11…0 0) RG[IR.Rproc] → IC, IC + 1 → RG[IR.Rret], ch0 ;

(IR.COP, INT = 11…0 1) RG[IR. Rproc] → IC, IC + 1 → RG[IR. Rret], micro_int

…

{continuation and termination of MUL execution phase}

mul0. …

…

mul5. …

…

…

…

{continuation and termination of LOAD execution phase}

ld0. (RDYIN, or(EX), INT = 0- -) nop, ld0;

(= 11 -) reset RDYIN, micro_exc ;

(= 10 0) reset RDYIN, DATAIN → RG[IR.Rdestination], IC + 1 → IC, ch0;

(= 10 1) reset RDYIN, DATAIN → RG[IR.Rdestination], IC + 1 → IC, micro_int

…

{continuation and termination of STORE execution phase}

st0. (RDYIN, or(EX), INT = 0- -) nop, st0;

(= 11 -) reset RDYIN, micro_exc ;

(= 10 0) reset RDYIN, IC + 1 → IC, ch0;

(= 10 1) reset RDYIN, IC + 1 → IC, micro_int

…

{continuation and termination of IF= execution phase}

if0. (Z, INT = 0 0) IC + 1 → IC, ch0 ;

(Z, INT = 0 1) IC + 1 → IC, micro_int ;

(Z, INT = 1 0) IC + IR.offset → IC, ch0 ;

(Z, INT = 1 1) IC + IR.offset → IC, micro_int

…

3.5 Performance evaluation

The processor Operation Part and Control Part are realized according to the general methodology of Section 2, along with the determination of the processor clock cycle τ, which is the clock cycle of all the CPU processing units, including MMU.

The main memory access time as seen by the processor is evaluated as in Section 2.4.3:

$$t_a = \tau_M + 2\,(\tau + T_{tr})$$

taking into account that 2 additional clock cycles are spent in MMU.

According to this knowledge and to the microprograms, we can evaluate the ***processing latency*** *for each class of D-RISC instructions*:

instruction fetch and decoding latency : $T_{ch} = 2\,\tau + t_a$

class 0.	ADD, SUB :	$T_0 = T_{ADD} = T_{ch} + T_{ex\_ADD} = T_{ch} + \tau$
class 1.	MUL :	$T_1 = T_{MUL} = T_{ch} + T_{ex\_MUL} \sim T_{ch} + 50\tau$
class 2.	LOAD, STORE :	$T_2 = T_{LD} = T_{ch} + T_{ex\_LD} = T_{ch} + 2\,\tau + t_a$
class 3.	IF= :	$T_3 = T_{IF} = T_{ch} + T_{ex\_IF} = T_{ch} + 2\,\tau$
class 4.	GOTO, CALL :	$T_4 = T_{GOTO} = T_{ch} + T_{ex\_GOTO} = T_{ch} + \tau$

where we distinguish between ‘short’ and ‘long’ arithmetic operations (respectively represented by ADD and by MUL), because of their very different latency of the execution phase. The latency of a 32-bit multiplication can be verified by microprogramming a typical add-&-shift algorithm.

This set of latency values is a good information to have an idea of the system performance. In particular, it is sufficient to evaluate the program ***completion time***. For example, the completion time of program 1 in Section 3.3.2 is given by:

$$T_c = N\,(3\,T_{LD} + 2\,T_{ADD} + T_{IF}) = N\,(6T_{ch} + 3\,T_{ex\text{-}LD} + 2\,T_{ex\text{-}ADD} + T_{ex\text{-}IF}) = N\,(22\,\tau + 9\,t_a)$$

Because $t_a >> \tau$, the processing latency per instruction and the completion time are decidedly dominated by the memory access time. In our example :

$$T_c \sim 9\,N\,t_a$$

This example confirms the discrepancy between processor and memory performances (*the Von Neumann bottleneck*). Memory hierarchies, istruction level parallelism and multiprocessing will try to solve or to alleviate this problems.

Completion time is used to compare different systems. For this purpose, standard ***benchmarks*** exist for various classes of application and algorithms, which evaluate a parameter which is a function the completion time. Example of benchmarks are the SPEC families. It is important to understand that benchmarks allow to evaluate *a system as a whole*, i.e. at all levels, and its programming tools, notably optimizing compilers.

Instruction service time and instruction bandwidth

The elementary processor, considered as an instruction execution, does not work on streams, since the processor itself is responsible to request instructions to the memory one after the other. However, virtually we can model the CPU as a stream-based computation, imagining that it is supplied with a stream of executable instructions: this model will be effective when the CPU will have a pipeline architecture (Part 1). With this abstraction in mind, we can speak about *instruction service time*, whose value coincides with the instruction latency for the elementary processor.

For a given machine with a certain assembler level, its firmware architecture is characterized by the *set of instruction service times*

$$\forall i = 1,\ldots,r: \quad T_i = k_i \cdot \tau$$

one for each instruction class (let *r* be the number of classes). We can characterize the applications through the *set of instruction probabilities* (also called *Instruction MIX*):

$$\forall i = 1,\ldots,r: \quad p_i \quad | \quad \sum_{i=1}^{r} p_i = 1$$

where p_i is the occurrence probability of instruction class *i-th*. Therefore, the *mean service time per instruction* is given by:

$$T = \sum_{i=1}^{r} p_i \; T_i$$

For example, if a hypothetical application class is characterized by program 1, Section 3.3.2, we have:

- class 0: short arithmetic: $p_0 = 2/6$, $T_0 = T_{ch} + \tau$
- class 1: long arithmetic: $p_1 = 0$
- class 2: Load, Store: $p_2 = 3/6$, $T_2 = T_{ch} + 2\tau + t_a$
- class 3: conditional branches: $p_3 = 1/6$, $T_3 = T_{ch} + 2\tau$
- class 4: unconditional branches: $p_4 = 0$

the service time per instruction is given by:

$$T = T_{ch} + \frac{2}{6}\,\tau + \frac{3}{6}\,(2\tau + t_a) + \frac{1}{6}\,2\tau = \frac{22\,\tau + t_a}{6}$$

By definition, the *instruction bandwidth* is:

$$\wp = \frac{1}{T} \quad instructions/sec$$

This very popular parameter is measured in Millions of Instructions per Second (MIPS) or multiples of it (GIPS, TIPS, …). Often it is simply called the system *performance*, if this does not create confusion with the generic meaning that we assign to the word "performance".

For example, the reader can verify that, with the typical values of τ and t_a, the order of magnitude of T is some 10^1 nsec, thus $\wp$ is order of magnitude of some 10^1 MIPS. These values are quite unsatisfactory with the current technology, and they will be decidedly improved in the real architectures studied during the course. We need to lower the memory access time with a low marginal cost increment (*memory hierarchies*) and to introduce parallelism between processor and memory, as well as inside the processor and inside the memory (*instruction level parallelism*).

Apparently, instruction bandwidth (performance) is an alternative metric to compare different systems. However, it can be used only *to compare different firmware architectures for the same assembler machine*. In fact, it is sufficient to consider a RISC and a CISC machine (thus, two systems with different assembler level): it is easy to verify that the RISC instruction bandwidth is greater than the CISC one; however, this is not sufficient to affirm that the former is "better" than the latter. A serious comparison can be based only on the *completion time*, not on the service time alone (while the commercial literature often contains such quite wrong evaluations based on MIPS!)

We can prove easily that, also at the instruction level,

$$T_c = m\,T$$

where *m* is average number of executed instructions.

(verify this relation for the example of program 1, Section 3.3.2, where *m = 6N*). Though apparently very simple, this relation plays a central role in architectures and technology, and it is the key for understanding the true nature of the RISC-CISC debate: *what is better, to decrease T with relatively greater m, or to decrease m with relatively greater T?*

For example, compare the D-RISC program 1 with the following CISC-like version:

```
LOOP:    ADD_MEM   RA, RB, Ri, RA
         INCR   Ri
         IF <   Ri, RN, LOOP
```

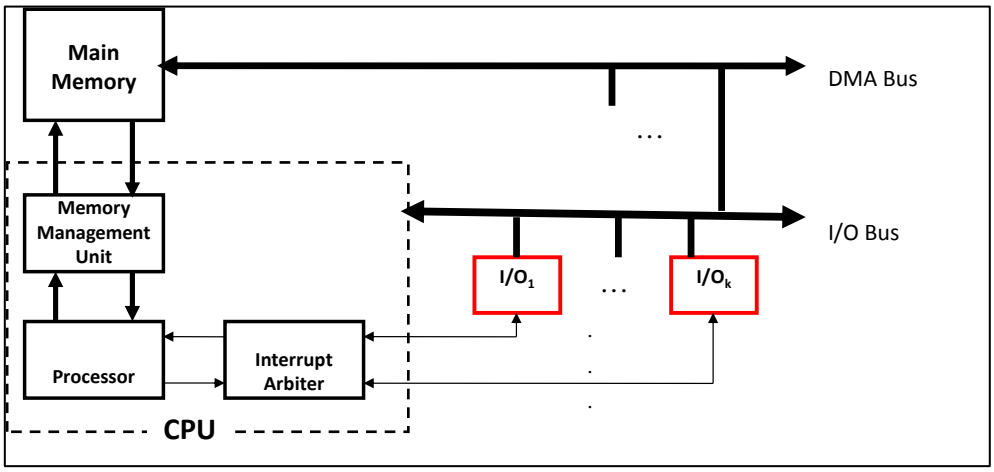
where ADD_MEM means:

$$VM[RG[RA] + RG[Ri]] + VM[RG[RB] + RG[Ri]] \rightarrow VM[RG[RA] + RG[Ri]]$$

The reader is invited to evaluate the service and the completion times of the RISC (with *m* = 6N) and CISC (with *m = 3N*) versions. In fact, the CISC completion time is lower, and, with good approximation, the main reason is that the number of memory accesses for instruction fetches is one half. However, in real CPUs with instruction level parallelism, we'll see that instruction fetch has a negligible effect on the completion time since it is substantially masked by other interpreter phases.

3.6 Input/output

The goal of the I/O subsystem is to allow CPU processes to exchange data with the external word constituted by various kinds of peripheral devices, including secondary memories, network cards, special engines, and other computers. Peripheral devices are interfaced by *I/O processing units* I/O_1, …, I/O_k:



In the simplest case, I/O unit realizes the physical interfaces according to the specific nature of the devices, as well as proper communication and synchronization strategies in order to optimize the device utilization, notably block transfers or low-level protocols for network. In the most advanced cases, I/O units provide more "intelligent" computational services in the form of data pre-elaboration and/or post-elaboration, e.g. editing interpreter for a laser printer, or higher-level protocols for networks.

Traditionally, I/O drivers (processes or procedures) are associated to I/O units in order to provide programmable interface services or, anyway, the required services when the I/O unit is not able to

do them. The border between simple units with driver and "intelligent" autonomous units is vanishing, depending on the technology adopted and on performance/flexibility trade offs.

I/O units can also be *powerful parallel machines* as in the case of vectorized co-processors and SIMD co-processors, of which GPU is a notable example. Provided that the data transfer between CPU-memory and I/O is not a bottleneck, such technologies make it possible to move heavy computation tasks from CPU to I/O co-processors.

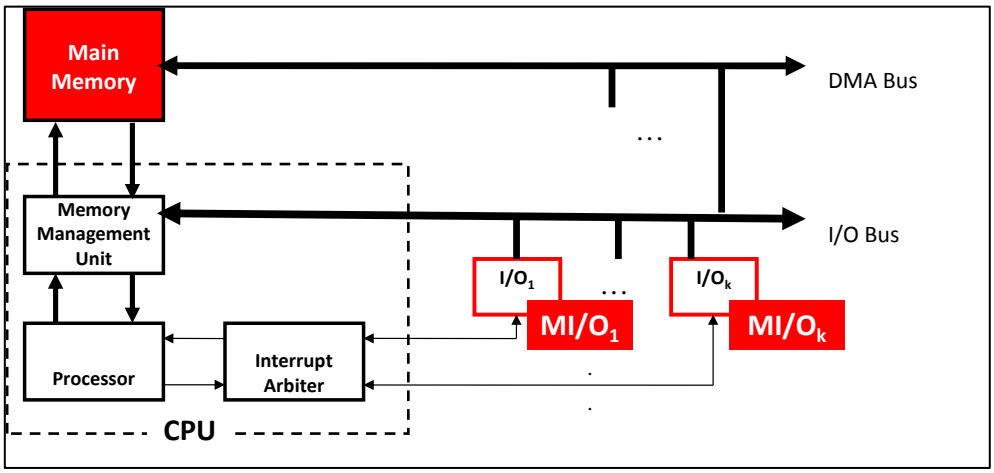
I/O units can be true firmware units, or they can be realized by CPU technology, e.g. an I/O "unit" can be a CPU-memory subsystem *per se*. In the latter case, the I/O unit functionalities are pre-compiled processes very similar to the CPU processes.

Conceptually, any I/O unit (firmware-specialized or programmable) can be thought of as an *"external" process*, cooperating with the internal (CPU) ones by means of the same mechanisms. Provided that the process run-time support is extended to cover both internal and external processes, a quite uniform management of I/O data transfer is implementable. This is an additional reasons for eliminating the distinction between I/O units and drivers.

3.6.1 Data transfers

For any I/O technology, supporting the CPU-I/O cooperation requires proper facilities for data transfers.

On the CPU side, the assembler machine could provide instructions dedicated to input and output transfers. This solution is rare: more often the so-called ***Memory Mapped I/O*** (MMI/O) method is adopted. In this method I/O is seen by the CPU as *memory*, so the normal LOAD / STORE instructions are used for I/O transfers too:



In fact, any I/O unit has its own *local memory*, either in the form of memory components in the Operation Part of a firmware-specialized unit, or in the form of an external memory unit (in the same way that CPU is connected to the main memory). Often, the real memory associated to an I/O unit is of relatively large capacity. However, independently of the local memory capacity, a maximum capacity per I/O unit is established.

Let M be the main memory, and let M/IO_1, …, MI/O_k the local memories associated to the I/O units. All these memory supports can be viewed by the processor as a unique modular physical memory. The result of the logical address translation, if successful, is a physical address belonging to M or to M/IO_1, …, or to MI/O_k. This is fully invisible to the running process and to the processor. As shown in the figure, it is natural to delegate MMU the task of routing a memory request to M or to M/IO_1, …, or to MI/O_k, according the memory module to which it belongs. For example, the most significant bits of the physical address identify the memory module.

Processor and I/O work *in parallel*, thus it is possible that the local I/O memories have to be arbitrated. Usually, the arbitration functionality is associated to the I/O unit.

On the I/O side, another possibility exist to transfer data: ***Direct Memory Access*** (DMA), i.e. the possibility to read/write directly from/into the main memory in the same way that the processor does. The direct I/O accesses to M occur in parallel with the CPU behavior. For this purpose, M is equipped with an arbitration functionality.

In general, both methods, MMI/O and DMA, co-exist with notable advantages of performance and flexibility.

For example, consider the following internal process (see program 1 referred to in the previous sections):

```
LOOP:   LOAD   RA, Ri, Ra
        LOAD   RB, Ri, Rb
        ADD    Ra, Rb, Ra
        STORE  RA, Ri, Ra
        INCR   Ri
        IF <   Ri, RN, LOOP
```

Arrays A, B can be physically allocated in the main memory or in the local memories of I/O units, and this is quite invisible to the process that generates logical addresses. Even if the physical allocation will be modified, statically or dynamically (e.g. A is initially allocated in M and later on it is moved, at least in part, into MI/O_3), the internal process remains unchanged (it is not re-compiled). More: assume that the internal process is started as soon as arrays A and B have been acquired from external data sources (disks, networks, etc). It is possible that, according to the external source and/or according to memory allocation strategies, A and B have been loaded by the I/O unit into M or into its local memory. Again, this is invisible to the internal process. And, equally important, *the internal and the external process behave in parallel* and *cooperate via shared memory*, where the physical shared memory space is the union of M and MI/O.

3.6.2 Synchronization and events: interrupts

During the cooperation between internal and external processes, synchronizations have to be implemented, as they belong to the process run-time support, for example synchronization of data exchange via shared memory.

From the internal process (CPU) side, synchronization is implemented via MMI/O, notably by writing proper information into MI/O. This is easy because the I/O unit can execute the external process only, thus it can test the occurrence of such events in a *busy wait* ("active waiting") state.

From the external process (I/O) side, events (e.g. the presence of new data) must be communicated explicitly. In fact, the internal process, that must be informed of the transfer occurrences, in general has been de-scheduled ("passive waiting") while waiting the event itself. In other words, such events are *asynchronous* wrt the CPU behavior.

Thus, a mechanism must be provided in order to detect and to handle asynchronous events from I/O units. This is the *interrupt* mechanism: *the I/O unit signals the event through an explicit communication to CPU at the firmware level*. Such communications are called interrupts. It is convenient to distinguish between the event occurrence (*interrupt signal*) and the *event content*: the latter is a message (*interrupt message*) consisting of the event identifier and some parameters used

in the event handing procedure. Usually, the interrupt message consists of very few words: in D-RISC it is a 2-word message sent from the I/O unit through the I/O Bus.

More than one unit could send an interrupt to the processor simultaneously. For this reason, an *Interrupt Arbiter* unit is provided, which selects one interrupt signal at the time.

Since interrupts are asynchronous, the processor must test the presence of an interrupt periodically. Usually, the interrupt test occurs *at the end of the execution phase of the firmware interpreter*, as it is shown in the microprograms of Section 3.4.

A 1-bit interrupt signal (INT) arrives to the processor. It is the logical OR of all the interrupt signals currently received by the Interrupt Arbiter. From the processor side, this is sufficient in order to know that at least one I/O unit is "requiring attention". If INT = 1, a positive reply is sent to the Arbiter, which forwards such acknowledgement to the selected I/O unit. Now, such unit sends the interrupt message. The processor enters an interpreter phase (starting at *micro_int* in Section 3.4) during which it receives the interrupt message, which will drive an interrupt handling procedure.

Interrupt and exception handling will be described in the next section.

Interrupt management instructions and annotations

A process must be able to ignore interrupts during some critical processing functionalities, notably for performance reasons (eliminating interrupt handing latencies during strict real-time processing), or for correctness reasons (some interrupt handlers could render inconsistent the state of one or more internal processes).

For this purpose, special instructions are provided at the assemble level. In D-RISC they are (Section 3.3.1):

- *DI*: all interrupts are disabled,
- *EI*: all interrupts are re-enabled,
- *MASKINT*: only the interrupts identified by a bit-mask are disabled.

In general, the *process run-time libraries are executed with disabled interrupts*: in a uniprocessor architecture this is a sufficient condition to achieve *indivisibility*, or *atomicity*, of the run-time support functionalities.

D-RISC provides also the possibility to execute DI or EI actions during any other instruction, through proper *annotations*. This is useful for performance reasons (saving instructions), and for correctness reasons, i.e. in some cases interrupt disabling/enabling must be an atomic event wrt some instructions. Examples will be seen in successive sections.

Finally, D-RISC provides the *WAITINT* instruction, causing the *busy waiting* a specific interrupt. This is useful in some dedicated/real-time applications, and/or in multiprocessor run-time support (Part 2).

3.7 Interrupt and exception handling

All the interrupts are (asynchronous) events generated at the firmware level. As seen, many interesting exceptions, notably those signaled by MMU to the processor, are (synchronous) events generated at the firmware level.

In both cases, we wish a mechanism according to which each event can fire an *event handler*, that is an assembler level procedure that provides all the actions caused by the event. For example:

- some interrupts signals the processor the event “process wake-up”, that is a certain process, which was in the Waiting state, must be waked up because the waiting condition is no more true. Typically, this event is signaled by an external process to a partner internal process. The interrupt message contains the following information: (event identifier, process name). The first parameter identifies a specific *interrupt handler*, which is the low-level scheduling library for process wake-up; the second parameter is the necessary input data of such library;
- the page fault exception signals the processor (thus, the currently running process) that the information required to the memory is not currently allocated. The event can be represented by the pair (event identifier, logical address). The first parameter identifies a specific *exception handler*, which provides to start the needed information transfer from secondary memory into main memory (of course, during the handler a context-switch will occur); the second parameter uniquely identifies the needed information that must be allocated into main memory.

In general, the mechanism we have looking for must have the following characteristic:

- the interpreter of any instruction, affected by an interrupt or an exception, calls the procedure corresponding to the event handler, with proper return address.

The procedure return is at the next instruction for interrupt handling, while different cases are specific of exceptions. For example, for page fault exception the procedure return is at the same instruction generating the exception, because the instruction has to be re-executed when the needed information will be allocated in main memory.

This motivates the general scheme of instruction semantics, introduced in Section 3.2.1:

{ execute the actions corresponding to operation code and other fields;

update IC to the logical address of the next instruction;

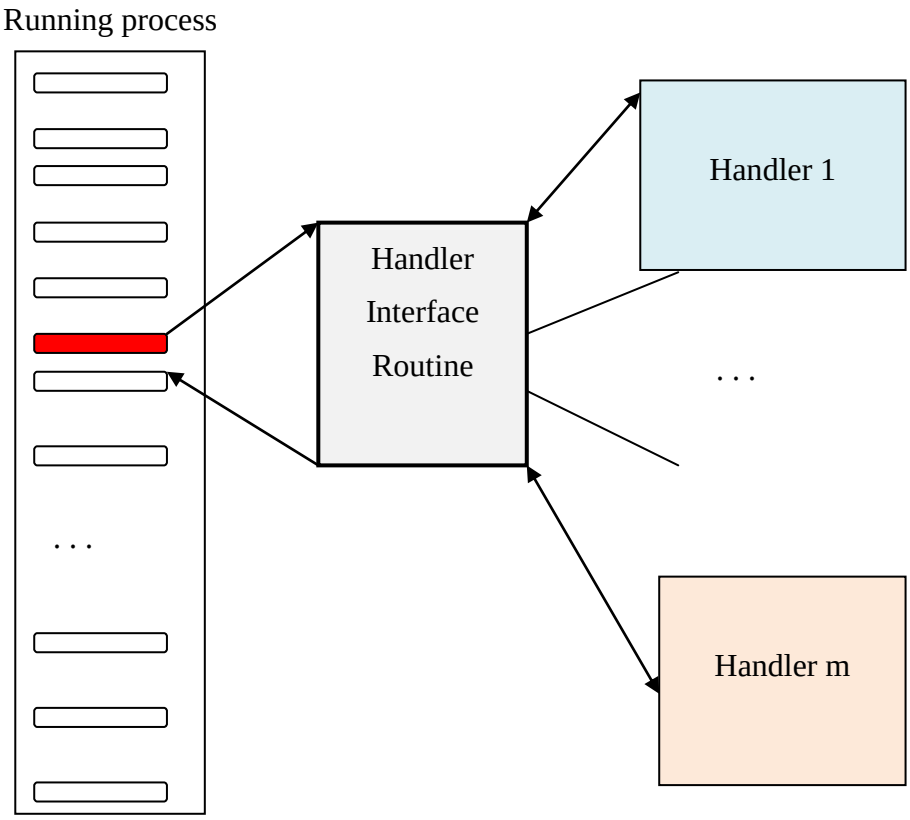
activate other actions (procedures) for event handling, both synchronous events (*exceptions*) and asynchronous events (*interrupts*) }.

By definition, this semantics is reflected in a proper structure of the firmware interpreter.

Let us describe in detail the ***interrupt handling for D-RISC machines***.

The firmware phase contains the following actions:

1. acknowledge the interrupt signal,
2. receive the interrupt message (event identifier, data) from the I/O Bus, thus from MMU acting as I/O Bus interface,
3. call the procedure corresponding to the identified event. For flexibility reasons, it is preferable to work in the following way: call a generic *Handler Interface Routine*, which will provide to call the proper handler, as shown on the following figure:



The firmware phase is implemented by the following microprogram:

micro_int. reset INT, set ACKINT, goto int0

int0. **if** (RDYIN = 0) **then** {nop, **goto** int0}

else {reset RDYIN, set ACKIN, DATAIN → RG[event], **goto** int1}

int1. **if** (RDYIN = 0) **then** {nop, **goto** int1}

else {reset RDYIN, set ACKIN, DATAIN → RG[data], IC → RG[ret_from_interrupt], RG[handler_interface] → IC, **goto** ch0}

The first microinstruction is the continuation of all microinstructions detecting INT = 1. Notice that, at this point, the execution phase has been concluded and IC has been correctly updated. The second and third microinstructions receives the two words of the interrupt message. Notice that the indicator ACKIN has been used since, in this phase, the cooperation with MMU is asynchronous (not request-reply). The received words are written into two predefined general registers, Revent and Rdata, to be used at the assembler level. The third microinstruction contains the operations for procedure call and for return address saving, using other two predefined general registers, Rhandler_interface and Rret_from_interrupt. Notice that the next microinstruction is ch0 (instruction fetch).

Here is the simple and efficient structure of the Handler Interface Routine, which uses a predefined table (Rhandler_table) containing the correspondence between event identifiers and base addresses of Handlers:

LOAD Rhandler_table, Revent, Rhandler_address

CALL Rhandler_address, Rret_from_handler

GOTO Rret_from_interrupt

The reader is invited to think about some specific interrupt and exception handler procedures.

3.8 Exercises

1. A given functionality F is implemented at the assembler level in computer C1, while it is fully implemented at the firmware level in computer C2.

   In both cases parallelism is not exploited at the architecture level, i.e. no instruction-level parallelism, the processor-memory architecture is sequential.

   Explain why the C2 implementation of F has latency less than, or equal to, the C1 implementation. Give an order of magnitude of their difference.

2. Study the compiling rules for D-RISC, and explain the compilation example of array addition.

3. A certain computer C1 is D-RISC. The assembler machine of computer C2 is D-RISC plus an instruction computing the square root of reals. Verify the following assertion: for the same application program using the square root function, C1 has a performance P greater than C2. Is it true, or false? Explain the answer.

4. Explain where and how in a D-RISC machine the decision is taken about the destination of a memory request (Main Memory or an I/O unit).

4. Processes and virtual memory

In this Section we study several interrelated aspects of general-purpose computer architectures at the various levels, by a thorough overview of the concepts and techniques introduced in Section 1. The goal is to acquire a deep and structured knowledge of the characteristics of a computing system as a whole. Apart to be important per se, this knowledge constitutes a strong background for studying advanced issues and trends in modern computer architecture, high-performance computing and complex computing platforms.

The interrelated aspects to be overviewed are:

- process representation, run-time support, and life cycle,
- virtual memory implementation,
- shared objects and shared pointers.

4.1 Data structures for process run-time support

The run-time support of a process P contains some important data structures, *partially initialized at compile time* and *belonging to* VM_P. Among them:

- the ***process descriptor***, or Process Control Block(PCB_P): the main information in PCB_P is the *image* of the initial value of the program counter and of the processor general registers. In the example of Section 3.1:

```
int A[N], B[N]; int x = 0;
        for (i = 0; i < N; i++)
                x = A[i]*B[i] + x
```

  - the image of IC in PCB_P is initialized at zero;
  - the images of general registers RA, RB, RN, Ri, Rx are initialized, respectively, to 16K (logical base address of array A), 1M+16K (logical base address of array B), 1M (array size *N*), 0 (initial value of index *i*), 0 (initial value of *x*). That is, these are the values chosen by the compiler to allocate the process data objects in VM_P and in the general registers. Other registers, like Ra and Rb, are chosen by the compiler to be used as temporary variables, however their images are not initialized. Assuming that the assembler machine has 64 general registers, as in D-RISC, in this example 5 out of 64 images of general registers are initialized in PCB_P, the remaining 59 images are left unspecified.

PCB_P contains *space for any other utility information concerning the run-time support of P*, notably *pointers* to the various run-time support data structures and to run-time support libraries (primitives procedures, low-level scheduling procedure, interrupt and exception handlers). Moreover, it contains the reference to the procedure stack, if provision for it is made. It is useful also that PCB_P contains additional space for *register saving* according to the strategy adopted by the compiler when the register array size is currently insufficient;

- the ***representation of the address translation function***, usually in the form of a table (*Relocation Table*, TABRIL). This function is only partially initialized at compile time. The compiler knows the domain of the function and its cardinality, however the most meaningful information is not yet known: the co-domain values, which are left unspecified. It is useful to store the logical base address of $TABRIL_P$ in a field of PCB_P;

- ***support data for the implementation of lists of processes***, notably the *Ready List* (see Section 1.2.1), which contains the PCBs of all the ready processes. Using linked lists, it is convenient that other fields of the PCB contain the reference to the head/tail list pointers, and the forward/backward pointers in the list;
- ***descriptors of data structures for the process cooperation***, notably all the communication channel descriptors (containing all the information needed in the implementation of send-receive primitives) in a message-passing framework, or semaphore/condition variable descriptors in a shared variable framework. VM_P contains all the descriptors of the cooperation objects used by P. The communication run-time support will be studied thoroughly in Session 6.

4.2 From process creation to process execution

Let us study the process life cycle after the compilation: from the creation to the effective execution by the processor.

Process creation

When the user decides that a compiled process P has to be executed, then the process creation is launched through a proper tool of the programming framework. P will be properly manipulated by a special system process, the *Process Manager*, who drives all the steps needed to render to process executable. These steps consist in

- main memory allocation,
- process *loading* into main memory,
- process *enabling*.

Created process will also be called *active* processes, i.e. processes that can be executed.

Main memory allocation and process loading

The *Main Memory Manager* service is delegated to allocate main memory space to process P. According to specific strategies, the Memory Manager choses the areas of M into which VM_P, or usually a part of it, has to be initially loaded. *The Relocation Table of the created process is updated accordingly* by the Memory Manager.

If a partial initial allocation is done, a *dynamic allocation of main memory* is adopted. This means that further objects will be loaded/unloaded into/from M at run-time when needed, again through the intervention of the Main Memory Manager. Dynamic memory allocation will be studied thoroughly in Section 4.3. In case of dynamic allocation, some specific parts of VM_P are necessarily loaded into M during this phase: at least PCB_P and $TABRIL_P$ are initially needed.

The memory allocation functionality requires further data structures (meta-data) in the form of a *configuration file* associated to the VM_P file. The configuration file contains information

- about the allocation of run-time support objects in VM_P, notably the position of PCB_P and $TABRIL_P$,
- about the position of the instruction section and about the position of the data that will be referred first and/or more frequently (this strategy is related to the virtual memory hierarchy exploitation, as studied in Section 4.3),
- about the shared objects that are referred by the process (see Section 4.4).

When the Memory Manager has completed the allocation of P, the Process Manager asks other system services to perform the VM_P file loading from the permanent store into M, by passing the proper information to them. Such services include

- the File Manager,
- the Secondary Memory Manager,
- the I/O Driver.

Process enabling

When VM_P, or the chosen part of it, has been loaded into M, a final service is invoked by the Process Manager: process *enabling*. That is, P must pass into the *Ready* state: thus, the low-level scheduling functionality of process run-time support (Section 1.2.1) is invoked in other to link PCB_P into the Ready List (remember that the PCB_P has certainly been loaded into M).

From now on the process P is *active* until it terminates.

Context-switch

P will effectively pass into the *Running* state when a *context-switch event* of the low-level scheduling occurs, such that the currently running process (say P1) exits the Running state (for example, P1 passes into the Waiting state) and P is in the highest priority position of the Ready List.

Before going on, an important issue must be understood: *all the process run-time support actions are done by the currently running process,* unless they are delegated to other processes as in some previous cases (e.g. the Process Manager delegates some tasks to the Memory Manager, to the File System and so on). In particular, it is P1 that executes all the instructions of the context-switch procedure: the last instruction of P1 must be the last context-switch instruction, therefore the next instruction executed by the processor interpreter will be the first instruction of P.

The context-switch support provides:

a. to save IC and general register values from the processor into their images in PCB_{P1};

b. to unlink PCB_P from the Ready List and to load the IC and general register images from PCB_P into the processor. At this point it is clear what means that *some registers are initialized at compile time*: their initial value has been put into the PCB image in the VM file, so during the context-switch such values will be effectively copied into the physical registers of the processor;

c. to acquire the $TABRIL_P$ pointer (which is in PCB_P) and use it to switch from the address translation function of P1 to the address translation function of P.

These actions are done each time P will come back into the Running state after having alternated phases Waiting-Ready-Running repeatedly.

Saving and restoring the registers (in phases *a, b*) is not a problem: they are merely sequences of LOAD and STORE instructions of P1. Phase *c,* along with the consistent updating of IC register, is more critical.

Assume that MMU uses the base address of TABRIL to refer to proper blocks of the relocation table of the running process: conceptually, though not physically, we can think that MMU reads the relocation table into a local fast memory of its Operation Part (the detailed behavior and implementation of MMU will be studied in Section 4.3.3). Thus, in pass *c* the $TABRIL_P$ pointer has to be sent to MMU. This action is necessarily executed at the firmware level, thus it requires a *special instruction*, called START_PROCESS in D-RISC (Section 3.3.1), whose microprogram

sends the $TABRIL_P$ pointer (copied into a general register RTabril by a previous instruction of phase *c*) to MMU via the processor-MMU interface:

START_PROCESS Rtabril, …

Now, assume that the previous phase *b* has initialized IC with the image in PCB_P, and that the next instruction is the START_PROCESS. This behavior is not correct: the START_PROCESS instruction is not executed because IC is pointing to the first instruction of P. On the other hand, exchanging the two actions is not even correct, because, if the START_PROCESS is executed first, then the address of the instruction updating IC will be translated by $TABRIL_P$ (instead of $TABRIL_{P1}$).

In conclusion the correct solution is that the START_PROCESS performs *both* the IC updating action and the MMU updating action *atomically*:

START_PROCESS Rtabril, RIC

where RIC is the general register into which the IC image has been copied during phase *b*.

Finally, since the context-switch procedure is executed with *disabled interrupts* (see Section 3.6.2), the START_PROCESS instruction must re-enable interrupts atomically with the actions described. An EI annotation in the instruction itself (Section 3.3.1) is an elegant and efficient solution to this problem:

START_PROCESS Rtabril, RIC, **EI**

Apart to be useful per se, this discussion shows important concepts of structured computer architecture:

- some process life cycle functionalities do not belong to the run-time support of processes, notably the initial phase of main memory allocation and process loading, including the initial definition of the address translation function;
- some functionalities of the run-time support of processes are implemented at proper levels, notably the address translation belongs to such run-time support and it is implemented at the firmware level (MMU) for efficiency reasons;
- some special commands are necessary (or very convenient) at certain levels in order to be able to effectively implement the run-time support of higher levels. In particular, if we wish that some specific actions are done at the *firmware* level, then it is necessary that a proper instruction exist, whose microprogram executes exactly such actions: a notable example is the START_PROCESS instruction.

This concepts will be exploited several times during the course.

4.3 Memory hierarchies and virtual memory

In this Section we study the implementation of virtual memory and the dynamic allocation of main memory in detail. As seen, the virtual memory VM_P of a process P is an abstraction of the memory used by P, independently of the effective realization of the physical main memory M. In order to have as more active processes as possible, M is shared by all the active processes, thus M is dynamically allocated at least for two reasons:

- application processes are activated and terminate dynamically,
- in order to optimize the number of currently active processes, it could be necessary to allocate a *proper* fraction of VM of each process instead the whole VM.

The relationship between VM (better: the set of VMs of all the active processes) and M is a notable application of the *memory hierarchy* concept. In Section 5 memory hierarchies will be formalized, and applied to the important case of caching. In the current Section we describe the implementation of the VM-M hierarchy, by assuming some quantitative parameters which will be explained in Section 5.

4.3.1 Introduction to memory hierarchies

In a memory hierarchy M2-M1, the highest level M2 has lower cost per bit, greater capacity, and greater access time than the lowest one M1. In the VM-M case, VM is implemented on secondary memory.

The goal of the memory hierarchy implementation is to optimize the performance/cost ratio by achieving a good approximation of the following ideal situation: *a)* the access time of any information is equal to (slightly greater than) the access time of the lowest level (M, i.e. random access technology), and *b)* the cost of the whole system memory is equal to (slightly greater than) the cost of the highest level (MV, i.e. sequential access technology). For point *b)* the approximation is easily satisfied: the marginal cost by adding a RAM to a disk is relatively low and acceptable.

The main problem of memory hierarchies is point *a)*: if only a fraction of VM is dynamically allocated in M, than we have to maximize the *hit probability* that the information required by the processor (instructions and data) are currently present in M. With a certain probability, called *fault probability*, the information referred to by the processor is not present in M: this *fault* event must be managed by transferring the required information from the highest level (disk) into the lowest one (M), possibly by replacing some already present information if no free space is currently available.

Of course, this information transfer is a relatively slow action, too slow indeed: the order of magnitude of disk access time is 10^{-3} sec, while we know that the M access time is of the order of 10^{-6} – 10^{-9} sec. Intuitively, the fault probability must be not greater than 10^{-5} – 10^{-6} in order to roughly achieve a good approximation of our goal.

A central issue is: in case of fault, how many information are transferred from the highest level to the lower one? In the VM-M case, information must be transferred from/into disks, which are organized to transfer ***blocks***, or ***pages***, of consecutive information, typically of 1K – 4K size. Thus, for each fault on a certain information *x*, the entire block containing *x* is transferred.

This technological feature gets on well with logical properties that statistically are peculiar of program behavior, called ***locality*** (or spatial locality) and ***reuse*** (or temporal locality). By postponing a rigorous definition to Section 5, we can intuitively understand such properties:

- ***locality***: typically, programs are built in such a way that, if some information *(x, y, z, ...)* are needed during a certain phase of the computation, than it is very probable that the information referred in the next future will belong to the page containing *x*, and/or to the page containing *y*, and/or to the page containing *z*, …;
- ***reuse***: usually programs are based on iterative (or recursive) structures, according to which some information could exist that, once referred, will be referred several times in the future.

These properties are statistically verified on very large samples of programs. For this reason ***paging***, i.e. the strategy of using *pages* as the transfer unit, has been adopted as the basic technique for memory hierarchies.

The fault probability of a paged memory hierarchy is a function of several parameters, the most important of which are the capacity of the lower level (M capacity in our case) and the page size. For a given capacity, the fault probability has a minimum for a certain page size. On large samples

of benchmarks, it has been verified that, for typical M capacities, the *optimal page size* of the VM-M hierarchy is just in the range 1K – 4K.

Of course, every program is characterized by its own degrees of locality and reuse, i.e. by its own fault probability, that has a profound impact on the program performances. For programs with low locality or reuse, the transfer overhead could prevail the processor computation time, leading to unacceptably high service times and high completion times.

As we'll study in Section 5, an important class of program optimizations concerns *program structures able to exploit the locality and reuse properties at best*. Though many results and methodologies exist, this is, and it will be in the long term, an extremely important research issue.

4.3.2 Virtual memory: address translation

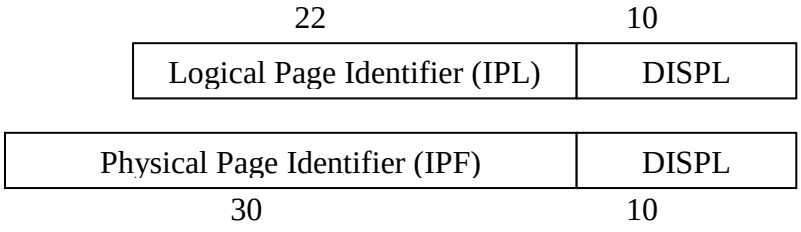
Formally, let

- $\lambda(X, P)$ denote the logical address of object X in the virtual memory VM_P of process P,
- $\pi(X)$ denote the physical address of object X in main memory M, provided that X is currently allocated in M,
- μ_P denote the address translation function of process P:

$$\mu_P\,(\lambda(X, P)) = \pi(X) \text{ or undefined}$$

if the function is undefined, then the reference to X generates a *fault* condition.

In a paged VM-M hierarchy, an address can be seen as composed by two fields: most significant bits represent the *page identifier*, and the least significant bits represent the address internal to the page, or *displacement*. For example, for 32-bit logical address (4G = maximum size of VM per process), 40-bit physical address (1T = maximum capacity of M), and page size of 1K, the logical and physical address configurations are respectively:



The address translation function μ is applied to *IPL* only. If the function is defined on such IPL, then the result of its applications is IPF:

$$\mu_P\,(IPL)) = IPF \text{ or undefined}$$

If an IPF is returned, the physical address is obtained by concatenating IPF with the DISPL field of the logical address.

Because M is allocated dynamically, and it is shared by more than one process, it is not feasible to define μ according to an analytical function. Thus, μ is defined by a table – the ***Relocation Table*** TABRIL – , which can be implemented as a linear array of *NP* entries, where *NP* is the number of logical pages of the process (in the example, $NP_{max} = 2^{22} = 4M$):

The table is indexed by IPL in order to read the corresponding IPF, or to detect the page fault condition; a *presence bit* (P) is sufficient for this:

P	IPF	MOD	REPL	PROT
1	30	1	…	…

Each entry of TABRIL contains other information, useful for the memory hierarchy management:

- MOD is the *modified-bit*, to indicate that at least one word of the page has been modified. This information is exploited when the page will be de-allocated, to decide if the physical page has to be re-written into the disk;
- REPL is a set of bits used to implement the *page replacement strategy*, i.e. to determine the physical page to be replaced in case of fault. Usually, according to the application of the locality and reuse properties, a good strategy consists in replacing the "Least Recently Used" (LRU) page. For this purpose, a counter is associated to each page: of course, the rigorous application of LRU would require an unlimited number of bits; however, few bits for REPL are used for a sufficiently approximated application;
- PROT is a field encoding the *protection rights* of the process on the object corresponding to this page. We associate to every object a set of permissible operations (the object *type*), a subset of which is allowed for each process having this page in its VM. During the address translation, a run-time comparison between the field content and the operation requested on that object detects possible *protection violations*. This is the third possible outcome of the address relocation function (in addition to IPF or undefined). In case of page fault or protection violation, an *exception* is generated, to which a proper handler corresponds (for page fault: the page transfer).

In the simplest case, the protection rights concern just variants of memory read/write operations. More sophisticated architectures provide much wider set of checkable operations (e.g. according to the language type of the object), thus much more powerful protection controls.

4.3.3 Memory Management Unit

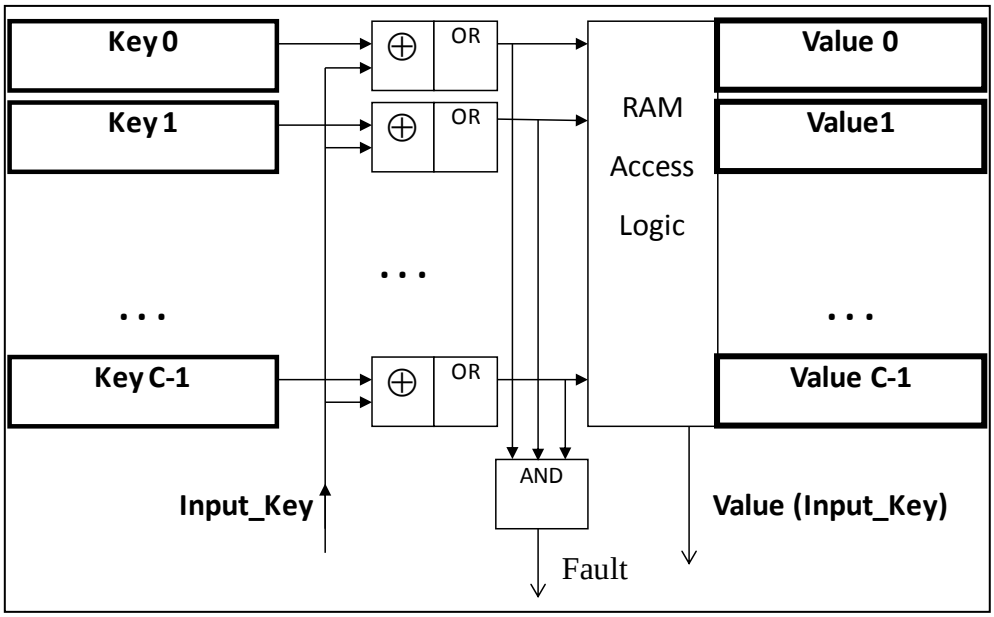
As introduced several times, the address translation task must be implemented at the firmware level with the lowest latency as possible. For modularity reasons, this task is delegated to a separate unit in the CPU, called Memory Management Unit (MMU). In the elementary architecture (Section 3.4) MMU and processor work serially, however in Instruction Level Parallelism CPUs MMU is one of the (several) units working in parallel: thus, both modularity and performance are optimized by an implementation of the address translation in a separate unit.

MMU uses and manipulates the Relocation Table (TABRIL). Of course, accessing the table in memory for any access has no sense. Instead, a form of memory hierarchy is applied to the Relocation Table itself: a small part of TABRIL is *dynamically* maintained inside MMU in order to exploit the reuse of its entries. Thus, MMU must contain a *content addressable table*, accessed through the logical page identifier (IPL). In addition, we require that such table is accessed in one clock cycle: this requirement cannot be satisfied using a RAM and some hashing technique (several clock cycles are needed just to compute the hash address), thus a specialized hardware component must be realized. This component is the so called ***Associative Memory*** (MA), which is the most direct implementation of a content addressable table.

In general, a content addressable table is composed of C pairs (Key, Value). For any Input_Key, the content addressable table returns the *Value(Input_Key*), i.e. the value associated to the Input_Key, if the Input_Key is present in (at least one location of) the table, otherwise a *Fault* condition is

returned. We are interested in content addressable memory in which *at most one* entry contains the Input_Key.

In the MA implementation, all the C Keys locations are compared in parallel with the Input_Key, obtaining C bits. If all such bits are equal to one, than the Fault bit is generated, otherwise their configuration uniquely identifies one of C Value locations:



This is a read-only version of MA. In general, the modifications are not implemented in an associative way, instead they are done by address, thus also the Key locations are accessible by address (not shown in the figure).

Usually, an LRU (Least Recently Used) strategy is applied to the TABRIL-MA hierarchy, i.e. the most recently used entries are dynamically maintained in MA. A Fault of MA is managed directly by MMU itself by accessing TABRIL in memory (the processor is not aware of this event). The TABRIL address has been communicated to MMU during the context-switch phase (START_PROCESS instruction, Section 4.2).

In general, the TABRIL-MA hierarchy is efficiently exploited with a low capacity MA, for example typically C is of the order of 10^1 with a MA-fault probability of 10^{-2}, which implies an acceptable overhead for the memory access latency.

In the case of MMU, the Values are TABRIL entries and the Keys are the corresponding IPLs. During the same clock cycle in which the processor request is received, the MA is accessed and, if no MA Fault is generated, the TABRIL fields are used to compose the physical address (Section 4.3.2) and the so-modified request is sent to the main memory. In case of MA Fault, once the requested TABRIL entry is inserted into MA, the presence bit of the entry (see Section 4.3.2) is tested. If the page is not present in main memory, a *page fault exception* is signaled to the processor, otherwise the address translation and the memory request are completed.

During the same clock cycle all the needed modification to MA fields are done.

During the same clock cycle, MMU performs protection controls (Section 4.3.2), and possibly generates a Protection Violation exception.

4.3.4 Page fault exception

In Section 3.7 we studied the implementation scheme for interrupt and exception handling. This scheme can be applied to the page fault exception:

- in the firmware phase the Exception Interface Routine is called with return address equal to the current IC and with parameters given by the exception code, the process identifier, IPL and memory operation;
- the Exception Interface Routine calls the Handler corresponding to the exception code;
- the Page Fault Handler interacts with the Main Memory Manager (see Section 4.2). If this is a process, a message is sent to it requesting the page allocation, then a receive primitive is executed in order to wait for the termination of the page allocation and loading in main memory, including the TABRIL updating, and the possible page re-writing if modified and meaningful. The receive execution implies that the running process passes into the waiting state, so the needed context switch occurs. The process is waked up when the Memory Manager sends the waited message. The continuation address is the saved IC.

The page replacement algorithm (e.g., LRU) can be executed in the Handler before the request to the Memory Manager if the algorithm is applied only to the pages of the running process itself. Otherwise, if the replacement strategy is applied to more than one process, than the algorithm is executed by the Memory Manager (remember that it has access to all the active processes TABRILs).

4.4 Virtual memory and shared objects

A process refers *private* objects (in the example of Section 3.1: instructions, arrays A and B) as well as objects that are *shared* with other processes, notably several data structures of the run-time support:

- communication channel descriptors are shared by all the processes using the to communicate;
- the Ready List head/tail pointers are shared by all the processes currently created (*active* processes). Therefore, the PCB of any active process is shared by all active processes. For example, during a process executing the context-switch procedure must refer the first ready PCB, which may be any PCB;
- the Relocation Table of any active process is shared by all the active processes (again, see the context-switch procedure).

Atomicity in shared objects manipulation is implemented by proper synchronization operations, that will be review in several parts of the course for any architecture.

In order to share an object, a process must to be able to address it, thus the process must possess that object in its own virtual memory. It is a task of the compiler *to allocate all the shared objects in all the virtual memories of processes sharing them*.

Notice that while the content of PCB_P in VM_P is meaningful and initialized, the content of any other PCB_Q in VM_P is unspecified. In fact, what we want is that P is able to address PCB_Q, without any need to update VM_P with the modifications on PCB_Q: such modifications are done in the unique physical copy of PCB_Q in main memory.

Formally, an object X is shared by process P and Q iff :

$$\mu_P(\lambda(X, P)) = \mu_Q(\lambda(X, Q)) = \pi(X)$$

The shared object X exists in VM_P and in VM_Q. In general $\lambda(X, P)) \neq \lambda(X, Q)$, since the compiler is free to allocate the virtual memory of a process independently of any other. However, because X is shared, it exist in single copy in main memory (if currently allocated in M).

As said in Section 4.2, the process *configuration file* contains the indication of the shared objects. When P is created, the Memory Manager controls if a shared object X has been already loaded by another process Q, in order to utilize the same physical address of X for correctly updating the Relocation Table of P.

The case of PCBs, all of which exist in any VM, poses the question about the potential waste of virtual memory space. In order to evaluate this issue, let us fix some parameters with typical values of current systems. The number of simultaneously active processes is of the order of 10^3 in large systems. Typical PCB size is about 10^2 words. Thus, we could have 10^5 words reserved in all the VMs. Even in a 32-bit address machine, the fraction of VM space for replicated PCBs is quite negligible (order 10^{-5}), and it remains negligible also by taking into account all the kinds of shared objects. However, if this space could be useful, we are two basic solutions to the problem of saving the space of replicated objects:

1. to allocate the shared objects of the run-time support *dynamically in VM*,
2. to allocate the shared objects of the run-time support into a *single, additional address space*, separated from the address spaces of the active processes, and referable by any active process.

Solution 2 is the popular technique adopted in several commercial machines, based on the existence of a *kernel space* (supervisor space) distinct from the *user space* (problem space). As known from the operating system study, the utilization of the kernel space implies a meaningful overhead: in order to pass from the user space to the kernel space through a supervisor call, all the parameters must be transmitted by-value, which is an heavy time-consuming operation for large shared objects (like messages, and even PCBs and communication channels). Traditionally, this technique is justified by protection reasons according to a hierarchical approach. However, other more effective protection methods exist in which no hierarchical entity is provided ("peer-to-peer" cooperation).

Solution 1 is very elegant and more efficient, and, in primitive form, it is implemented in advanced architectures. For the majority of time, an object X exists statically in only one VM, and it is allocated dynamically in other VMs only when needed and for the time strictly needed. For example, by default PCB_P is statically allocated in VM_P only. When a distinct process Q needs to utilize PCB_P (e.g. in context-switching from Q to P) it acquires PCB_P in VM_Q in a free position, then PCB_P is used by Q with logical addresses, and finally PCB_P is released by Q when the procedure is completed. This method, called *capability based addressing*, can be implemented efficiently and supports a "peer-to-peer" protection.

A case of special importance when the shared objects are ***pointers***: It will be studied in the next Section, where we'll realize that it includes the problem of replicated objects too.

4.5 Shared pointers

If a shared data structure S_0 contains pointers to other data structures S_1, …, S_n, thus S_1, …, S_n themselves are shared by the same processes. For example:

- all the PCB pointers of the Ready List,

- PCB and message pointers in Channel Descriptors to implement the process synchronization and scheduling in interprocess communication primitives run-time support (see Section 6).

The problem is: how to refer S_1, …, S_n by any process, i.e how to implement shared pointers.

Three *static* methods exist:

1. Distinct Logical Addresses

as in the general case, each process sharing S_i refers S_i by its own logical addresses. The pointer in S_0 is implemented as a unique identifier, which is translated by each process into the correct logical address by means of a private table;

2. Coinciding Logical Addresses

S_i is referred with the same logical address by every process sharing S_i. This implies a less efficient exploitation of the logical address spaces, which inevitably contain "holes";

3. Physical Addresses

S_i is referred directly by the physical address. Though popular and apparently intuitive, this method is prone to implementation difficulties and inefficiencies:

- the distinction between logical and physical addresses must be visible at the assembler level too,
- serious constraints are imposed in the main memory allocation,
- no protection mechanism can be efficiently implemented, as in the case it is associated to the logical address translation (Sections 4.3.2, 4.3.3).

As discussed at the end of the previous Section, static methods are affected by some inefficiencies in the virtual memory exploitation. The Capability Based Addressing method solves the problem by implementing a *dynamic allocation of virtual memory* without sensible overhead compared to the static methods.

Capability Based Addressing

The shared pointer is implemented as a special object, called *capability*, that enables the acquiring process *to allocate space* and *to refer* the indirectly referred shared object by means of logical addresses.

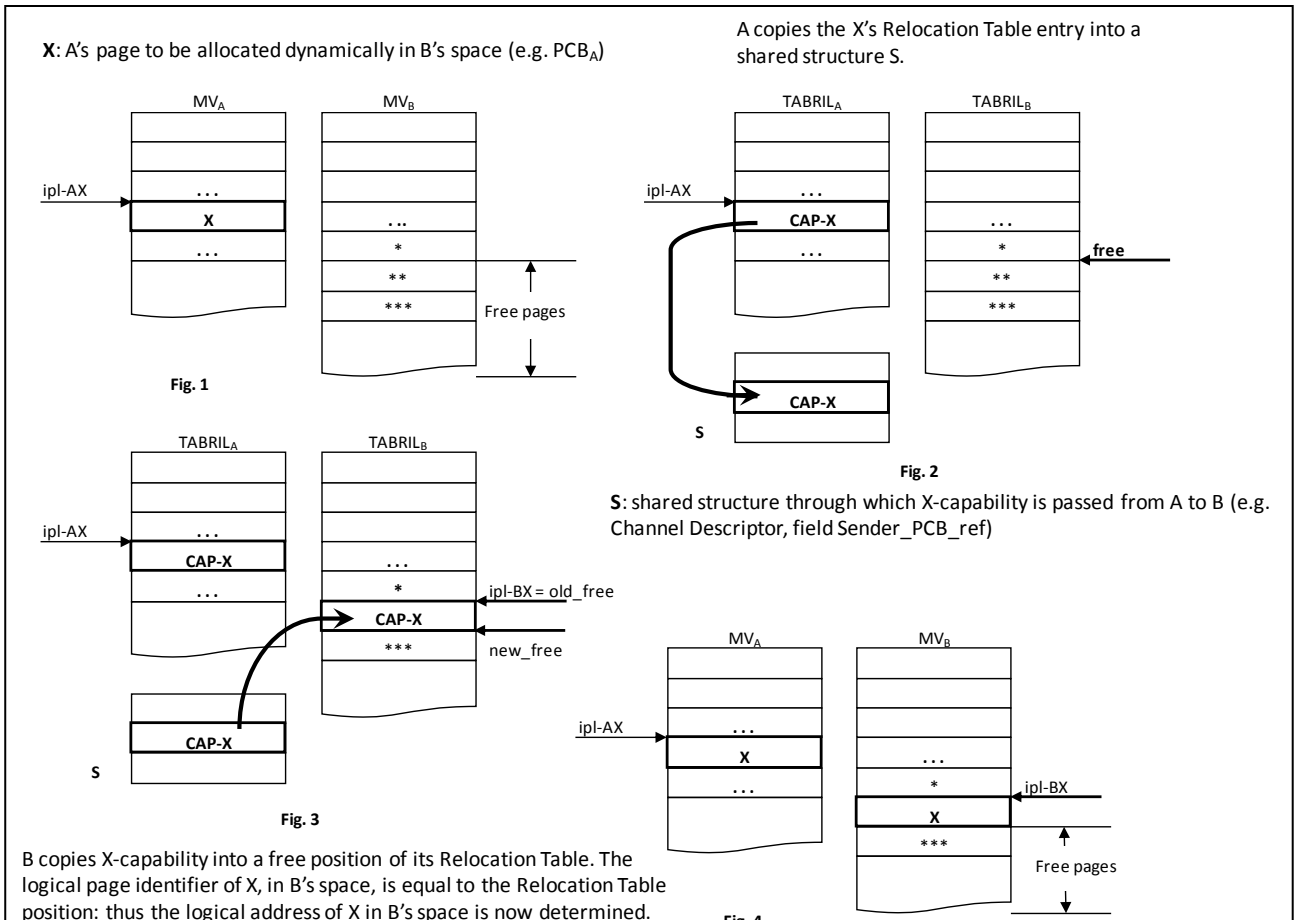
Formally, a capability is a couple

(object identifier, access rights)

plus a mechanism to generate the object reference, thus it is a technique related to the concept of Protected Addressing Spaces.

In practice, the implementation of a capability is the *entry of the Address Relocation Table* relative to the shared object. This entry is added to the Relocation Table of the acquiring process, thus allowing this process to determine the logical address of the shared object.

The method is described in the following figure, where an object X (for simplicity one-page size) is passed from a process A to a process B dynamically.



Let *ipl_AX* the logical page identifier of X in A's address space. The method assumes that the acquiring process B has some "free" pages (i.e. pages not allocated statically) into which the acquired object can be allocated (fig. 1). A (statically allocated and directly referred) shared structure S is used to pass the capability.

The method consists of the following steps:

1. A copies the capability of X (the $TABRIL_A$ entry in position *ipl_AX*) into S (fig. 2). Proper protection rights for B are decided by A;
2. B copies the capability of X from S into a free position of $TABRIL_B$ (fig. 3). Let *ipl_BX* such position;
3. at this point, the logical page identifier of X in B's address space is just *ipl_BX* (fig. 4). A very simple calculation is done in order to provide the base logical address of X in B's space: if *d* is the DISPL field size, right shift *ipl_BX* by *d* positions.

B is now able to refer X by its own logical addresses and the proper access rights. When B has finished to use X, the object can be released by annulling the $TABRIL_B$ entry and updating the free position.

The following operations are defined on a capability:

- *Transmit_capability (C, S, new_rights):* copy capability C, with the protection rights field modified according to the new_rights value, from the process Page Relocation Table into a shared structure S,

- *Read_capability (C1, S, C2):* read capability C1 from a shared structure S and save it in a local variable C2,
- *Acquire_capability (C, Addr):* write capability C into the process Page Relocation Table into the "Free" position, generate a base logical address Addr = concatenation (logical page identifier = Free, offset = zero), and update Free,
- *Release_capability (C):* cancel capability C in the process Page Relocation Table (e.g. put the presence bit at false and the protection rights at null), and update Free.

Capability-based addressing is efficient because the dynamically allocated object is *already in physical memory*. Few Load/Store instructions are sufficient to perform the dynamic allocation.

Capability-based addressing is "efficiently protected", because *the permission on X has been passed explicitly by the owner*. Moreover, as for any other object, the access to X is filtered by the *MMU* where the address translation *and* the protection checks are performed in a single clock cycle.

Finally, the importance of this technique lies in the absolute *necessity* (not merely convenience) to allocate objects dynamically in some specific situations, i.e. when for the compiler is impossible to statically predict the request of some kinds of run-time support objects. Notable examples will be met in Part 2.

We can see that the method is a very general one for dynamic virtual memory allocation; for example, it can be used to implement language mechanisms like *new/malloc*. It is characterized by low overhead in time (about two memory copies), and solves all the problems traditionally approached with the kernel space (Solution 1 *vs* 2 at the end of Section 4.4). Some advanced systems provide this method as a primitive mechanism.

5. Memory hierarchies and cache architecture

In this Section we complete the memory hierarchies treatment and study the main memory – cache hierarchy in detail: cache architecture, caching levels, performance evaluation, and code optimizations.

5.1 Memory hierarchies

The principle of memory hierarchy has been introduced in Section 4.3.1. It has been applied to the VM-M hierarchy in the rest of Section 4. Now we complete the general treatment of memory hierarchy. The successive sections will apply the concepts to the M-Cache hierarchy and its optimized utilization.

The memory hierarchy principle is a fundamental one in Computer Science: it extends from virtual memory and cache architecture to file systems and data bases, as well as to the internet services, in order to optimize the ratio between the information access latency and the information storage cost. Often a ***paged hierarchy*** is implemented, where objects are decomposed into fixed size blocks, which represent the *transfer unit* between the memory levels of the same hierarchy M2-M1, where M2 is the upper level.

In the following we'll denote by γ the capacity of a memory level (γ_1 or γ_2), by σ the page size (by definition, the same for M2 and M1), and by $\nu = \gamma/\sigma$ the number of pages of a memory level. Typical values of these parameters for VM-M and for M-Cache have been mentioned in Section 4.3.1.

5.1.1 Locality and reuse

As introduced in Section 4.3.1, locality and reuse are the basic properties to be exploited to optimize the utilization of a memory hierarchy. Let us define them formally. Consider the following simple computation:

```
int A[N];
        ∀ i = 0 .. N-1
                A[i] = A[i] * A[i]
```

compiled as:

```
LOOP:   LOAD  RA, Ri, Ra
        MUL   Ra, Ra, Ra
        STORE  RA, Ri, Ra
        INCR  Ri
        IF<  Ri, RN, LOOP
        END
```

Let us observe a trace of logical addresses generated by the program execution, assuming that instructions are allocated in MV_A starting at address 0 and A at address 1024. The initial prefix of the trace is the following (in absence of interrupts and exceptions):

0, 1024, *1*, *2*, 1024, *3*, *4*, *0*, 1025, *1*, *2*, 1025, *3*, *4*, *0*, 1026, *1*, *2*, 1026, *3*, *4*, *0*, 1027, …

We observe that, inside a sufficiently wide temporal window, *some* (two in the example) *address sequences are intermixed*:

[0, 1, 2, 3, 4]

[1024, 1024, 1205, 1025, 1026, 1026, 1027, …]

In any sequence addresses are relatively close one another compared to the size of the temporal window; in the example, they are consecutive or equal two-by-two. This concept corresponds to the ***locality*** property (also called *spatial locality*).

Moreover, some sequences are repeated several times in the temporal window (in the example, the first sequence), or some references are repeated in the temporal window (as in the second sequence of the example). This concept corresponds to the ***reuse*** property (also called *temporal locality*).

By organizing information in pages, the probability that referred information belong to a limited number of pages, and that the referred pages vary slowly, is relatively high. In our example, the same code page is used for the whole duration of program execution, while the same page of data is used for 2σ consecutive references to A.

Thus, in the example:

- once the code page has been loaded into the lowest level of the hierarchy, the transfer time is no more paid provided that we are able to impose that this page is not replaced. This is the way to exploit the reuse property;
- once a page of A has been loaded into the lowest level of the hierarchy, the transfer time is no more paid for 2σ references to A, provided that such page is not replaced during the time interval of σ iterations. This is the way to exploit the locality property.

The total number of faults during the program execution is given by:

$$1 + N/\sigma \sim N/\sigma$$

The importance of reuse

The following example shows a computation in which potential reuse exist for the data too:

```
int A[N], B[N];
    ∀ i = 0 .. N-1
        ∀ j = 0 .. N-1
            A[i] = F (A[i], B[j])
```

The pages for the code (their number depends on the compilation of function *F*) are characterized by reuse.

The pages of A and B are characterized by locality. Moreover, the pages of B are characterized by reuse too: for each outermost iteration, the *N* innermost iterations refer all B's elements. Depending on the capacity of the lowest level M1, reuse is exploited provided that all B's words (*N* words) can be contained in M1 for the whole duration of the program execution.

For example, consider a M-Cache hierarchy where the cache capacity is γ_2 = 64K words and the block size is σ = 16 words, thus ν_2 = 4K blocks (the term *block* is used for the M-Cache hierarchy as synonymous of page). Assume that the D-RISC code consists in 128 instructions, and that *N* = 32K. The code allocation requires 8 blocks, B allocation requires 2K pages. *If proper mechanisms are provided at the assembler and firmware level*, then all the blocks of code *and* of B can be maintained permanently in cache once loaded for the first time. For array A, it is sufficient that only one block at the time is present in cache (the current block). As a results, 2K + 9 (~ 2K) blocks over

4K are used by this program execution. In this condition locality and reuse are exploited optimally. The total number of cache faults is given by

$$8 \text{ (for code)} + N/\sigma \text{ (for A)} + N/\sigma \text{ (for B)} = 2\ N/\sigma + 8 \sim 2\ N/\sigma$$

However, if N = 128 K, then the potential reuse on B *cannot* be exploited, since B cannot be contained entirely in cache. In this case, the number of cache faults for B is given by:

$$\sim N^2/\sigma$$

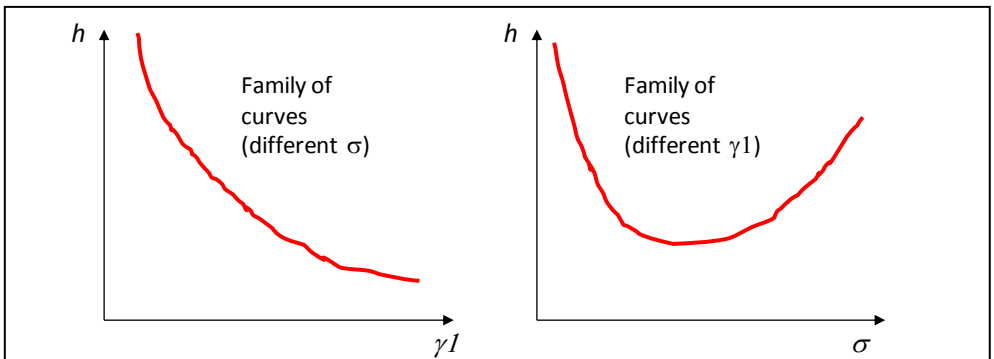
since *only the locality property* is exploited for B. *The total number of cache faults for the program is one order of magnitude greater than in the case in which reuse can effectively be exploited.*

Notice that, in the latter case, a very marginal, or even no, advantage is achieved by maintaining in cache more than one block of B at the time.

The explained importance of reuse in program optimizations opens a very interesting area of research about *cache-aware* algorithms and computations. In our example, if B is too large, in principle it is possible to think about a different algorithm organized as a sequence of steps, during each of which only a smaller fraction of B is employed, so that reuse can be exploited during every step, thus for the whole computation (*block-structured algorithms*). This is a case of cache-awareness: the programmer is aware of the cache performance problems, and designs the application with proper algorithms. In alternative, the compiler could restructure the program by adopting a different algorithm: a much more open field of research. By now, the importance of these issues in real applications has been widely recognized and justify intensive efforts of pure and applied research.

5.1.2 Optimal page size

As introduced in Section 4.3.1, conceptually the fault probability h, as a function of γ_1 and σ, has the following qualitative shape:



Experimental values for large benchmarking program sets in given application areas

While the function $h(\gamma_1)$ is quite self-explicative, the analysis of the function $h(\sigma)$ deserves more attention. The existence of a *minimum*, thus of an *optimal page size*, is consistent with the locality and reuse properties:

- on one hand, by increasing σ the locality inside every single page is improved,
- however, by increasing σ the number of pages in M1 decreases and, under some values, this number is not sufficient to simultaneously maintain all the pages belonging to the intermixed sequences.

For example, in a computation like:

int A[N], B[N], C[N], D[N];

$\forall$ *i = 0 .. N-1*

A[i] = F (A[i], B[i], C[i], D[i])

we have five intermixed address sequences (belonging to the code, A, B, C, and D), thus the block size must be such that at least five blocks are contained in the lowest level of the hierarchy.

Large benchmarks of programs for given application areas provide values of σ and γ_1, also according to the architectural characteristics of the memory supports. See the typical values mentioned in Section 4.3.1 for MV-M and M-Cache hierarchy, e.g.

- σ = 1K and γ_1 = 1G for MV-M (note: 1G per process),
- σ = 8 and γ_1 = 64K for M-cache.

Such values are very rough average evaluations that give just an order of magnitude. Each computation could have its own optimal values of σ and γ_1, however these values cannot be exploited in practice, since σ and γ_1 must be necessarily fixed for each architecture at the firmware level. We can say that the typical values obtained with the benchmarks are just used for designing the firmware architecture. It is a task of the compiler/programmer to exploit the firmware level at best.

5.1.3 Working set

Given a M2-M1 hierarchy, the working set of a program for such hierarchy is defined as the set of pages (blocks) which, if simultaneously present in M1, minimize the fault probability.

This concept is a quite fundamental one for the optimization of programs, since, in modern architectures, the page/block transfer time is one of the most important sources of performance degradation. Thus, the objective of an optimizing compiler is to recognize *how many* and *which* pages/blocks belong to the working set, and to verify whether the working set pages can be effectively maintained in M1.

In the examples of Section 5.1.1, the working set WS for the M-Cache hierarchy is evaluated as follows:

- first example (single loop on A): WS is given by the code block and the current block of A. This very small working set can be effectively exploited in any machine;
- second example (double nested loop): WS is given by 8 blocks for the code, the current block of A, and all the blocks of B (total: $\sim N/\sigma$ blocks). This working set is effectively exploited *according to the cache capacity*: the compiler knows this architectural characteristics (*the compiler knows all the most relevant architectural characteristics under the form of the abstract machine definition*: see Section 1, and 1.6 in particular), thus it is able to verify whether the program is implicitly optimized (at least from the cache exploitation point of view), or more sophisticated optimizations and program restructurings have to be possibly implemented (cache-aware approaches).

The impact of the architecture and of the *specific* memory hierarchy is very important in studying the working set of a program.

Consider again the second example from the point of view of the *VM-M* hierarchy. This situation has strong differences compared to the M-Cache hierarchy. First of all, to be executed the program must have been already loaded into M (Section 4.2), while the program is loaded into the cache only when it is running. For the VM-M hierarchy, the working set includes the whole program

(code, A, and B), since this is the situation that minimizes the number of main memory faults. The system will try to load the whole program into M at loading time, thus implicitly satisfying also the reuse requirements for B. Of course, this is feasible because of the rather large capacity of the main memory. Possibly, some *annotations* are associated to applications (using the configuration file, Section 4.2) in order to characterize them from the point of view of the main memory requirements. The process creation and loading will be affected by this feature: possibly, if the required conditions cannot be satisfied, the process creation is postponed or some main memory pages are subtracted to other lower-priority processes.

5.1.4 Paging on-demand *vs* prefetching

All the previous examples have been done under the assumption that (for a M2-M1 hierarchy) a page/block is loaded into M1 only if and when it is affected by a fault. This is the basic, and most common, hierarchy architecture, called ***on-demand*** paging.

In principle, it is possible to think about a more efficient strategy by loading pages/blocks in advance wrt their effective references. For example, in the first example of Section 5.1.1, a static analysis of the computation shows that, once the *i-th* block is loaded into M1, the next block to be used will be the *(i+1)-th* one. Thus, a proper architecture can provide to load the *(i+1)-th* block in parallel, while the program is working on the *i-th* one. In this example, the number of fault reduces to zero (more precisely, to 2). Analogous considerations apply to the second example. This strategy is called ***prefetching***.

Provided that the firmware architecture is able to load new blocks in parallel with the access to the currently loaded ones, prefetching technique has an impact on the assembler level and the compiler. That is, the compiler must detect that a certain form of prefetching can be applied and insert *special instructions or annotations* to the code.

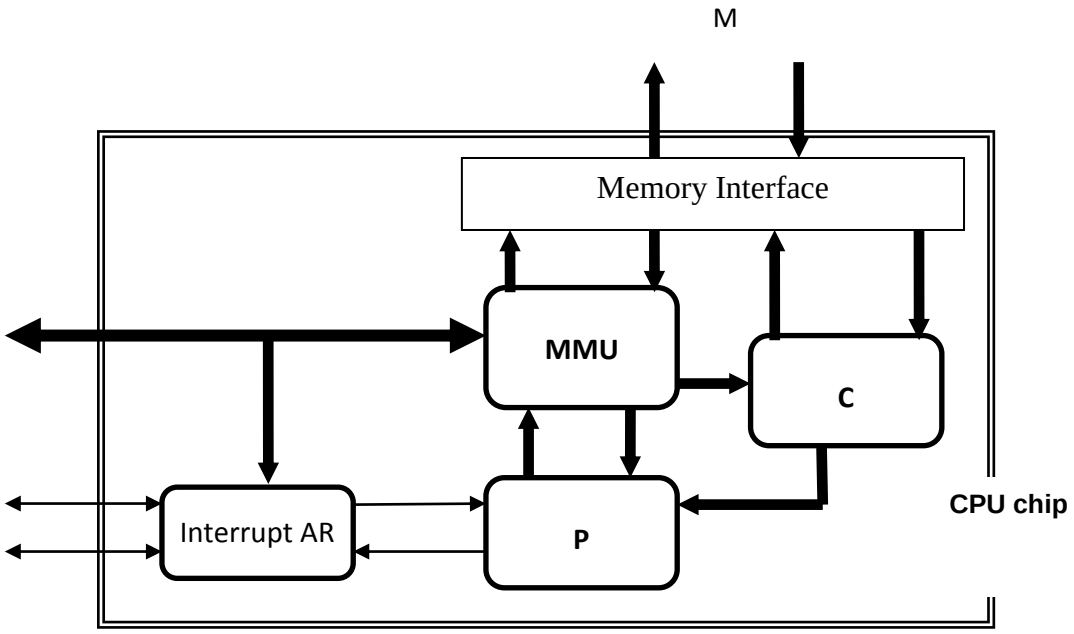
Though interesting, the prefetching strategy is not so simple to apply and implements. Not all the programs have simple characteristics as the ones considered in the previous examples. In general, after the *i-th* block the computation could require the *j-th* one, with $j \neq i+1$, or even *j* variable. Not only this situation must be detected and *j* possibly evaluated, but the *i vs j* relation must be represented at the assembler level. Many machines adopt a "by-default", i.e. automatic, prefetching strategy where the *(i+1)-th* block is *always* loaded in parallel to the *i-th* one utilization, without a compiler analysis. In fact, several benchmarks exist showing that some programs are under-optimized, because some useless blocks are loaded and subtract space to other useful blocks. In the second example of Section 5.1.1, applying prefetching to A can subtract useful (and much more important) space to B.

Automatic prefetching with $j = i+1$ can be effectively applied to *instructions*, and in fact this technique is adopted in many machines. In the same way, *reuse* to instructions is applied automatically, at least for small-medium size loops.

5.2 Cache memory

The following figure shows a possible scheme of the CPU with the elementary architecture of Section 3.4.

The cache processing unit C contains the *cache memory logical component* MC (e.g. 32K - 128K capacity) and all the needed hardware resources in the Operation Part. After the successful logical address translation, MMU sends the request, including the main memory physical address, to C.



The cache unit provides to translate the main memory physical address into the cache address and to serve the request sending the access result to P. In case of *cache fault* (or *cache miss*) C requests the cache block to the main memory and stores it into a MC block, in general by replacing an existing block through a proper replacement strategy. The cache fault and the corresponding block transfer is *fully invisible to the processor*, that merely waits the access results from C (an MMU exception is signaled to P by the MMU itself or through C).

5.2.1 Treatment of writing operations

Two main methods are used to implement writing operations in a memory hierarchy:

- ***Write Back***: a modified block is copied into the upper memory level only when it is replaced;
- ***Write Through***: every writing operation is executed simultaneously in cache and in the upper memory level.

Write Back is used also in the MV-M hierarchy, while Write Through cannot be applied in MV-M for obvious latency reasons. In the latter method, MMU can send the memory request both to the cache unit and to the upper memory level.

In cache-based architectures, because of the higher access latency of the main memory, Write Through requires that the writing operations are not implemented in a request-reply manner, i.e. the writing operation is launched but the final outcome is not waited by the processor/MMU. Moreover, we'll see that the program completion time might be affected by problem of insufficient memory bandwidth.

Another issue related to writing operation concern the ***evaluation of the cache faults*** when a writing operation is executed. Let us consider the following example:

int A[N], B[N];

$\forall$ *i = 0 .. N-1*

A[i] = F (B[i])

Array A is used in a "write-only" mode: thus, when a cache fault on an A element occurs, it is not necessary that the corresponding block is transferred from the upper memory level into the cache; it is sufficient that the block is allocated in MC *without any block transfer*. The cache unit can be implemented exactly in this way, and the time overhead for handling such faults is negligible.

For this reason, though this program has $2N/\sigma$ faults for data (in an on-demand cache strategy), only N/σ block transfers occur. By convention, for the sake of performance evaluation, we will evaluate the number of faults as

$$N/\sigma$$

since we are interested in evaluating only the cache faults causing block transfers.

If the cache unit behaves in this way for the writing operations (i.e., no block transfer is requested), the compiler must take care to distinguish between write-only data structures and data that, at least with a non-null probability, can be read *and* written. If the first operation on such data is a reading one, there is no problem: the read fault causes the block to be loaded into the cache. For example:

```
int A[N], B[N];
        ∀ i = 0 .. N-1
                A[i] = F (A[i] , B[i])
```

Each element of A is first read then written, thus the read operation causes the block transfer. In general, the correctness sufficient condition for read-write blocks is that *a block reading operation must the executed first*.

In fact, if the first operation on such block is *not* a reading one, there are cases in which the referred data are inconsistent. For example, let us consider the following execution sequence on a certain $block_h$, in which the *i-th* position is written before the *j-th* position is read, with $j \neq i$,

```
….
write blockh[i]
read blockh[j]
…
```

and the writing operation is *not* preceded by a reading operation on the same block. The fault occurring on the writing operation causes the block to be allocated in cache without transferring it, thus the reading operation returns an inconsistent value for the *j-th* position, which has not been read from the main memory.

In conclusion, if the cache unit is designed with the write-only optimization, then for read-write blocks the compiler must ensure that the first operation on the block is a reading one: if it is not, then the compiler must *force* such reading by restructuring the code in such a way that writing of an element is preceded by a reading operation on an element of the same block (a LOAD instruction is sufficient). For example, the correct code for the previous example could be:

```
….
read blockh[0]
write blockh[i]
read blockh[j]
…
```

so that the fault is caused by the *read* $block_h[0]$ operation and the block is transferred from the main memory into the cache.

5.2.2 Address translation methods

Three main methods exist for translating main memory addresses into cache addresses, characterized by precise *correspondence laws between main memory blocks and cache blocks*. Such method have to be efficiently implemented in the firmware design of the cache unit. Correspondence functions can be defined in an analytic way (the so called *direct* cache), or in a table-based way (*associative* cache), or a mix of these two solutions (*set associative* cache).

In the following, we denote by *BM* the identifier of a block in main memory, by *BC* the identifier of a block in cache, and by *D* the displacement of the information inside a block. For example, with a main memory of 4G maximum capacity (32-bit address), a cache of 64K capacity (16-bit address), and blocks of 8 words (3-bit displacement D), the main memory address is seen as the concatenation (BM, D), where BM is 29-bit wide (512M blocks in main memory), and the cache address as (BC, D) BC is 13-bit wide (8K blocks in cache).

1. Direct method

In this method, a given main memory block corresponds to, i.e. it can be transferred into, a well-defined cache block. A simple and efficient function is the following one:

$$BC = BM \ \textbf{mod} \ NC$$

where *NC* denotes the number of cache blocks. If, as usually, *NC* is a power of 2, then BM can be seen as (TAG, BC), where TAG = MB ***div*** NC. Thus, if the referred block is currently allocated in cache, the C address (BC, D) is just a part of the M address. For this reason, the address translation *per se* is very simple.

The address translation must be coupled with the verification that actually the referred block BM is the block currently allocated in the cache block BC = BM *mod* NC. For example, if the M address is (8K, D) the M block identified by BM = 8K corresponds to the BC = 0 cache block. If currently the 0-th C block contains just the 8K-th M block, then the (BM.BC, D) configuration is the correct cache address and the cache access is done in a single clock cycle. Otherwise, a cache fault condition occurs, and the 8K-th M block must be transferred into the 0-th cache block.

This verification is done by comparing the BM.TAG field of the M address with the TAG of the block currently allocated in the BM.BC cache block. Notice that the TAG information uniquely identifies the M block corresponding to the BM *mod* NC cache block. The implementation uses a register memory TAB, of *NC* capacity, in the cache unit Operation Part. Each location of TAB contains the TAG of the currently allocated block, if any, a presence bit, and other information for cache management (modified block, reuse, and so on). The location in the BC position is compared with BM.TAG in a single clock cycle.

Moreover, the verification and the cache access can be done in parallel in the same clock cycle, Thus, in this method the cache unit service time and latency is equal to τ. Taking into account the MMU latency, the cache access latency as seen by the elementary processor is given by:

$$t_C = 2\ \tau$$

The direct method is very efficient and simple to be implemented. However, the correspondence law is very “rigid”: in general, the consequence is an increased fault probability when more than one data structure is allocated in blocks corresponding to the same cache blocks.

Only some particular information has the property that such problems don’t arise, notably the *instructions* of a program. For data, we need a more flexible method.

2. Associative method

The maximum flexibility is achieved if the correspondence law is fully random, that is each M block can be dynamically allocated into *any* C block. The block replacement algorithm, e.g. LRU, assumes a special importance in this case.

As we know, this is exactly the method used in the MV-M hierarchy. Notice that any other method cannot be applied to the MV-M hierarchy, because M contains pages of more than one process.

The implementation of the associative method requires the utilization of an *associative memory* of *NC* locations (see Section 4.3.3) in the Operation Part of the cache unit. The key is BM and the returned information is BC, if the BM block is currently allocated in cache.

The cache unit service time and latency is equal to 2τ, since the associative memory access and the cache memory access are sequential. Thus, taking into account of the MMU latency:

$$t_C = 3\ \tau$$

is the ideal cache access latency for the elementary processor.

Compared to the direct method, in the associative method the maximum flexibility is paid with an increased access latency and with the utilization of a more complex hardware component (associative memory in place of a normal RAM). This issue of hardware complexity was relevant during the previous generations of computer architecture, while now it has an almost negligible impact.

3. Set associative method

If the hardware complexity issue is considered relevant, the trade-off between the direct and the associative method consists in a "mixed" correspondence law: in part analytic and in part associative. Let us logically organize the cache blocks into *sets*, for example 4 consecutive blocks per set. The analytical correspondence is established between M blocks and C sets, i.e. a given M block can be allocated in a given C set only. Inside the C set, the M block can be allocated in any block.

By denoting by *NS* the number of C block sets, and by *SET* the identifier of a C set, the direct correspondence can be efficiently defined as

$$SET = BM \ \textbf{mod}\ NS$$

If *NS* is a power of 2, SET is simply given by the least significant $log_2 NS$ bits of BM. The method implies that, if the referred M block is currently allocated in the C set identified by SET, then we have to search for the block, inside the set, in which the referred block is allocated. Let TAG = BM ***div*** NS. The search is efficiently implemented with a RAM table, in the cache unit PO, each location of which contains NC/NS TAGS of the currently allocated M blocks, plus the usual C management information and presence bits. This table is acceded by SET and the read NC/NS TAGS are simultaneously compared with BM.TAG. If no comparison is successful, then a cache fault is detected. Otherwise, the position (say J) with the successful comparison uniquely identifies the block inside the set, thus the cache address is given by the concatenation (SET, J, D).

The cache access latency for the elementary processor is again:

$$t_C = 3\ \tau$$

The hardware complexity is comparable to the direct method, while the flexibility depends on the number of blocks per set NC/NS. In large benchmarks, it has been verified that 4-8 block per set are sufficient to achieve a fault probability close to the associative method. However, there are

programs in which the NC/NS parameter is critical, i.e. when the program works simultaneously on more than NC/NS information corresponding to the same set.

As a conclusion, usually the cache memory is decomposed into two units: one for instructions only, and the other for data only. The ***Instruction Cache*** is a direct one, while the ***Data Cache*** is associative or set associative. This decomposition will be really exploited in parallel CPUs to increase the parallelism degree, while in the elementary processor it is just a technique to optimize the fault probability.

5.2.3 Cache optimizations and annotations at assembler level

In Sections 4.3 and 5.1 we have seen that often, in order to minimize the fault probability, the optimal exploitation of the locality and reuse properties requires the introduction of special instructions or annotation in the assembler code at compile time.

Notably, the D-RSC machine provides the following annotations:

LOAD …, …, …, **prefetching**

LOAD/STORE …, …, …, **don't_deallocate**

LOAD/STORE …, …, …, **deallocate**

LOAD/STORE …, …, …, **rewrite**

As usually, the firmware interpreter of the cache unit provides to implement them.

In D-RISC, "prefetching" is referred to the next consecutive block only. Other more complex functions could be provided (non in D-RISC), at the expense of space in instruction format and firmware complexity.

The "don't_deallocate" is used for *reuse optimizations*: the referred block is maintained in cache memory without replacing it.

For example, if the B reuse can be applied to the kind of programs of Section 5.1.1:

int A[N], B[N];

$\forall i = 0 .. N-1$

$\forall j = 0 .. N-1$

$A[i] = A[i] * B[j]$

the compiled code is the following:

```
LOOPi :   LOAD   RA, Ri, Ra
          CLEAR  Rj
LOOPj :   LOAD   RB, Rj, Rb, don't_deallocate
          ADD    Ra, Rb, Rc
          STORE  RA, Ri, Ra
          INCR   Rj
          IF <   Rj, RN, LOOPj
          INCR   Ri
          IF <   Ri, RN, LOOPi
          END
```

A typical example of prefetching annotation, provided that the cache is properly designed, is just:

```
int A[N], B[N], C[N];
        ∀ i = 0 .. N-1
                C[i] = A[i] + B[i]
LOOP:   LOAD   RA, Ri, Ra, prefetching
        LOAD   RB, Ri, Rb, prefetching
        ADD    Ra, Rb, Ra
        STORE  RC, Ri, Ra
        INCR   Ri
        IF <   Ri, RN, LOOPi
        END
```

Explicit block deallocation and explicit block rewriting in main memory will be studied in Part 2 about multiprocessor architectures.

Some machines provide also a *FLUSH* instruction to reinitialize the cache tables containing the correspondence function between M blocks and C blocks (Section 5.2.2).

5.2.4 Performance evaluation

The completion time of a program will be evaluated as follows:

$$T_c = T_{c\text{-}id} + T_{fault}$$

where

- $T_{c\text{-}id}$ is the completion time *in absence of cache faults*, i.e. for the elementary computer by evaluating the memory access latency as the cache access latency t_C;
- T_{fault} is the *time overhead* (penalty) *paid for the blocks transfers* caused by cache faults. It is evaluated as:

  $$T_{fault} = N_{fault} * T_{transf}$$

  where N_{fault} is the ***average number of cache faults*** of the program, and T_{transf} is the ***latency of a block transfer*** into the cache: the former is dependent on the program and its compilation, the second is an architectural characteristic.

The degree of utilization of a cache architecture is measured by the ***relative efficiency***:

$$\varepsilon = \frac{T_{c-id}}{T_c}$$

The cache utilization is as better as ε approaches one.

For example, consider the array addition program without prefetching:

```
LOOP:   LOAD   RA, Ri, Ra
        LOAD   RB, Ri, Rb
        ADD    Ra, Rb, Ra
        STORE  RC, Ri, Ra
        INCR   Ri
        IF <   Ri, RN, LOOPi
```

By using the D-RISC instruction service times of Section 3.5, we have:

$$T_{c\text{-}id} = N\,(\,6\,T_{ch} + 3\,T_{ex\text{-}LD} + 2\,T_{ex\text{-}ADD} + T_{ex\text{-}IF}) = N\,(22\,\tau + 9\,t_C)$$

Assuming an associative or set associative cache, with $t_C = 3\,\tau$:

$$T_{c\text{-}id} = 49\,N\,\tau$$

Without prefetching, $\sigma = 8$ and automatic instruction reuse (implicit "don't deallocate"):

$$N_{fault} = N_{fault\,-instr} + N_{fault\,-A} + N_{fault\,-B} = 1 + \frac{N}{\sigma} + \frac{N}{\sigma} \sim 2\,\frac{N}{\sigma}$$

Array C is write-only, thus we don't evaluate its faults because, with a proper cache unit design, they don't cause block transfers (Section 5.2.1). N_{fault}, thus T_{fault}, is of the same order of magnitude $O(N)$ of the ideal completion time: thus, the relative efficiency is highly dependent on T_{transf}.

With prefetching

$$N_{fault} = 3$$

thus negligible compared to the ideal completion time, independently of T_{transf}. In this case, $\varepsilon \sim 1$.

We'll evaluate T_{transf} in the next section.

Let consider the other example of the previous section:

```
int A[N], B[N];
        ∀ i = 0 .. N-1
                ∀ j = 0 .. N-1
                        A[i] = A[i] * B[j]
```

```
LOOPi :     LOAD   RA, Ri, Ra
            CLEAR  Rj
LOOPj :     LOAD   RB, Rj, Rb
            ADD    Ra, Rb, Rc
            STORE  RA, Ri, Ra
            INCR   Rj
            IF <   Rj, RN, LOOPj
            INCR   Ri
            IF <   Ri, RN, LOOPi
```

The completion time is clearly dominated by the innermost loop:

```
LOOPj :     LOAD   RB, Rj, Rb
            ADD    Ra, Rb, Rc
            STORE  RA, Ri, Ra
            INCR   Rj
            IF <   Rj, RN, LOOPj
```

whose completion time is:

$$T_{innermost\text{-}id} = N\,(\,5\,T_{ch} + 2\,T_{ex\text{-}LD} + 2\,T_{ex\text{-}ADD} + T_{ex\text{-}IF}) = N\,(18\,\tau + 7\,t_C)$$

Therefore:

$$T_{c\text{-}id} \sim N\, T_{innermost\text{-}id} = N^2\,(18\,\tau + 7\,t_C)$$

If *B reuse cannot* be applied because of the excessive size of B, then:

$$N_{fault} = N_{fault\,-instr} + N_{fault\,-A} + N_{fault\,-B} = 2 + \frac{N}{\sigma} + \frac{N^2}{\sigma} \sim \frac{N^2}{\sigma}$$

which is of the same order of magnitude of the ideal completion time.

If reuse *can* be applied (LOAD RB, Rj, Rb, **don't_deallocate**):

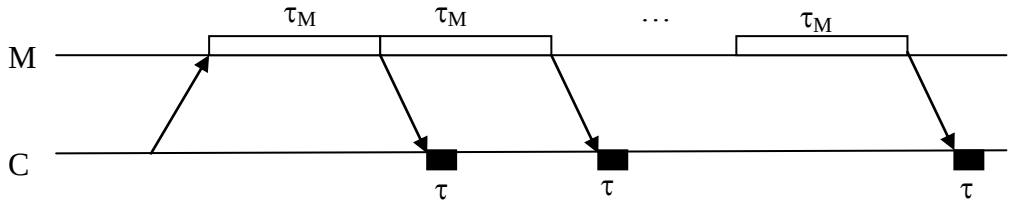
$$N_{fault} = N_{fault\,-instr} + N_{fault\,-A} + N_{fault\,-B} = 2 + \frac{N}{\sigma} + \frac{N}{\sigma} \sim 2\,\frac{N}{\sigma}$$

thus $T_{fault} = O(N)$, which is negligible compared to the ideal completion time $O(N^2)$. The important result is that, if reuse can be applied, the completion time is minimized and $\varepsilon \sim 1$.

5.2.5 Block transfer and memory hierarchy organization

As seen in the previous examples, the block transfer latency plays a secondary role only when N_{fault} is of an order of magnitude less than $T_{c\text{-}id}$. In all the other cases, T_{transf} is very critical for performance evaluation: thus, a general-purpose architecture must be able to satisfy the worst case, i.e. to minimize T_{transf}.

Let us consider a *traditional organization of the main memory* with relative bandwidth of one word per access time, i.e. a main memory subsystem realized by a single unit. Let us assume that a single block-transfer request can be done by the CPU to M, instead of a sequence of σ reading requests. The timing behavior is:



Since the writing operations into cache, except the last, are overlapped to the main memory clock cycle, we have:

$$T_{transf} = 2\,T_{tr} + \sigma\,\tau_M + \tau \sim 2\,T_{tr} + \sigma\,\tau_M \sim \sigma\,\tau_M$$

$$T_{fault} = N_{fault} * T_{transf} \sim \sigma\,\tau_M N_{fault}$$

Consider the array addition example without prefetching (a case which is T_{transf} *sensible*):

$$T_{c\text{-}id} = 49\,N\,\tau$$

$$N_{fault} \sim 2\,\frac{N}{\sigma}$$

$$T_{fault} \sim 2\,N\,\tau_M$$

With $\tau_M = 20\,\tau$, $T_{fault} = 40\,N\,\tau$, which is almost equal to $T_{c\text{-}id}$, thus $T_c = 89\,N\,\tau$ and $\varepsilon = 0.55$.

With $\tau_M = 50\,\tau$, $T_{fault} = 100\,N\,\tau$, which even greater to $T_{c\text{-}id}$, thus $T_c = 149\,N\,\tau$ and $\varepsilon = 0.33$.

Similar qualitative results are obtained in general.

In conclusion, for T_{transf} *sensible* programs, low efficiency and high completion times are achieved with traditional organization of the main memory.

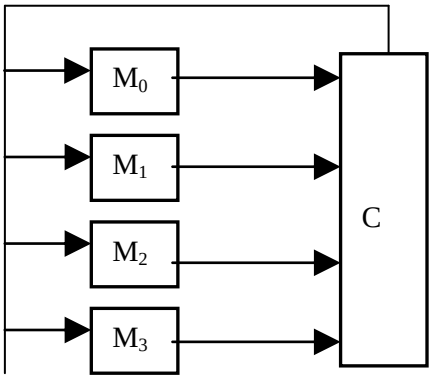
In other words, unless prefetching and/or reuse can be applied, high performances and good cache utilizations cannot be achieved by exploiting the locality property only with a traditional organization of main memory.

Thus, we must realize the main memory with (much) higher bandwidth in order to reduce the block transfer latency.

Interleaved memory

In Section 2.5 we studied high bandwidth memory organizations. In particular, the *modular interleaved memory* is the most suitable organization for our current purposes, as we need to access blocks.

In case of fault, the cache unit requests a block to the whole interleaved memory, whose modules returns all the block words in parallel:



If $m < \sigma$, more than one access must be done. Exploiting the memory-cache overlapping, the block transfer latency is given by:

$$T_{transf} = 2\,T_{tr} + \frac{\sigma}{m}\,\tau_M + m\,\tau \sim \frac{\sigma}{m}\,\tau_M$$

As it is intuitive, for $m = \sigma$ the block transfer latency is equal to a single word access latency, and

$$T_{fault} = N_{fault} * T_{transf} \sim \tau_M N_{fault}$$

In the previous example of array addition, for $m = \sigma$:

$$T_{fault} \sim \frac{2\,N\,\tau_M}{\sigma}$$

achieving a one order of magnitude reduction.

With $\tau_M = 20\,\tau$, $T_{fault} = 5\,N\,\tau$, thus $T_c = 54\,N\,\tau$ and $\varepsilon = 0.91$.

With $\tau_M = 50\,\tau$, $T_{fault} = 12.5\,N\,\tau$, thus $T_c = 61.5\,N\,\tau$ and $\varepsilon = 0.8$.

For larger blocks and, even more important, for slower memories, the *Ttransf* effect continues to remain relevant. In fact, because of the technology trend towards much larger main memories, and increasing differences between τ_M and τ, even the interleaved memory is not a definitive solution.

Secondary cache and more cache levels

The next improvement is to add another cache level, called ***secondary cache*** C2, while the cache described till now is called ***primary cache*** C1 (often the technical literature uses the symbols L1 and L2). C2 is interposed between M and C1 to implement the C2-C1 hierarchy. At the same time, C2 belong to the M-C2 hierarchy.

The C2 capacity is much higher than C1, typically 1M words as order of magnitude.

The C2-C1 blocks have the size discussed till now, while the M-C2 blocks are wider, for example 64-128 words. According to the general principles studied in Section 5.1.2, the M-C2 hierarchy has a fault probability substantially less than C2-C1.

The interleaved main memory is now exploited to reduce the transfer latency into C2.

While C1 contains information of the currently running process only, C2 contains blocks of *more than one process*. Moreover, C2 can prefetch blocks from M (blocks belonging to the running process or other processes) in parallel to the C1-P processing: this contributes to improve the performance of C2-C1 hierarchy, thus of whole system.

The advantage of this solution is that, with typical capacities mentioned above, the secondary cache can be implemented *on the same chip* of CPU. Thus, the C2-C1 link latencies are negligible, and C2 access time can be 1 or 2 clock cycles. We assume

$$t_{C2} = 2\,\tau$$

The block transfer exploit a full overlapping between C2 and C1: while C2 reads the *i-th* word of the block, C1 writes the *(i-1)-th* word. Thus:

$$T_{transf} \sim \sigma\, t_{C2} = 2\,\sigma\,\tau$$

$$T_{fault} = 2\,\sigma\,\tau\,N_{fault}$$

In the array addition example, assuming that the M-C2 fault probability is negligible:

$$T_{fault} = 2\,\sigma\,\tau\,N_{fault} = 4\,N\,\tau \qquad T_c = 53\,N\,\tau \qquad \varepsilon = 0.93$$

In order to further reduce the latency of block transfers into C2, the same principle of secondary cache can be extended to the ***tertiary cache*** C3 (L3), with a much larger capacity (e.g. 100M – 1G) and larger blocks (e.g. 128-256 words). With current technology, C3 could be integrated into the same CPU chip too, especially in multi-core chips, thus approximating the situation of having a "small" main memory on-chip! The same considerations about the multi-process memorization and the prefetching-parallelism behavior apply to C3-C2 and M-C3 too.

In multiprocessor architectures (Part 2) we'll see that the various caching levels can be associated to single processors or shared between processors in several interesting combinations.

5.2.6 Write-Through evaluation

Let us consider again the array addition example without prefetching on a Write-Through machine:

```
LOOP:   LOAD   RA, Ri, Ra
        LOAD   RB, Ri, Rb
        ADD    Ra, Rb, Ra
        STORE  RC, Ri, Ra
        INCR   Ri
        IF <   Ri, RN, LOOPi
```

Assume an architecture with secondary cache: M-C2 and C2-C1 are the memory herarchies to be considered.

A memory writing operation is caused by the STORE instruction at every iteration. In a Write-Through architecture, MMU can send the writing request to C1, C2 and M in parallel, without waiting for the outcome reply from C2 and M. While the bandwidth of C2 is sufficiently high to absorb the request load, it is possible that at least M become a bottleneck.

The CPU (MMU) sends a writing request every T_{iter}, i.e. the completion time of an iteration. Thus, the bandwidth for memory writing request is

$$B_w = \frac{1}{T_{iter}}$$

Denoting by B_M the memory bandwidth, if

$$B_w > B_M$$

M is a bottleneck that will delay the request stream. Thus, the completion time increases: it is evaluated by replacing T_{iter} by $1/B_M$.

In our example:

$$B_w = \frac{1}{49\ \tau}$$

With a traditional sequential main memory:

$$B_M = \frac{1}{\tau_M}$$

If $\tau_M > 49\ \tau$, M is a bottleneck, so the ideal completion time becomes

$$T_{c-id} = N\ \tau_M$$

With an interleaved memory:

$$B_M = \frac{m}{\tau_M}$$

thus M is bottleneck if $\tau_M > 49\ m\ \tau$, a condition that is more unlikely for large m.

5.3 Exercises

1. Describe and explain the Virtual Memory structure for the process corresponding to the array addition program.
2. A cache unit has a service time of 1 or 2 clock cycles (depending on the caching technique). Verify and explain this assertion.
3. Explain why fault handling for the MV-MM hierarchy is visible to the Processor, while for the MM-Cache hierachy it is not.
4. Assume that Secondary Cache resides on CPU chip. Why distinguishing between Primary Cache (L1) and Secondary Cache (L2) ? *i.e.*, why not just one level of cache only, with larger capacity (sum of capacity of L1 and of L2) ?

6. Interprocess communication and its run-time support

In order to develop and to exemplify our methodology to the process level and its run-time support, we need to refer to some specific mechanisms for process cooperation. We will adopt a simple, didactic concurrent language based on the ***local environment***, or ***message-passing***, cooperation model. This choice is not limiting, because of the proved duality of message-passing and shared-variable models. Moreover, the didactic nature of the language is just for clarity and for simplification of concept presentation and understanding. Such a language is defined according to the general semantics of Hoare's Communicating Sequential Processes (CSP), which is adopted by any existing message-passing library, e.g. MPI.

Let us call ***LC*** this concurrent language ("***L****inguaggio* ***C****oncorrente*").

6.1 Concurrent language definition

6.1.1 Structure of parallel programs

A LC parallel program is a collection of processes, declared as:

```
parallel  < list of unique process names >; < possible declarations of parametric names>;
<possible declarations of parametric data types>
< process definition >;
…
< process definition >;
```

The *parallel* command cannot be used inside process definitions, i.e. processes cannot be nested each other.

A *unique process name* can be any character string. The process definition has a simple structure:

```
< unique process name> ::
< declarations >
< code >
```

Processes can be defined *parametrically* using free variables. For example, a unidimensional *process array* is expressed as:

```
parallel PROC[N]; int A[N];
PROC [j] :: … int A[j];….; A[j] = F (…, A[j], …); …
```

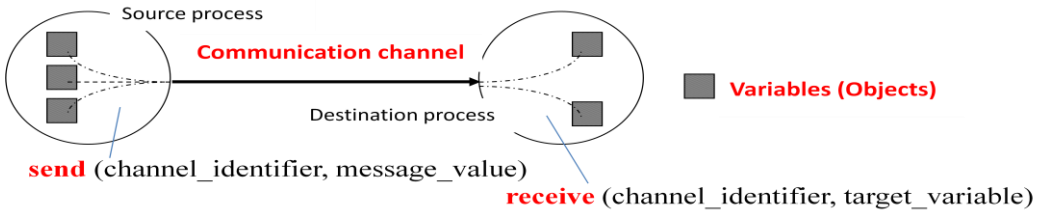
In this case the A[N] components are partitioned among processes PROC[0], …, PROC[N–1], and each of them declares and uses, as a local variable, the array element corresponding to its own index.

Process arrays are a powerful, yet simple, mechanism to express computation that are *replications* of the same code, possibly with a parametric partitioning of data types: a typical property of structured parallel paradigms.

The sequential part of LC is a typical imperative C-like language. In the following we will use an algorithmic pseudo-language with assignment (=), control constructs *if-then-else*, *case*, *for*, *while*, command sequences enclosed in brackets {…}, procedures and functions, as well as typical static data types.

6.1.2 Typed communication channels

In LC semantics, communication channels have a *type*: it is the type of the *messages*, which are sent over the channels, and the type of the *target variables*, to which received messages are assigned:



Channel types are recognized and checked *at compile time*. Messages cannot contain pointers. A channel has a *unique name*, expressed as a character string.

The LC commands for operating on communication channels are the *communication mechanisms* we wish to study: ***send*** and ***receive*** commands are the basic message-passing primitives, whose syntax is shown in the figure. Two *corresponding send* and *receive* commands refer the same channel.

Messages and target variables can be espressed in the form of *tuples*. For example *(operation, value1, value2, index)* is a legal message or target variable.

Communication channels are declared by the process that use them, distinguishing between *input channels* and *output channels*. For example:

```
parallel A, B;
A ::    … channel in go, ch2; channel out compute; …
        { … send (compute, …); …;  receive (go, …); …; receive (ch2, …); … }
B ::    … channel in compute, ch3; channel out go, ch4, ch5; …
        { send (go, …); …; receive (compute, …); … }
```

In this example, channel names are *constant*. Alternatively, in LC a channel can be identified by a *variable*: the language types include the ***channlename*** type, declared using the **var** keyword. For example:

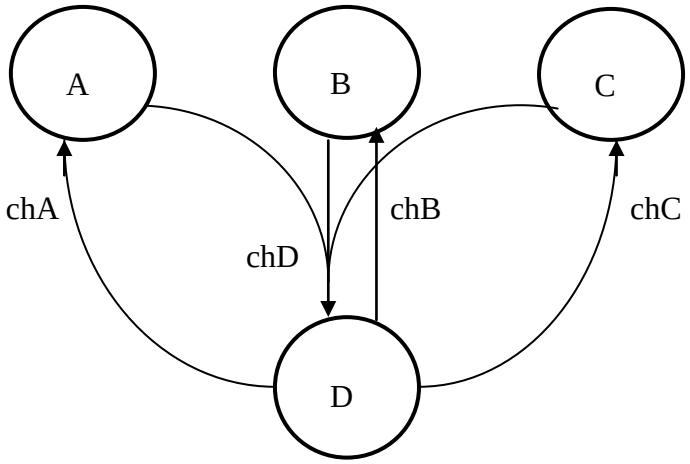
```
channel in ch1, var ch2; channel out var ch3, var ch4, ch5…
```

The possible values that can be assumed by a channelname variable are determined at compile-time through a static analysis of the parallel program.

Assignment operations are defined on channelname variables. This allows the programmer to manage and control the communication channels parametrically, notably *using channel names in messages and in target variables*. For example:

```
parallel A, B, C, D;
A :: … channel in chA; channel out chD; …
        {… send (chD, (chA, valore)); …; receive (chA, variable); … }
B :: … channel in chB; channel out chD; …
        {… send (chD, (chB, valore)); …; receive (chB, variable); … }
C :: … channel in chC; channel out chD; …
        {… send (chD, (chC, valore)); …; receive (chC, variable); … }
D :: … channel in chD; channel out var ch_out; …
        {… receive (chD, (ch_out, value)); …; send (ch_out, variable); … }
```

corresponding to the following *computation graph*:



In this example, processes A, B, C act as "clients" of a "server" process D. Each client sends onto *chD* channel a message (the service "request") including the name of the channel from which the client wishes to receive a message from the server (the service "reply"). The *ch_out* variable of D is a target variable that is assigned the value *chA*, or *chB*, or *chC*. In this example *ch_out* is used in a *send* command to parametrically identify an output channel.

The following program is another example about the utilization of channelname variables and structured channels to parametrically select input and output channels:

```
parallel CLIENT[N], SERVER; channel request [N];
CLIENT[j] :: … channel in ch_result, …; channel out request[j], …;
        { … send (request[j], (ch_result, x)); …; receive (ch_result, y); …}
SERVER:: … channel in request [N], …; channel out var ch_out; …
        { … for (j = 0; j < N; j++)
            { receive (request [j], (ch_out, x)); y = F(x); send (ch_out, y) }; …}
```

6.1.3 Communication forms

The communication forms have been defined in Section 2.4 and applied to the firmware level. Let's review their definition and apply them to LC:

1) *symmetric* or *input asymmetric*, which are not distinguished by specific declarations. The *channelname* mechanism expresses a form of symmetric or asymmetric parametric communication;

2) *synchronous* or *asynchronous* The ***asynchrony degree*** *k* of a channel is a non-negative integer constant $k \geq 0$, where the case $k = 0$ corresponds to synchronous communication. The asynchrony degree of a channel is associated to the channel declaration in the destination process only; if $k = 0$ it can be omitted. For example:

```
channel in ch1 (4), ch2 (1), ch3, ch4(0); channel out …
```

Channels *ch1*, *ch2*, *ch3*, *ch4* have asynchrony degree equal to 4, 1, 0, 0 respectively.

In an *asymmetric* channel with asynchrony degree *k*, *every* sender process has an asynchrony degree *k* with respect to the destination process.

6.1.4 Non-determinism control in communications

Alternative guarded commands (ECSP concurrent language, University of Pisa) are used for this fundamental task in message-passing computations.

An alternative command expresses the possibility to receive a message from *any* channel belonging to a given *channel set*, which in general can be variable and controllable by program. Moreover, a priority mechanism can be applied when more than one channel belonging to the specified set are ready for communication.

The syntax is:

```
alternative
{    priority (pr_1), pred_1 (…), receive (…)   do { … command_list_1 …}
     or priority (pr_2), pred_2 (…), receive (…)   do { … command_list_2 …}
…
     or priority (pr_n), pred_n (…), receive (…)   do { … command_list_n …}
}
```

A ***guard*** as

```
priority (pr_i), pred_i (…), receive (…)
```

contains a *priority* (value of an integer variable), a *local guard* consisting in the evaluation of a predicate on the process internal state, and a *global guard* expressed by a *receive* command. *One or more of these three elements may be absent*.

A guard is said to be

- ***verified*** if *pred_i* is true and the *receive* command can be executed;
- ***suspended*** if *pred_i* is true, but the *receive* command cannot be executed;
- ***failed*** if *pred_i* is false.

If one or more guard are verified, the guard having the highest priority is selected, the *receive* command is executed, then the corresponding command list is executed, and the command terminates. If more than one verified guard have the same priority, one of them is selected according to a *random* strategy which is invisible to the programmer (it is implemented in the run-time support).

If *all* guards are suspended, the process is suspended in the alternative command. It will be waked up as soon as one or more guard will become verified.

If *all* guards fail, the alternative command terminates without executing any *receive* command or command lists.

The so called *repetitive command* is not primitive in LC, however it is emulated by an alternative command inside a loop. For example:

```
parallel A, B, C, DEMON;
A ::      channel in chA (1); channel out chD_A; …
          {… send (chD_A, (chA, value)); …; receive (chA, variable); … }
B ::      channel in chB (1); channel out chD_B; …
          {… send (chD_B, (chB, value)); …; receive (chB, variable); … }
C ::      channel in chC (1); channel out chD_C; …
          {… send (chD_C, (chC, value)); …; receive (chC, variable); … }
```

```
DEMON ::  channel in chD_A (1), chD_B (1), chD_C (1); channel out var ch_out; int pr[3]; T x, y ;
        { for (i = 0; i < 3; i++) do pr[i] = i;
          while (true) do
                alternative
                        { priority (pr[1]), receive (chD_A, (ch_out, x) do
                                { y = F(x); send (ch_out, y); for (i = 0; i < 3; i++) do pr[i] = (pr[i]) mod 3 + 1}
                          or priority (pr[2]), receive (chD_B, (ch_out, x) do
                                { y = F(x); send (ch_out, y); for (i = 0; i < 3; i++) do pr[i] = (pr[i]) mod 3 + 1 }
                          or priority (pr[3]), receive (chD_C, (ch_out, x) do { y = F(x), send (ch_out, y) }
                                { y = F(x); send (ch_out, y); for (i = 0; i < 3; i++) do pr[i] = (pr[i]) mod 3 + 1 }
                        }
        }
```

The classical example of buffer management is shown in the following. The buffer object and the management policy are implemented by proper data structures, by the functions *empty_buffer*, *full_buffer*, *get* and by the procedure *put*.

The example is meaningful since it illustrates the power of non-determinism control through the mechanism of local + global guards.

The first version is an infinite cycle (demon process):

Version 1: demon process

```
BUFFER_MANAGER::
channel in ch_prod (1), ch_cons (1); channel out var ch_out; item x; …
        { while true do
                alternative
                        { not full_buffer ( ), receive (ch_prod, x)   do put (x)
                          or not empty_buffer ( ), receive (ch_cons, (ch_out))   do
                                {x = get ( ); send (ch_out, x)}
                        }
        }
```

In the second version, the process *termination* can be caused by information received by other partner processes. The command list *TERM* is the termination handler:

Version 2: termination handling

```
BUFFER_MANAGER::
channel in ch_prod (1), ch_cons (1); channel out var ch_out; item x; boolean end; …
        { end = false;
          while not end do
                alternative
                        { not full_buffer ( ), receive (ch_prod, (end,x)  do { put (x); if end then TERM }
                          or not empty_buffer ( ), receive (ch_cons, (ch_out))  do
                                {x = get ( ); send (ch_out, x)} }
```

6.2 Interprocess communication run-time support

In order to build the executable version of a parallel program, LC compiler utilizes a set of *run-time libraries* belonging to the interprocess communication run-time support. We will study the run-time support of basic *send* and *receive* commands, which are implemented as *procedures* with the following signatures:

send (ch_id, msg_addr)

receive (ch_id, vtg_addr)

where parameter *ch_id* is a unique constant channel identifier, and parameters *msg_addr*, *vtg_addr* are, respectively, the logical base address of the message data structure in the sender logical address spaces and the logical base address of the target variable data structure in the receiver logical address spaces.

Run-time support procedures are executed in an indivisible way: for a uniprocessor architecture, indivisibility is implemented by *disabling interrupts*, for a multiprocessor architecture by disabling interrupts *and* locking mechanisms (Part 2).

We consider the case of ***symmetric, synchronous or asynchronous, deterministic channels*** (i.e. not referred in alternative commands), for ***uniprocessor*** architectures. In Part 2 the run-time support for multiprocessors and multicomputers will be derived by proper modifications to the uniprocessor run.time support.

6.2.1 Process support and shared data structures

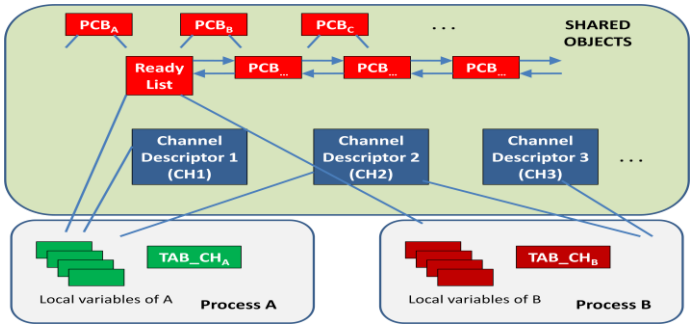
The run-time support operates on proper data structures, which can be private of, or shared by, the sender and receiver processes.

The basic data structure is the ***channel descriptor*** (CH), *shared* by the sender and the receiver process. We will study several implementations of *send-receive*, for each of which a different CH structure will be provided. The compiler has several *versions* of run-time support libraries and, for each channel, selects one of them in order to introduce optimizations depending on the application and/or on the underlying architecture.

The *Channel Table* is a private data structure (TAB_CH), used to obtain the CH logical address as a function of the channel identifier *ch_id*.

As usually, each process has a *Process Descriptor* (PCB), shared with all the other processes, containing utility information (internal state, pointer to the Ready List, to the Channel Table, to the Relocation Table, and so on).

The following figure reviews the concept of shared data structures at the run-time support level:



The reader is invited to resume the definition of *shared objects* (Section 4 and 4.4. in particular).

6.2.2 "Generic" implementation

The first version we study is valid both for synchronous and for asynchronous channels. For this reason, with this version we are not able to introduce optimizations, which instead will be introduced in the other versions.

The channel descriptor is a data structure containing the following fields:

- *sender_wait*, *receiver_wait:* boolean variables, denoting the possible waiting condition of the sender or of the receiver process;
- *message length, L,* expressed as number of words. This number corresponds to information size for that channel type.
- *buffer*: a FIFO queue of $N = k+1$ positions, each one of L words. The queue is implemented as a circular vector with insertion and extraction pointers (indexes). The sender process inserts the message into the queue and, if the queue becomes full, then it passes into the waiting state. The solution with $N = k+1$ avoids that the sender process re-executes the *send* primitive once waked up and resumed into execution; instead, it is resumed at the *return* logical address of the *send* procedure. In this first version, the receiver, when the empty buffer condition is found, is suspended at the logical address of the *receive* procedure itself (i.e., the receive procedure is re-executed when the receiver execution is resumed);
- *sender_PCB_ref*, *receiver_PCB_ref:* references to sender PCB and receiver PCB. The implementation of such references (logical addresses, or unique identifiers, or physical addresses, or capabilities), as in any other case of indirectly referred shared data structures ("shared pointers"), will be studied in Section 3.4.2.

The pseudo-code of *send* and *receive* procedures is the following:

```
send (ch_id, msg_address) ::
        CH address =TAB_CH (ch_id);
        put message value into CH_buffer (msg_address);
        if receiver_wait then { receiver_wait = false;
                                wake_up partner process (receiver_PCB_ref) };
        if buffer_full then { sender_wait = true;
                              process transition into WAIT state: context switching }

receive (ch_id, vtg_address) ::
        CH address =TAB_CH (ch_id);
        if buffer_empty then { receiver_wait = true;
                               process transition into WAIT state: context switching }
          else get message value from CH buffer and assign it to target variable (vtg_address);
         if sender_wait then { sender_wait = false;
                               wake_up partner process (sender_PCB_ref) }
```

It is worth noting that the interprocess communication run-time support does not consist merely in message buffering and copying ("*real communication*"): it contains also *synchronization* operations and *low-level scheduling* operations.

6.2.3 Direct copy into the target variable in asynchronous communications with suspended receiver

The initial version can be optimized in several respects.

First of all, it is complicated by the fact that the same "generic" algorithms covers both synchronous and asynchronous communication. In the asynchronous communication at most one of the partner process can be in waiting state, while in the synchronous case a waked up process can find the partner itself in the waiting state.

From now on, we will distinguish distinct libreries, the one for *synchronous* and the other for the *asynchronous* communication.

Let us consider the asynchronous communication ($k \geq 1$).

The most evident source of inefficiency is when the *send* procedure finds the receiver process in the waiting state. In this case, the *double copy* of message – first into the buffer queue (during *send*) then into the target variable (during *receive*) – is unnecessarily time-consuming, especially for long messages. *In this situation* (suspended receiver), the optimization is simply the following: *the send procedure provides to copy directly the message into the target variable*, i.e. the *send* procedure performs also the *receive* task. Moreover, the receiver is suspended at the *return* address of *receive* procedure (*receive* is not re-executed)..

Notice that target variable becomes a *shared* data structure for the sender and the receiver process.

The channel descriptor has now the following structure:

- *wait:* boolean variable, assuming the true value if one of the two partners is suspended;
- *message length*, *L*, expressed as number of words.
- *buffer*: FIFO queue, as in the initial implementation;
- *vtg_ref*: reference to the target variable (indirectly referred shared data structure): written dynamically by the receiver when it is suspended;
- *PCB_ref:* reference to the PCB of the waiting process: written dynamically.

The pseudo-codes are the following:

send (ch_id, msg_addr)

```
CH_address = TAB_CH (ch_id);
if wait then { wait = false;
    copy message into the target variable (msg_addr, vtg_ref);
    partner wake-up (PCB_ref) }
        else { put message value into CH_buffer (msg_addr);
       if buffer_full then { wait = true;
     copy sender PCB_ref into CH;
     process transition into the waiting state}
}
```

receive (ch_id, vtg_addr)

```
CH_address = TAB_CH (ch_id);
if buffer_empty then { wait = true;
    copy receiver PCB_ref and vtg_ref into CH;
```

```
        process transition into the waiting state}
    else { get message value from CH_buffer and assign it to target variable (vtg_addr);
                if wait then { wait = false;
        partner wake-up (PCB_ref) }
    }
```

6.2.4 Synchronous communication: "peer" implementation

The previous solution is generalized to the synchronous communication, leading to a very efficient and elegant implementation. Now, the *send* and *receive* behaviour are *dual*: as soon as one of the two partners tries to execute the respective command, it is suspended; subsequently the other process copies directly the message into the target variable and awakes the suspended partner:

Now both the message and the target variable are *shared* data structures.

The channel descriptor does not contain any buffer - only synchronization and low level scheduling information - :

- *wait:* as in the previous solution;
- *message length:* as in the previous solution;
    - *val_ref:* reference to the message or to the target variable (indirectly referred shared data structures): written dynamically by the sender or by the receiver, respectively, when it is suspended;
- *PCB_ref:* as in the previous solution.

The dual algorithms for the synchronous *send* and *receive* run-time support are:

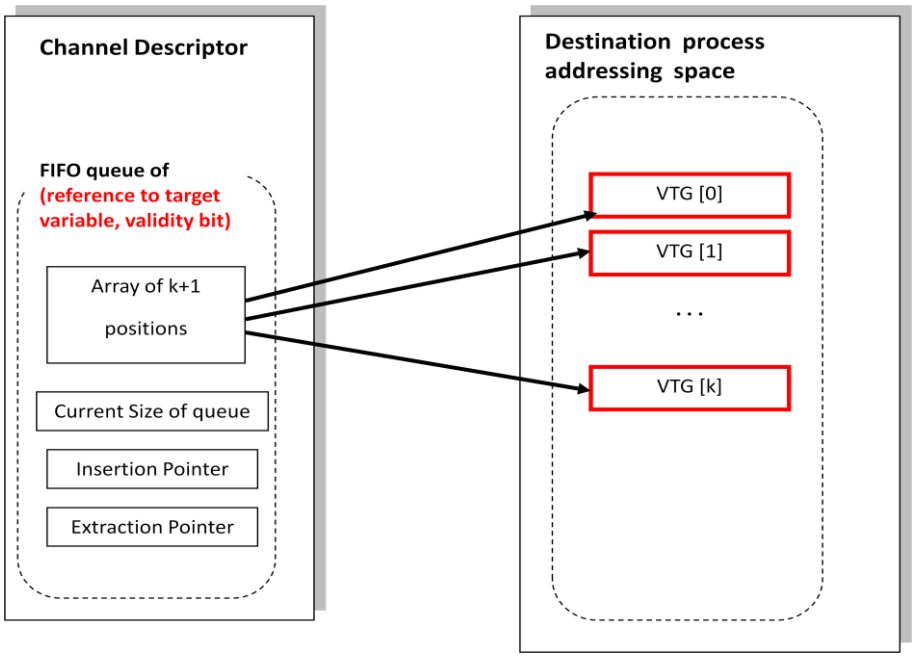
```
send (ch_id, msg_addr) ::
        CH_address = TAB_CH (ch_id);
        if wait then {
                wait = false;
                        Copy message into target variable (msg_address , VAL_REF);
                        partner wake-up ( PCB_REF) }
                else { wait = true;
                        copy message_reference and PCB_reference into VAL_REF and PCB_REF;
                        process transition into WAIT state: context switching }

receive (ch_id, vtg_addr) ::
        CH_address = TAB_CH (ch_id);
        if wait then {
                wait = false;
                        Copy message into target variable (vtg_address , VAL_REF);
                        partner wake-up ( PCB_REF) }
                else { wait = true;
                        copy vtg_reference and PCB_reference into VAL_REF and PCB_REF;
                        process transition into WAIT state: context switching }
```

6.2.5 "Zero-copy" communication

The optimizations studied in the previous Sections can be applied, in the most general case, to any asynchronous communication channel, as demonstrated by some advanced research libraries (VIA, Fast Messages): now we are able to reduce the number of message copies to just one, i.e. to the minimum, *independently of the receiver progress state.*

The basic principle for "zero copy" communication is shown in the following figure:



Multiple copies of every target variable are provided. That is, each time the receiver process refers the target variable VTG, in fact it refers to a different *instance* of VTG according to a circular ordering. The programmer or the compiler is able to structure the process according to this principle: in some cases, it is equivalent to realize a *k+1* loop-unrolling in the receiver process.

A FIFO queue of k+1 target variable instances is defined in the receiver process address space, and they are *shared* (statically or dynamically) with the sender process. Consequently, the channel descriptor contains, besides the information for synchronization and low level scheduling, a FIFO queue of *references* to the *k+1* target variables:

- *wait:* as in the previous solution;
- *message length:* as in the previous solution;
  - buffer: FIFO queue of tuples (references to target variable, validity bit);
  - *PCB_ref:* as in the previous solution.

Notice that, by definition of this method, the receiver process works directly on the target variable instances (without copying them, of course). For this reason, a *validity bit* is associated to every target variable instance, in order to grant the mutual exclusion between sender and receiver. In fact, a critical race could occur when the sender tries to copy a message into a target variable instance and the receiver *is still utilizing it*:

- the validity bit is reset (= 0) in the *receive* command and is set again (= 1) when the receiver process is no more willing to utilize the target variable instance. Thus a **set_validity_bit** primitive must be provided in the concurrent language;
- the sender is suspended if the validity bit of the target variable referred by the CH buffer is equal to 0.

The pseudo-code of zero-copy *send* run-time support is the following:

```
send (ch_id, msg_address) ::
        CH address =TAB_CH (ch_id);
        if (CH_Buffer [Insertion_Pointer].validity_bit = 0)  then
                        { wait = true;
                          copy reference to Sender_PCB into CH.PCB_ref
                          process transition into WAIT state: context switching };
        copy message value into the target variable referred by
                                CH_Buffer [Insertion_Pointer].reference_to_target_variable ;
        modify CH_Buffer. Insertion_Pointer and CH_Buffer_Current_Size;
        if wait then
                        { wait = false;
                          wake_up partner process (CH.PCB_ref) };
        if buffer_full then
                        { wait = true;
                          copy reference to Sender_PCB into CH.PCB_ref
                          process transition into WAIT state: context switching }
```

Notice that the validity bit can be eliminated if the following constraint is imposed: after each receive only the received target variable instance is used (and not any other instance).

Otherwise, the zero-copy implementation reduces the communication latency *at the expense of the validity bit overhead*, i.e. while the validity bit manipulation has a negligible effect, possible additional context-switchings can occur in the sender process. This effect can be minimized by adopting some proper strategies:

a) a general strategy is *to increase the asynchrony degree* in order to reduce the probability of finding the validity bit equal to 0;

b) moreover, there are *some parallel program structures that are able to minimize (to eliminate) this problem* owing to their implicit behavior. For example, consider a farm structured parallel program: a scheduler process distributes the input stream tasks to a collection of executor processes according to an "on demand" load balancing strategy, i.e. an executor explicitly signals the scheduler its availability to receive a new task. Thus, it is impossible that the scheduler tries to write into a target variable instance while the executor is still working on it.

Exercise

Write an equivalent representation of the following benchmark, using zero-copy communication on a *k*-asynchronous channel:

```
Receiver ::
        int A[N]; int v; channel in ch (k);
        for (i = 0; i < N; i++)
                        { receive (ch, v);
                          A[i] = A[i] + v }
```

The answer has to include the definition of the *receive* run-time support (procedure), and the receiver code before and after the *receive* procedure for a correct utilization of zero-copy communication.

6.3 Communication latency

The interprocess communication latency, $\mathbf{L_{com}}$, is the mean time needed to execute a complete communication, that is the mean time between the beginning of the *send* execution and the copy of the message into the target variable, including synchronization and low level scheduling operations.

Let us consider the implementation of a *send* command. Its latency, for a message of length *L*, can be expressed by the following formula:

$$T_{send} = T_{setup} + L * T_{transm}$$

where

- $\mathbf{T_{setup}}$ is the average latency of all the actions that are independent of the message length: synchronization, manipulation of buffer indexes, low level scheduling;
- $\mathbf{T_{transm}}$ is the average latency for a single iteration of the message copy loop, i.e. the latency to copy one word of the message.

A similar formula can be written for the *receive* latency in any implementation, except the zero-copy ones. We can assume that the T_{setup} and T_{transm} values are comparable, and very similar indeed, for *send* and *receive*. Thus, we can approximately write:

$$L_{com} = 2\ T_{send} = 2\ (T_{setup} + L * T_{transm})$$

except for cases in which the message is copied directly into the target variable.

This latency is reduced to about T_{send} in the *zero-copy* run-time support. In fact, consider that the *receive* latency is negligible compared to the *send* latency, because the receive run-time support does not perform message copies. This approximation is as more valid as long the message is.

In the following, we assume that the *zero-copy communication latency* is given by the *send* latency only:

$$L_{com} = T_{send} = T_{setup} + L * T_{transm}$$

Typical values for uniprocessor architectures have the following orders of magnitudes:

$$T_{setup} = (10^2 - 10^3)\ \tau \qquad T_{transm} = (10^1 - 10^2)\ \tau$$

For parallel architectures we must add one or more orders of magnitude, according to the characteristics of memory hierarchy, interconnection network, and so on (this analysis will represent a key issue in Part 2).

6.4 Design issues for interprocess communication run-time support

All described versions of *send-receive* run-time libraries have been designed according to a basic principle: they are executed in *user space*, i.e. no privileged hierarchical states are exploited. Consider the typical implementation of a primitive in supervisor (or kernel) state: by its nature, a supervisor call must perform the parameter passing by values. This implies several additional copies of the message and other parameters, thus no optimization can be pursued.

On the other hand, if an implementation in supervisor space is adopted, this means that the run-time support is implemented on top of an operating system (or similar virtual machine), thus interposing a set of interpretation mechanisms that, in general, have been defined for quite different purposes.

In the HPC world, the most efficient run-time libraries of industrial quality are implemented in user space.

Another important design issue concerns the clear and efficient implementation of the so-called problem of indirectly referred shared data structures, or *shared pointers problem.* See Section 4.5.