EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Description and implementation in hardware of digital systems

van Dort, E.J.

Award date:
1988

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

EU 13

Eindhoven University of Technology
Department of Electrical Engineering
Digital Systems Group

DESCRIPTION AND IMPLEMENTATION IN HARDWARE OF DIGITAL SYSTEMS

by E.J. van Dort

Master Thesis by E.J. van Dort

Coached by Prof. ir. M.P.J. Stevens

Eindhoven, The Netherlands

August 1988

The department of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of this report.

TEAMWORK

There were four people named Everybody, Somebody, Anybody, and Nobody. An important job had to be done, and Everybody was asked to do it. Everybody was sure that Somebody would do it. Anybody could have done it, but Nobody did it. Somebody got angry about that, because it was Everybody's job. Everybody thought Anybody could do it, but Nobody realized that Everybody couldn't do it. It ended up that Everybody blamed Somebody when Nobody did what Anybody could have done.

ABSTRACT

Designing complex digital systems (hardware) in a structured way is a problem that can be solved by using the 'Top-Down' approach. This approach starts with an informal (english) specification of the system. The specification must then be transformed to a formal description. A Pascal-like description language can be used for this. Basic synchronization and communication statements are developed to describe synchronization and communication in hardware for procedure calls, mutual exclusion, etc.. Describing the system formally is a process with stepwise refinement. his description process stops when all operations in the system description can be realized in hardware. After describing the system in all of its detail, the description has to be transformed to a set of synchronized digital circuits. A general system architecture, which can be used for transforming the system description to digital circuits, is presented.

During all phases of the design process, verification and performance analysis must be done to check if the system fulfills its specifications. This can be done by simulating the system description at all levels, and calculating the synchronization and communication behaviour if the system contains parallelism.

ACKNOWLEDGEMENTS

At this place I would like to thank

- my parents. They have contributed to my future in both immaterial and material ways, and they did it all with love.
- prof. ir. M.P.J. Stevens. It was very pleasant cooperating with him. His contribution to my work was significant. I hope to stay in contact with him after leaving the university.
- those members of the Digital Systems Group with who I had very interesting and fruitful discussions.

TABLE OF CONTENTS

INTRODUCTION 1

1. DIGITAL SYSTEMS IN THEIR ENVIRONMENT 3
1.1. General digital system behaviour 3
1.2. Decomposition of digital systems 4
1.3. Conclusion 5

2. HIGH LEVEL PROGRAMMING LANGUAGES 6
2.1. Programming language analyses 6
2.2. Information transformation statements 8
2.3. Flow control statements 8
2.4. Procedures and functions 9
2.5. Synchronization and communication 12

3. DIGITAL SYSTEM DESIGN METHODOLOGY 15
3.1. Informal system specification 15
3.2. Formal system description: functional decomposition . . . 16
3.2.1. Behaviour description 18
3.2.2. Synchronization and communication 20
3.2.2.1. Petri Net Theory 24
3.2.2.2. Interpretation of Petri net graphs 29
3.2.3. Examples 33
3.2.3.1. Conditional synchronization 33
3.2.3.2. Competition 34
3.2.3.3. Cooperation by sharing 37
3.2.3.4. Cooperation by communication 37
3.3. Formal system description: structural (de)composition . . . 37
3.3.1. Implementation of processes 38
3.3.1.1. Digital circuit implementation 38
3.3.1.2. Macro implementation 40
3.3.2. Data transport between processes 41
3.4. Conclusion 43

4. A GENERAL SYSTEM ARCHITECTURE 44
4.1. System architecture decomposition 44
4.2. Process implementation 45
4.2.1. The processing unit 46
4.2.1.1. Storage elements 47
4.2.1.2. Primitive operations and status 47
4.2.1.3. State machine representation 49
4.2.1.4. Example 50
4.2.2. The control unit 52
4.3. Event management implementation 53
4.4. Conclusions 60

CONCLUSIONS AND FURTHER RESEARCH 61

REFERENCES 62

APPENDIX A: Sequential Circuit Configurations 63

APPENDIX B: Process Configurations 69

APPENDIX C: Event Management Configurations 102

INTRODUCTION

During the last decade the complexity of digital systems has increased tremendously. This also goes for integrated circuits, due to the increasing number of gates (transistors) which can be integrated on a chip. For this reason the design of digital systems, or ICs, is becoming an increasing problem. The existence of a structural design method will be necessary to control over the design process. Today's trend is problem decomposition at the several stages during the design process. During the specification and description phase of the system one can decompose to different **functions** of the system. This is called **functional decomposition**. At implementation level there is decomposition to different **parts** of the system. The **structure** of the system must be defined, so this is called **structural (de)composition**. These two kinds of decomposition can result in the same decomposition choices, but they don't have to. Although, the functional decomposition always causes important consequences for the final implementation.

Functional decomposition cannot be done by a Silicon Compiler, in these days a very popular conception. The digital system design process must be seen as a tree. The root of this tree is the system specification and there are a lot of leaves which represent possible implementations of this system. These implementations are all correct but there is only one optimal design. This optimum depends on the criteria which are coupled with the system specification. When a designer starts at the root, he has to make decisions (on functional decomposition) at every node in the tree. If he is able to evaluate the consequences of several possibilities, he can choose the path that most likely leads to the optimal design. If the designer can be guided through this 'design process tree' by very powerful computer tools (consequence analyzers), then the system design process can be made faster and lead to better designs. Silicon Compilers can be used in the final step of the design process: transformation of exactly defined hardware behaviours into digital circuits.

An interesting comparison can be made between the design of systems in hardware and software. In fact, software is a special case of hardware. When designing software, one (the designer and compiler) always has to have the architecture of the computer system in mind, on which the software has to be executed. Designing software is describing the behaviour of a digital system (the computer system). Designing software is designing a high level controller which controls low level controllers and processing units (the computer system). Since a software implementation is a special case of a hardware implementation, any digital system can be realized both in software and in hardware. So the application fields of hardware and software are equal. The only criteria for choosing between hardware and software realizations of a particular system are the costs/performance trade off (criterium for consumer) and the difference in system development equipments (criterium for producer). Because of the

close relationship between hardware and software design, it would be very interesting to be able to use the large experience of software design methods, like high level (concurrent) programming languages, for designing hardware. Of course there is one big difference between hardware and software implementations. The architecture of a computer system is fixed, so in software design only **functional** decomposition has to be dealt with. On the other hand for designing hardware there is no architecture; there is nothing but the designer him/herself. The goal is to design the architecture, so both **functional** and **structural** decomposition has to be dealt with.

This report will handle the transformation of a digital system, described in a high level concurrent programming language (Pascal like), to elementary hardware building blocks. The problems of the algorithmic description, definition of primitive operators, realization of high-level operators, shared hardware, communication and synchronization, architecture performances, etc. will be discussed.

To describe a design methodology it is necessary to analyze the general properties (behaviour) of digital systems with respect to their environment. This will be the subject of chapter 1. In chapter 2 the characteristics of high level programming languages and software development methodologies will be described. This will be used to design digital systems. A methodology to describe and implement digital systems will be presented in chapter 3. Finally, in chapter 4, a concept of system architectures will be discussed which can be used to implement high level system descriptions.

1. DIGITAL SYSTEMS IN THEIR ENVIRONMENT

To be able to present a structured methodology for designing digital systems one first needs to know what digital systems are. The properties and behaviour of a general digital system, considered as a black box, must be analyzed.

A digital system always is a part of a larger system. It never operates isolated, it always communicates with a particular environment. For example, communication with human beings can be performed by using keyboards and displays. In general, sensors and activators are the interface between a digital system and its environment. This chapter will describe the behaviour of digital systems with respect to its environment.

1.1. General digital system behaviour

A digital system can show two different kinds of behaviour with respect to its environment.

1) The system performs particular tasks on request of its environment. This kind of behaviour will be called the **slave** behaviour of a system.

2) The system requests its environment to perform particular tasks. This will be called the **master** behaviour of a system.

In general a digital system shows both master and slave behaviour, so it is involved in an interactive process with its environment. For example, a human being can enter a command on the keyboard of a Personal Computer to start the execution of a computer program. In this case the human being is the master, and the Personal Computer shows slave behaviour. During execution of this program, the Personal Computer might ask the human being to insert a particular floppy disk in the disk drive. Now the Personal Computer shows master behaviour and the human being is the slave.

Another keyword in the behaviour of digital systems is **task**. As mentioned before a system performs particular tasks. A task is a compilation of different activities:

- Receiving information from its environment.

- Transmitting information to its environment.

- Transforming information.

- Requesting its environment for performing particular tasks.
- Waiting until its environment has performed particular tasks.

Only information transformation is an internal activity of the digital system. The other four activities are interactions between the system and its environment. It is clear that these activities have to be **synchronized**. To continue the example of the Personal Computer and the human being; when the human being has inserted the floppy disk then he or she has to tell the Personal Computer that the task is fulfilled, for example by pressing a key.

In figure 1 a schematic representation of a digital system is shown, in relation to its environment.

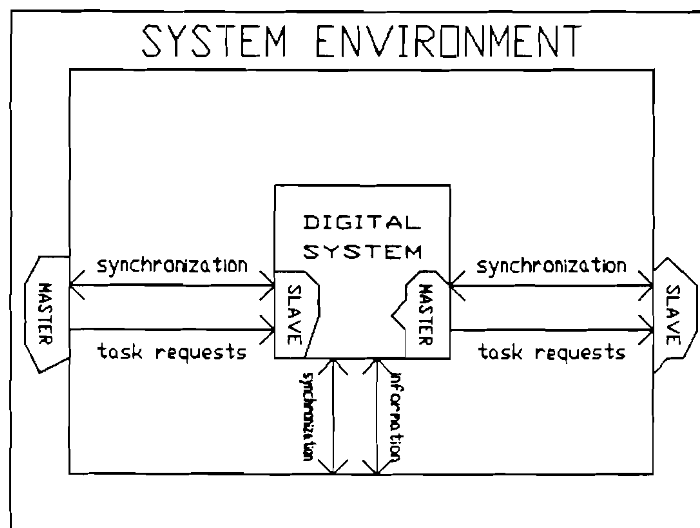


Figure 1. A digital system and its environment

1.2. Decomposition of digital systems

As described before in the introduction, a digital system can be a composition of several subsystems as a result of functional and/or structural decomposition. These subsystems are interconnected, they communicate with each other, and they have to be synchronized. In fact, a subsystem behaves exactly in the same way as a complete system, it is just one level deeper in the hierarchy of a system and its environment. A schematic representation of a subsystem in its environment looks exactly the same as that of a complete system. It is shown in figure 2.

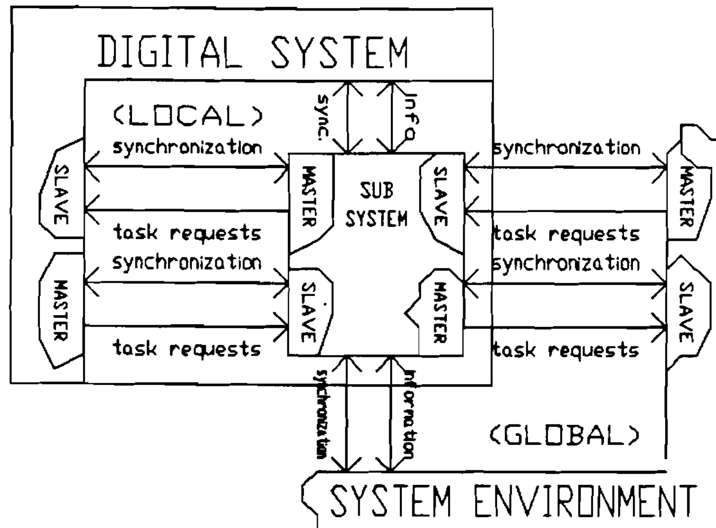


Figure 2. A subsystem placed in its local and global environment

1.3. Conclusion

A digital system can be considered as a black box. See for example figure 1. It is impossible to see how this system is realized. It can be a software or hardware realization. For this reason it seems to be possible to describe the behaviour of digital systems independent of their final way of realization.

During the last decades there has been done much research to software design. This has resulted in software design methodologies and high level programming languages. It is interesting to apply this large experience in describing digital systems, to be realized in hardware, by using high level programming languages. This will be the subject of the following chapters.

2. HIGH LEVEL PROGRAMMING LANGUAGES

As described in the introduction, the goal of this report is to develop a method to transform a system, described in a high level programming language, to elementary hardware building blocks. For this reason it is necessary to analyze the general characteristics of high level programming languages and software development methodologies. This will be the subject of this chapter.

2.1. Programming language analyses

Most high level programming languages have equal basic characteristics. These basic characteristics are important now. The details in which many languages differ are not important in this context. For readability reasons a Pascal-like notation will be used in this report.

As mentioned before, software is a special case of hardware. The behaviour of a computer system depends on the software which is executed on that system. During the last decades a lot of research has been done to develop structural methods for developing software. High level programming languages are part of the result of this work.

As an example of the application of high level programming languages, consider the producer-consumer problem. A process, called the producer, sends messages to another process, the consumer. The messages produced but not yet consumed are stored in a buffer memory. There is a fixed number of buffers: n. They are used in a circular way: a buffer which contents have been consumed may be reused. Two procedures are provided for producer and consumer processes, to place messages in the buffer or withdraw them from it: **produce** and **consume**. Execution of these procedures must be synchronized in such a way that messages produced cannot be consumed more than once (the consumer cannot 'overtake' the producer), no messages can be lost through overloading a buffer (producers cannot 'lap' consumers), and messages are consumed in the order in which they are produced.

One possible solution to this problem is described in [RAYN]. The variables used, which are global for the two processes, are:

```
var buffer : array[0..n-1] of message;
    in,out : 0..n-1;
```

in and **out**, initialized to 0, show the next free and next occupied message in **buffer** respectively. Noting that the buffer is full (and

therefore one cannot produce) when **(in+1) mod n = out**, and that it is empty (so one cannot consume), when **in = out**, we obtain the following procedures:

```
procedure produce(m: message);
    begin
        repeat
            (* just wait *)
        until ((in+1) mod n <> out);
        buffer[in] := m;
        in := (in+1) mod n;
    end;


procedure consume(m: message);
    begin
        repeat
            (* just wait *)
        until (in <> out);
        m := buffer[out];
        out := (out+1) mod n;
    end;
```

These procedures can be used by other processes, for example to communicate with each other. In principle they can execute concurrently, however, mutual exclusion for the global variables must be provided in some way.

In this simple example all characteristics of high level programming languages are embedded. These languages are built on four basic principles:

1) information transformation statements (mod, addition).

2) flow-control statements (repeat .. until).

3) functional decomposition for describing processes by using procedures and functions (produce, consume).

4) Synchronization and communication mechanisms of processes, procedures and functions (mutual exclusion of buffer access).

These principles will be discussed in detail now.

2.2. Information transformation statements

This type of statements can be used for logical and arithmetic operations on variables. Examples are addition, subtraction, multiplication, sinus, exclusive-or, shift-left, assignment. Note that assignment is a special case: it only describes transportation of information. The available operations are defined by the syntax of the programming language.

Some of the statements can be executed by the hardware in the computer system. These statements are called **primitive operations**. However, a lot of statements cannot be executed by this hardware. These are the **high level operations**. Algorithms for these operations are required to transform high level operations to primitive operations. These are built in the high level programming language compiler.

It may be clear that the collection of primitive operators only depends on the hardware on which the software has to execute. The language syntax definition has nothing to do with this.

Example: In some microprocessors unsigned integer multiplication is a primitive operator. Other microprocessors have to use an algorithm for this kind of multiplication, for example repeated addition:

```
function mult(a,b : integer) : integer;
var temp : integer;
     begin
        temp := 0;
        while (b <> 0)
          do
            temp := temp + a;
            b := b-1;
          end;
        mult := temp;
     end;
```

Since a simple microprocessor has addition, subtraction and assignment as primitive operations, this algorithm can be executed by this microprocessor.

2.3. Flow control statements

Flow control statements are used to affect the program flow, depending on the present value of variables. After evaluating particular conditions, particular operations will be executed, executed repeatedly, or not executed at all. Examples are:

```
if <condition> then <statements> else <statements>

repeat <statements> until <condition>

while <condition> do <statements>

case <condition> of
     <val_1>: <statements>
     <val_2>: <statements>
     ....
     <val_n>: <statements>
```

2.4. Procedures and functions

There are several reasons for using procedures and functions in software.

1) Functional decomposition of a big problem into smaller problems makes it easier to solve the problem and the solution will be more understandable. Structured developed software is important for testing, debugging and maintenance.

2) High level operations can be defined by using procedures and functions. This is necessary when particular operations are not provided by the syntax of the language. In this way the programming language can be adapted to the software which has to be developed. In this context, the word 'operation' better can be interpreted as 'task'. For example Disk I/O can be a high level operation.

3) Introducing parallelism in software can increase the throughput, or the execution speed. For being able to execute tasks in parallel, it is necessary to develop a number of procedures which can execute concurrently.

Since procedures and functions perform particular tasks, they behave in the same way as processes or tasks, so they can be called so. The result of introducing procedures and functions is a hierarchical structure in the software. Consider for example a computer system on which a Data Link Layer receiver protocol (layer 2 of the ISO OSI model) is programmed. So the input of the computer system are received frames from the Physical Layer (layer 1 of the ISO OSI model), coded in a particular way, and the output are messages for the Network Layer (layer 3 of the ISO OSI model). In figure 3 a schematic description of this system is given.

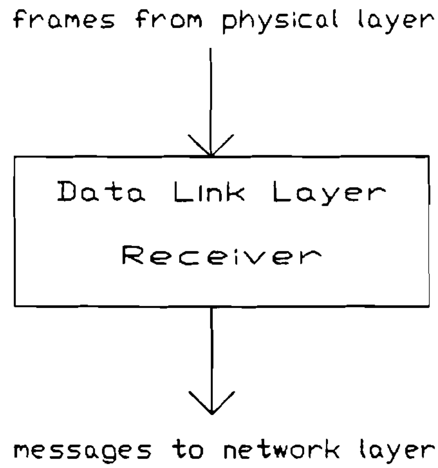


Figure 3. A Data Link Layer receiver system

When looking to the implementation of the system, one can distinguish several procedures, for example one procedure for decoding the message, and one procedure for buffering messages. This decomposition is shown in figure 4.

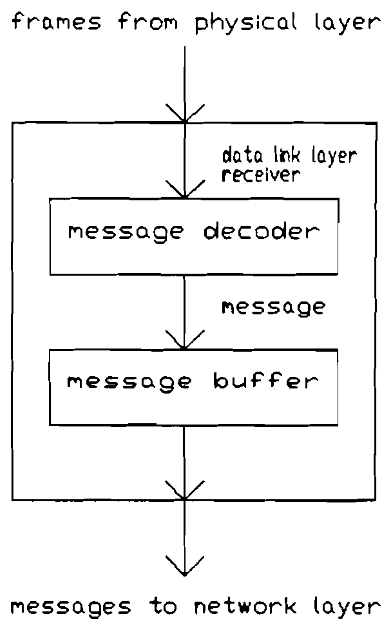


Figure 4. The decomposed receiver system

As a next step the implementation of the buffer procedure can be viewed in more detail. It can for example be implemented by using the producer and consumer procedures, as described before. Refer to figure 5 for the schematic representation.

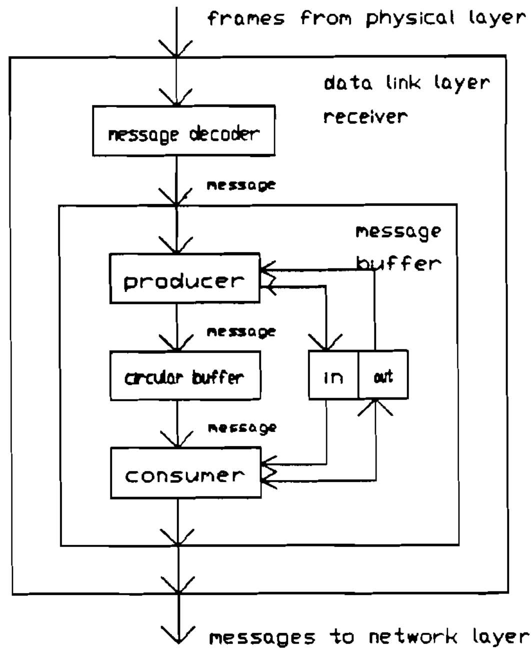


Figure 5. The detailed decomposed receiver system

Even more detail is shown when the operation **mod** is assumed to be a high level operation. Then this operation is used by the two procedures **produce** and **consume**. This is shown in figure 6.

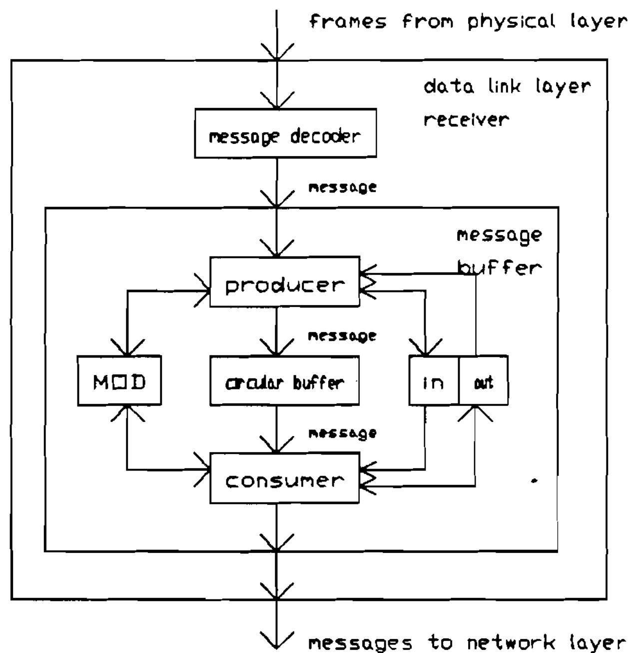


Figure 6. The receiver in more detail

This example shows that a system can be viewed at several levels. The highest level only shows the I/O behaviour and a functional description of the system. This level will be called the **specification level** of the system. At every lower level more detail of the implementation is shown. At the lowest level the complete software realization will be visible, so that level is called the **realization level.**

Due to the use of procedures and functions (functional decomposition) a hierarchical structure of systems can be defined.

2.5. Synchronization and communication

Processes have input and output of information, so they communicate with each other. Communication has to be synchronized in some way to guarantee a proper exchange of information. There are two basic methods for communication between processes:

1) Parameters can be used to exchange information. The moment of information exchange is during the call of the process, at the beginning of the execution (input parameters are read by the called process) and at the end (output is generated by the called process) of the execution of the process. In the receiver example of the previous section, the process **mod** for example can be implemented this way. When for instance the producer calls this process, then it has to pass the parameters (in+1) and n. After execution of the process **mod**, the result will be passed to the producer.

2) Global variables can be used to exchange information. When processes execute concurrently, these variables are called shared variables. In the receiver example the circular buffer is a shared variable, and so are the buffer pointers **in** and **out**.

Both methods of communication have to be synchronized, specially when concurrent execution is allowed. Four reasons for synchronization can be distinguished:

1) Mutual exclusion of shared variables. It must be prevented that shared variables can be manipulated by more than one process at the same time since this can result in incorrect results. Manipulation of shared variables must be done in non-interruptable so called **critical sections.** So other processes which want to manipulate the same variable have to wait until the critical section is completed. In the receiver example there are some shared variables. Suppose there is one bufferpointer for selecting a buffer location. This pointer can be manipulated by the two processes consume and produce. It must be

prevented that the following sequence of actions occurs (single access of the buffer is assumed):

1) the consumer passes the value **out** to the buffer pointer.

2) the producer passes the value **in** to the buffer pointer.

3) the consumer reads the selected buffer location in the message variable **m**.

4) the producer transfers its local message variable **m** to the selected buffer location.

If this sequence of actions is executed, then the consumer reads the wrong buffer location, since the producer had redefined the buffer pointer. So in this example two critical sections can be defined:

1) The consumer passes the value of **out** to the buffer pointer and reads the addressed buffer location in the local message variable **m**.

2) The producer passes the value of **in** to the buffer pointer and transfers its local message variable **m** to the addressed buffer location.

2) Synchronization of shared processes. Problems can occur when more than one process wants to execute another particular process. So mechanisms must be provided to solve this problem. For example, processes can be implemented in such a way that reentrance is allowed. Another solution is mutual exclusion of processes, so only one process at a time can be served by a shared process, and all others have to wait for their turn. In the receiver example there is the shared process **mod**, which can be called by both the producer and the consumer.

3) Communication between processes. Sometimes processes have to wait for other processes, because they need information of these processes. In the receiver example, when the consumer process has been started and the buffer is empty, then the consumer has to wait until the producer has put a message in the buffer.

4) To make sure tasks will be performed in a particular order.

A well known method of software synchronization of all four types is the semaphore mechanism, as described in [RAYN].

In this chapter the main characteristics of high level programming languages have been analyzed. In the next chapter these characteristics will be used to define a structured method to design hardware corresponding to the software design methodology.

3. DIGITAL SYSTEM DESIGN METHODOLOGY

Digital systems are very complex systems today. So there is a need for a structured design methodology. The top-down approach, in which more detail will be added at every next phase of the design process, is an often applied methodology. Exactly the same problem occurred a few decades ago when software became more complex. In that time there was a need for a structured software design methodology. This has resulted in high level programming languages and software engineering methodologies. Using this software design methodology, and adapting it, a structured hardware design methodology can be constructed.

For designing digital systems in a structured way, three important subjects can be distinguished:

1) A system description language.

2) A system design methodology.

3) A system architecture.

As discussed before, high level programming languages can be used as a hardware description language. For a structured transformation of a system description to hardware, it is necessary to define some system architecture requirements. In the next chapter a general system architecture will be discussed which can be used to implement high level system descriptions. This chapter will deal with the methodology of structured system design, using high level programming languages.

The design of a digital system requires three phases. In the first phase an informal (english) description must be developed to describe the system requirements. These include both functional behaviour and communication & synchronization requirements. In the second phase a formal description, using a high level programming language, must be developed. Finally, in the third phase, the formal description must be transformed into an implementation. Economic requirements, which are important facts in all three of the design phases, are beyond the scope of this (technical) report.

3.1. Informal system specification

A digital system that has to be designed, must be described in some way. Several characteristics must be written down.

1) Functional behaviour. The system has to perform one or more tasks. These tasks have to be described (in english) in detail to make the translation to the formal description as easy as possible.

2) Communication and synchronization with the environment. In a lot of cases, the environment of the new system already exists. This means that the way of communication and synchronization (the protocol) with this environment is (partly) fixed. This protocol has to be specified.

3) Timing requirements. As a special but very important case of 2), the timing characteristics of a system can be mentioned. In some cases the execution time for performing particular tasks, or the throughput of (manipulated) information is bounded by timing requirements. In other cases systems have to be designed to operate just as fast as possible. These timing requirements are important during the implementation and realization phases of the design process. As a remark it can be stated that systems which operate much faster than the requirements ask for, are badly designed, since they are too expensive to produce (expensive IC technology, large Silicon surface, etc.).

These three kinds of characteristics form the base for the second and third phase in the design process. Errors, ambiguities and incompleteness in this phase always cause very big problems if they are discovered too late. Then a lot of redesign has to be done to correct the system description and, in worst case, also the implementation.

3.2. Formal system description: functional decomposition

The purpose of a formal system description is to be able to transform this description to a collection of coupled digital circuits which will be the implementation of the digital system. At the end of the system description phase there will be a number of tasks, written as procedures and functions corresponding to software. In general these procedures communicate with each other and they must be synchronized. During the implementation phase every procedure and function will be transformed into a digital circuit. For this reason it is very important that the system description results in an acceptable number of procedures and functions. This can be done during two phases.

1) First there is a **functional decomposition** which results in many interacting procedures and functions. This is the same as in software development, in which it is advisable to use many

procedures and functions for developing software in a structured way. Functional decomposition will be the subject of this section.

2) Secondly there will be an additional phase only for hardware development to reduce the number of procedures and functions into an acceptable number. This is the **structural (de)composition**. The result of this work is a number of interacting procedures and functions (processes) that describe the digital system. All of these processes will be transformed into digital circuits during the implementation phase. The next section will deal with structural (de)composition.

So based on the informal description of the system characteristics, functional decomposition has to be applied, resulting in distinguishing the several tasks of the system. This section will deal with the functional decomposition process. During functional decomposition two different problems occur.

1) The formal behaviour description of tasks must be developed.

2) The communication and synchronization between tasks must be described.

The behaviour description of a system has to describe the system completely, including communication and synchronization between tasks. This implies that everything is described by the behaviour description. However, there are good reasons to make a separate description of the communication and synchronization in the system.

1) When only a behaviour description in a high level programming language is being developed, then the interactions of all processes is not clearly visible. This is because all processes are described separated, so the description of the interactions is scattered. That is hard to read.

2) When tasks will perform concurrently, there is always the danger of deadlocks. By describing the communication and synchronization of processes separated, for example by using Petri net theory [PETE], or the Calculus of Communicating Systems CCS [KOOM], one can calculate the synchronization and communication behaviour of coupled processes. In this way one can **calculate** if there are deadlocks in the system. To perform this calculation, only synchronization and communication behaviour of the tasks must be known, so no details of the internal actions are necessary.

3) By simulating only the synchronization and communication behaviour, statistical analyses can be made about how

frequently the tasks of a system will be performed. If one can estimate the execution time of tasks, potential bottle necks in the functional system description can be detected. During the implementation phase these bottle necks can be avoided by making a proper design. When no estimation of execution time can be made, upper bounds in execution time can be defined in such a way that no bottle necks appear. It is also possible to detect possible starvation. In this way restrictions in execution times of tasks and priority algorithms for task servicing can be defined, without simulating the whole functional system behaviour which may take much time. These restrictions will be important in the implementation phase.

The development of the behaviour description and the communication and synchronization of a system will now be discussed in more detail.

3.2.1. Behaviour description

Separate behaviour descriptions of all tasks must be developed. These descriptions can be made by applying the four basic principles as described in chapter 2:

1) information transformation statements.

2) flow-control statements.

3) functional decomposition of tasks by using procedures and functions.

4) synchronization and communication mechanisms.

The meaning of the fourth item will be discussed in detail later on.

The meaning of the third item is that functional decomposition is a nested process. The design of a system by applying the Top-Down approach is making a functional decomposition at all levels in the design process. In this way more detail will be added to the system description each step. The main question now is when to stop the functional decomposition process, and when to stop adding more detail to the formal system description. There can be two reasons for stopping the decomposition process:

1) The procedure or function performs a very elementary task which does not have to be decomposed. It can be implemented easily in hardware. This criterium is ambiguous. The designer has to use his knowledge and experience to decide when to stop the decomposition process for this reason.

Consider for example the multiply operation. One can decide to see this operation as a primitive operation, or one can decompose this operation and write down an algorithm for multiplication. When choosing for writing an algorithm, there will be less freedom during the system implementation phase. This can be an advantage or a disadvantage, depending on the particular design problem. In general flexibility at the start of the implementation phase is a good way of designing.

Procedures and functions (compound operations) which will not be decomposed functionally, due to this criterium, will be implemented in combinatorial logic during the implementation phase. This is so because sequential logic is always an implementation of a (sequential) algorithm. For this reason these operations will be called **primitive operations**. This is comparable to primitive operations in software, which can be executed by the hardware of a computer system.

2) The procedure or function will be performed by a (sub)system which already exists.

Consider for example a system which uses a hard disk as background memory. The designer may decide to use for example the Intel 82062 Winchester Disk Controller. This means that the functions, performed by the disk controller don't have to be described in a detailed formal way. Functions, performed by this controller chip, can be considered to be primitive operations.

Since these procedures and functions are implemented in general in sequential logic (an implementation of an algorithm), these are **high level operations**, instead of primitive operations. Due to this property there must be a synchronization mechanism to use these operations.

This last remark typically describes the main difference between primitive (combinatorial) and high level (sequential) operations in hardware. To use combinatorial logic only operators are required to perform the (primitive) operation. The result automatically appears after a particular time (the combinatorial delay) at the output of the combinatorial circuit. In sequential circuits however, it is not known when the result of the operation will be available after the beginning of execution of this operation. So at least a 'ready' signal is necessary to indicate that the result of the (high level) operation is available. This is a kind of synchronization.

In this part of the functional decomposition all tasks have to be described in detail. The manipulation of information can be described easily by using the information transformation statements, the flow-

control statements, and procedures and functions. This is just one part of a task. The others are receiving and transmitting information, and requests for task execution. For these parts a synchronization and communication mechanism must be defined. This will be discussed in the next section.

3.2.2. Synchronization and communication

There are three different kinds of interaction between processes, as described in [RAYN]:

1) **Competition**
   Several processes can make service requests on a resource (this is in general another process which shows slave behaviour) that can only serve one process at the time. Mutual exclusion on that resource must be guaranteed by some control mechanism.

2) **Cooperation by sharing**
   Processes can interact indirectly by using shared data. This data can be manipulated by several processes, but there is no explicit communication between those processes. The access to shared data is allowed to one process at the time to guarantee that the shared data remains coherent. So mutual exclusion on shared data must be guaranteed by some control mechanism.

3) **Cooperation by communication**
   Processes can exchange data explicitly. To perform communication sources and destinations of data are necessary. A mechanism has to control this process. Both sides must be ready for the data transfer(s). They must wait for each other.

In chapter 2, four types of synchronization were presented.

1) mutual exclusion of shared variables. In [RAYN] this is called cooperation by sharing.

2) synchronization of shared procedures. In [RAYN] this is called competition.

3) communication between processes. In [RAYN] this is called cooperation by communication.

4) performance of processes in a particular order. This is not mentioned in [RAYN]. However, it is an important property of concurrent processes.

In practice, these kinds of interaction will not be found separated.

- When a process requests service from a shared process (competition), then in general data will be exchanged (communication) after the request is granted.

- Cooperation by sharing is in fact a mixture of competition and communication. First a process has to request access to the shared data (service request is competition) and then it will read and or write the shared data (communication).

Another kind of process interaction is the procedure call. In software this looks very simple. For example

multiply(operand_1, operand_2, result);

can be the call for executing the procedure (or process), named **multiply**. Parameters are enclosed by parentheses. In this example the operands are input, and the result is output of this procedure. What happens when such a procedure call will be executed?

1) The variables operand_1 and operand_2 will be transferred to the procedure multiply. The calling process stops and waits for the result.

2) The procedure multiply will be executed.

3) The variable result will be transferred to the calling process.

4) The calling process will resume execution.

In the final system description no procedure calls may be left since these are not primitive operators. The procedure calls must be described in detail, like the four steps in the previous example. However, there are other ways. The calling process and the called procedure might execute concurrently for example. In this case synchronization is needed to transfer the result.

As mentioned at the beginning of section 3 of this chapter, the system description (consisting of tasks, processes, procedures, functions) will be transformed to a hardware implementation (digital circuits). For this reason statements are needed in which all kinds of process interaction (as mentioned before) can be described. These statements must be very primitive so they can be transformed to hardware easily.

Since tasks can be performed concurrently, a method for describing concurrency must be defined. In Concurrent Pascal the following notation is used for this:

```
cobegin
  task_1;
  task_2:
  ....
  task_n;
coend;
```

This means that the tasks task_1 to task_n will be performed concurrently. Unfortunately, this notation does not apply to describe all possible kinds of interaction between processes. For this reason four additional synchronization and communication statements will be introduced now. These will be considered to be basic operations. This means that these operations will be directly transformed into hardware. Since interaction between processes is more than just (combinatorial) logic, these operations are not primitive operations, but primitive **protocols**. All higher level and more complex protocols can be built using the next four basic operations.

The synchronization primitives are based on the principle of event indication. This means that processes can cause the occurrence of particular events to indicate that they want to interact in some way. For example, a process that wants to print something, can indicate a printer process that it requests service. On the other hand, processes can wait until particular events have occurred before execution can resume. To continue the example, the printer process does not execute until some process has indicated that it requests service.

1) **signal**(ev[1],...,ev[n]);

Execution of this statement represents the occurrence of the events ev[1],...,ev[n]. So by executing one statement, a process can cause the occurrence of several events at the same time.

Example: Suppose an application process has to print a particular text. The text must also be stored on a disk. For these purposes there are separate processes. There is one process that can print text. Another process can store text on a disk. The application process can indicate that it requests service from both the printer process and the disk access process by executing the statement:

```
signal(print_rqst, write_disk_rqst);
```

The event **print_rqst** is for indicating service request from the printer process. The event **write_disk_rqst** is for indicating service request from the disk access process.

2) **wait**(ev[11] &..& ev[1n] | .. | ev[k1] &..& ev[km]);

ev[ij] represents an event which can occur in another process.
events, coupled by the **&** operator, form a **combination** of events.
combinations of events are separated by the | operator.

Execution of the wait-statement causes the process to wait until a particular combination of events occurs. If such a combination has occurred, then the process will indicate that the events are detected and execution will resume. If there occur more combinations of events, then only one of these combinations will be detected. A priority selection mechanism will decide which combination will be chosen. The other events will be remembered so they can be detected during the next execution of a **wait** statement.

Example: Suppose there is one printer process that can serve three application processes. These processes can request service by executing the **signal** statement by which resp. the events **prt_rqst_1**, **prt_rqst_2**, and **prt_rqst_3** will occur. If the printer process is idle, then it is waiting for service requests by executing the statement:

wait(prt_rqst_1, prt_rqst_2, prt_rqst_3);

If there occurs more than one request at the same time, only one will be granted. The others have to wait until the first application process has been served.

The communication statements will only describe the exchange of data. The synchronization of the transmitting and receiving processes can be described by using the **send** and **wait** statements. The reason for choosing this method is that there are many methods in which communication can be synchronized. All these synchronization methods can be described by using the **wait** and **signal** statements. Then the communication statements only have to describe the exchange of data.

3) **send**(proc_id[1],..,proc_id[n],data);

The transmitting process executes the send statement. This means transmitting the **data** to the receiving processes. The receiving processes are named **proc_id[1]** to **proc_id[n]**.

4) **receive**(proc_id,data);

The receiving process executes the receive statement. This means receiving the **data** from the transmitting process. The transmitting process is named **proc_id**.

Examples: Processes can communicate **synchronous** by first synchronize with each other, and then exchange data directly to each other. This method requires that both processes wait for each other.

Processes can communicate **asynchronous** by using a buffer. The transmitting process first has to synchronize with the buffer, which is described as a process. Then the process can transmit data to the buffer. On the other side the receiving process first will have to synchronize with the buffer, before data from the buffer can be received. In this case the transmitting process does not have to wait until the receiving process is ready to accept data.

Both synchronous and asynchronous communication can be described by using the four previously defined synchronization and communication primitives.

Note that the send and receive statements, which cooperate by using process identifiers, are always executed simultaneously.

For the reasons, mentioned at the beginning of this section (no details, keeping clear overview in communication & synchronization, deadlock and statistical analyses) a separated description of the communication and synchronization behaviour of a digital system will be developed. This can be done by looking only at the four synchronization and communication statements. However, for a better overview a graphical representation of the four statements **wait**, **signal**, **send** and **receive** will be introduced now to describe the synchronization and communication behaviour of coupled tasks. This method uses Petri Nets. First some definitions of Petri net theory will be presented. Secondly an interpretation of Petri nets will be defined when used in digital system description.

3.2.2.1. Petri Net Theory

Petri nets are a tool to study systems. Petri net theory allows a system to be modeled by a mathematical representation of the system. Analysis of the Petri net can reveal important information about the structure and dynamic behaviour of the modeled system. This information can then be used to evaluate the modeled system and suggest improvements or changes.

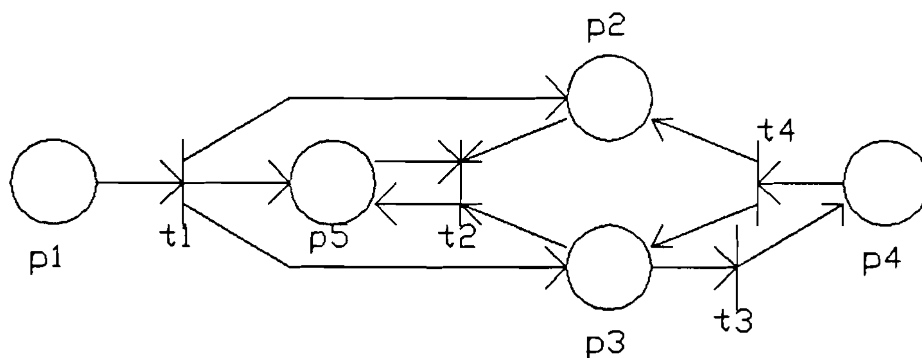
In this section formal definitions for the basic Petri net concepts are presented briefly. For more detail refer to [PETE].

Petri net structure:

A Petri net is composed of four parts: a set of **places**, a set of **transitions**, an **input** function, and an **output** function. The input and output functions relate transitions and places. The input function is a mapping from a transition to a collection of places. The output function maps a transition to a collection of places. The structure of a Petri net is defined by its places, transitions, input function, and output function. Although there is a mathematical representation of Petri nets, only the graphical representation will be presented in this section.

Petri net graph:

A Petri net graph is a representation of a Petri net structure as a bipartite directed multigraph. A Petri net structure consists of places and transitions. Corresponding to these, a Petri net graph has two types of nodes. A **circle** represents a place, and a **bar** represents a transition. Directed arcs (arrows) connect the places and the transitions, with some arcs directed from the places to the transitions and other arcs directed from transitions to places. An arc directed from a place to a transition defines the place to be an input of the transition. Multiple inputs to a transition are indicated by multiple arcs from the input places to the transition. An output place is indicated by an arc from the transition to the place. Again, multiple outputs are represented by multiple arcs. In figure 7 an example of a Petri net graph is given.



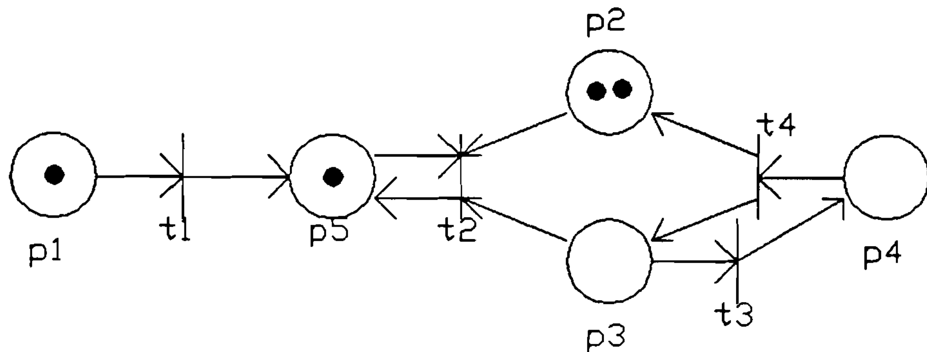
places:	p1, p2, p3, p4, p5		
transitions:	t1, t2, t3, t4		
inputs p1:	none	outputs p1:	t1
inputs p2:	t1, t4	outputs p2:	t2
inputs p3:	t1, t4	outputs p3:	t2, t3
inputs p4:	t3	outputs p4:	t4
inputs p5:	t1, t2	outputs p5:	t2

Figure 7. Example of Petri net graph

Petri net markings:

A marking is an assignment of **tokens** to the places of a Petri net. A token is a primitive concept for Petri nets (like places and transitions). Tokens are assigned to, and can be thought to reside in, the places of a Petri net. The number and positions of tokens may change during the **execution** of a Petri net.

On a Petri net graph, tokens are represented by dots in the circles which represent the places of a Petri net. Figure 8 shows an example of a marked Petri net. The Petri net structure is the same as figure 7.



Marking: p1 contains 1 token
p2 contains 2 tokens
p3 contains 0 tokens
p4 contains 0 tokens
p5 contains 1 token

Figure 8. Example of a marked Petri net graph

Execution of a Petri net:

The **execution** of a Petri net is controlled by the number and distribution of tokens in the Petri net. Tokens reside in the places and control the execution of the transitions of the net. A Petri net executes by **firing** transitions. A transition fires by removing tokens from its input places and creating new tokens which are distributed to its output places.

A transition may fire if it is **enabled**. A transition is enabled if each of its input places has at least as many tokens in it as arcs from the place to the transition. Multiple tokens are needed for multiple input arcs. The tokens in the input places which enable a transition are its **enabling tokens**.

A transition fires by removing all of its enabling tokens from its input places and then depositing into each of its output places one token for

each arc from the transition to the place. Multiple tokens are produced for multiple output arcs. Firing a transition will in general change the marking of a Petri net. In figures 9, 10 and 11 an illustration of the execution of a Petri net is given.

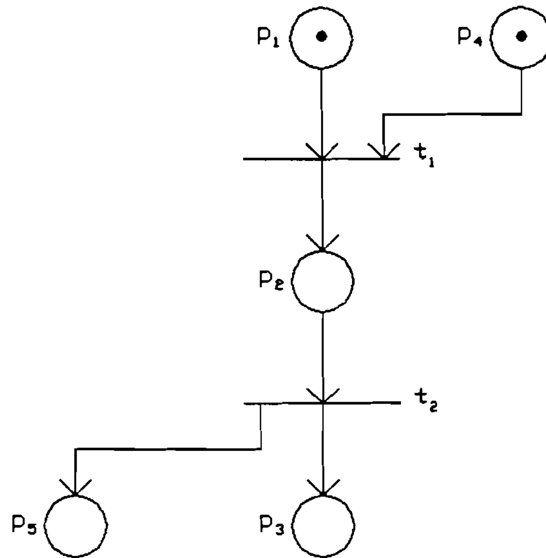


Figure 9. A marked Petri net to illustrate the firing rules. Only transitions t1 is enabled.

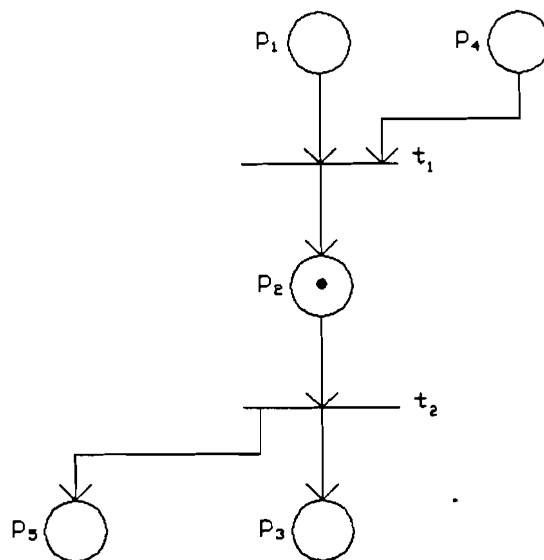


Figure 10. The marking resulting from firing transition t1 in figure 9. Now transition t2 is enabled.

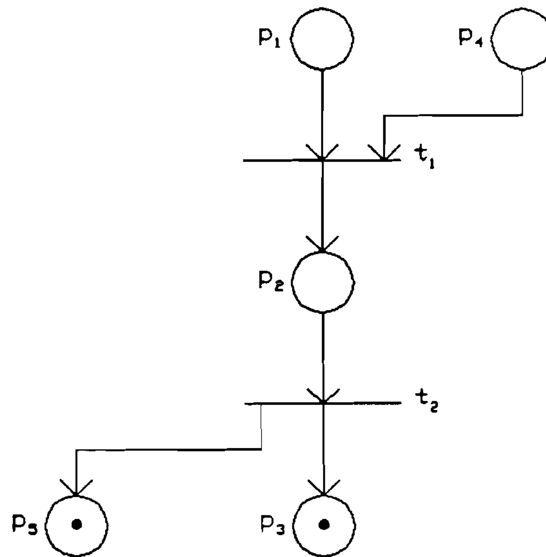


Figure 11. The marking resulting from firing transition t2 in figure 10.

Multiple enabled transitions may fire concurrently if these are not in **conflict**. Enabled transitions are in conflict if firing of one of these transitions removes the enabling token of another enabled transition which results in disabling this transition. An example of a conflict is given in figure 12.

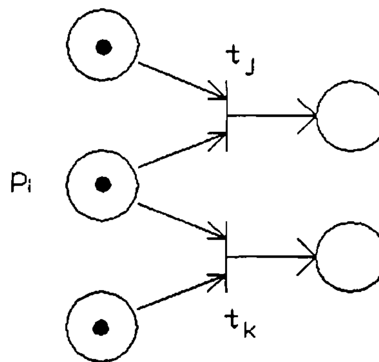


Figure 12. Conflict. Transitions t_j and t_k are in conflict since firing either will remove the token from p_i, disabling the other transition.

So far only theoretical properties of the Petri net theory are given. In the next section the interpretation of Petri nets in relation to digital systems will be discussed. Two remarks can be made to end this section.

1) A very important **advantage** of using Petri nets to describe communication and synchronization is the illustrative power of the graphical representation of Petri nets.

2) A **disadvantage** of Petri net theory is the mathematical weakness, compared to Milners Calculus of Communicating Systems (CCS). However, it is possible to translate Petri net structures into CCS, which makes it possible to calculate the dynamic behaviour of the modeled system. For example, deadlocks can be detected this way. The translation of Petri net structures into CCS is beyond the scope of this report and will not be discussed in detail.

3.2.2.2. Interpretation of Petri net graphs

In this section the modeling of the communication and synchronization behaviour of digital systems will be discussed. As mentioned before, Petri nets will be used to describe this. The question now is: how to interpret places, transitions, and arcs: how to model processes. This question will be answered now.

The description of a digital system contains interacting processes. The synchronization and communication behaviour of these processes depends on when synchronization and communication primitives will execute, resulting in the occurrence and detection of events. Internal actions inside the processes are not important.

In the Petri Net representation of a digital system, two kinds of **places** can be distinguished:

1) Places that represent **internal actions** inside processes, which are not important to know in detail.

2) Places that represent **events**, which can occur and be detected by the execution of the wait and signal statements.

Transitions in a Petri Net represent the synchronization statements wait and signal. The representation of the communication primitives send and wait is joined with the synchronization primitives. This is because these primitives do not describe synchronization, but only data exchange.

Two categories of **tokens** can be distinguished:

1) Tokens in places that represent **internal actions** indicate which part of a process is in execution.

2) Tokens in places that represent **events** indicate that an event has ocurred.

The **arcs** between places and transitions indicate two things:

1) The order in which internal actions and synchronization and communication statements in processes will be executed.

2) the events by which processes interact.

The four previously defined synchronization and communication primitives can be represented in Petri Nets as follows.

1) The graphical representation of the statement

signal(ev[1],..,ev[n]);

in a Petri Net is shown in figure 13.

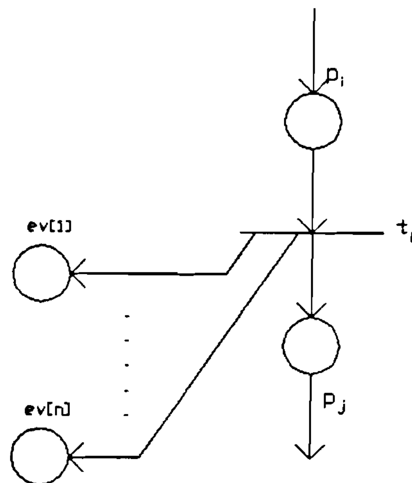


Figure 13. Petri net graph representation of the signal statement

Figure 13 shows the two kinds of places. The places labeled ev[1] to ev[n] represent the corresponding events (the parameters of the signal statement). The places p_i and p_j represent internal actions of the process. When place p_i is marked with a token, then the process performs internal actions; the signal statement has not been executed yet. When place p_j contains a token, then the signal statement is executed. The transition t_i has fired, so the places ev[1] to ev[n] contain a token. This represents the occurrence of the events ev[1] to ev[n].

2) The graphical representation of the statement

wait(ev[11]&ev[12]&..&ev[1n] | ... | ev[k1]&..&ev[km]);

in a Petri Net is shown in figure 14.

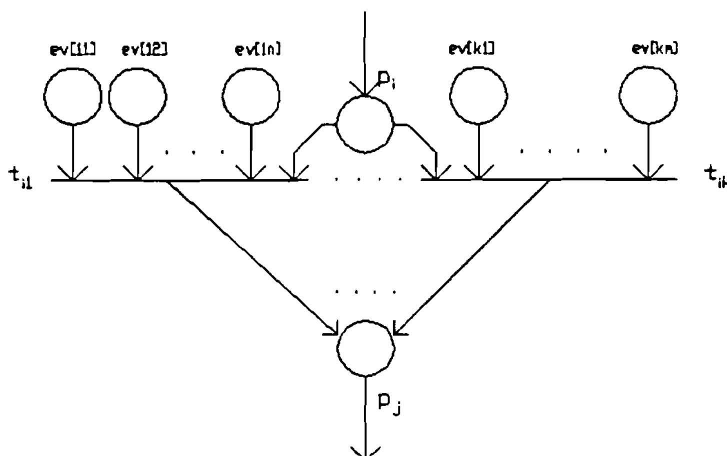


Figure 14. Petri net graph representation of the wait statement

The places labeled ev[11] to ev[km] represent the corresponding events (the parameters of the wait statement). The places p_i and p_j represent internal actions of the process. Figure 14 shows separated transitions for all possible combinations of events. When place p_i is marked with a token, then the task performs internal actions, or is waiting for the enabling of one of the transitions t_{i1} to t_{ik}. When place p_j contains a token, then the execution of the wait statement is completed. One of the transitions t_{i1} to t_{ik} has fired. This means the corresponding combination of events has been detected.

3) The representation of the **send** and **receive** statements in a Petri Net.

Since these two statements are coupled and executed simultaneously, the graphical representation will be combined in one picture.

As mentioned before, the send and receive statements do not synchronize the sender and the receiver. The statements signal and wait will be used for synchronizing the communicating processes. Suppose the processes **proc_i** and **proc_j** have to communicate. An example of a communication protocol in a high level programming language:

```
PROCESS proc_i; {transmitting process}
begin
  {internal actions}
  signal(trx_rdy);  {ready for transmitting}
  {internal actions}
  wait(rec_rdy);     {wait for receiver ready}
  send(proc_j,data); {send data to proc_j}
  {internal actions}
end.

PROCESS proc_j; {receiving process}
begin
  {internal actions}
  signal(rec_rdy);  {ready for receiving}
  {internal actions}
  wait(trx_rdy);     {wait for transmitter ready}
  receive(proc_i,data); {receive data from proc_i}
  {internal actions}
end.
```

The Petri net graph representation of these processes is shown in figure 15.

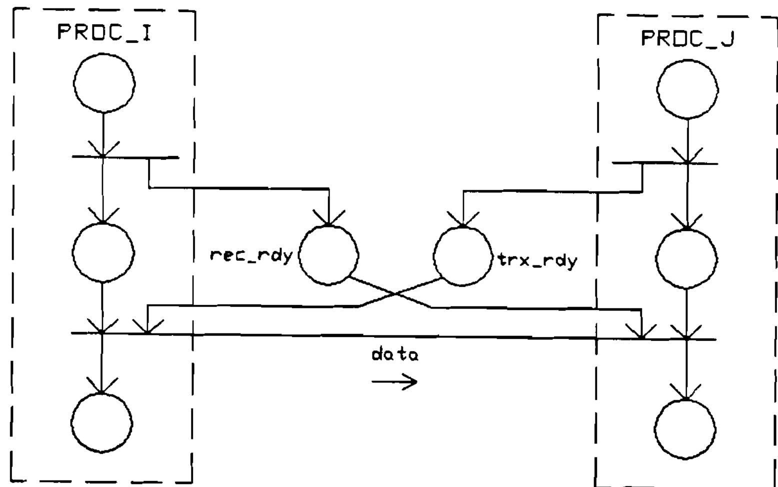


Figure 15. Petri net graph representation of the send and receive protocol

Remarks:

- The places labeled **trx_rdy** and **rec_rdy** represent the corresponding events that are used for synchronization.

- The send and receive statements must execute simultaneously to guarantee a proper exchange of data. For this reason the statements

  ```
  wait(rec_rdy);
  send(proc_j,data);
  ```

  of proc_i, and the statements

  ```
  wait(trx_rdy);
  receive(proc_i,data);
  ```

  of proc_j are represented by only one transition in the Petri Net. This transition is part of both the transmitting and receiving process. When this transition fires, the data will be exchanged.

- Once the communicating processes are synchronized, as many data can be exchanged as will be necessary.

To conclude this section, about the functional decomposition of a digital system into interacting processes, some examples will be discussed to show the ideas of the presented method.

3.2.3. Examples

A lot of literature is available on developing algorithms (sequences of information transformation statements and flow-control statements). For this reason no attention will be paid to the behaviour description of systems. The examples will only deal with communication and synchronization. This includes the mechanism of procedure and function calls.

The last three examples will deal with the different kinds of process interactions as mentioned in section 3.2 of this chapter. The first example explains how to model conditional interactions between processes.

3.2.3.1. Conditional synchronization

Communication and synchronization behaviour can depend on particular conditions. Since these conditions usually come from local information in the process, these are not visible for an observer outside off the process. So for this observer the process shows non-deterministic behaviour. This can be modeled with Petri nets.

Suppose the following very simple process in a high level programming language.

```
PROCESS proc;
begin
  {internal actions}
  if condition
     then wait(event);
  {internal actions}
end.
```

This process can be modeled by a Petri net graph as shown in figure 16.

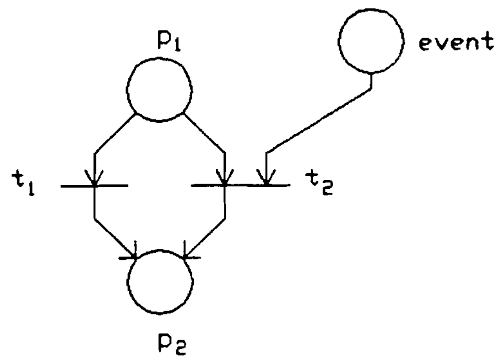


Figure 16. An example of a conditional synchronization action.

Remarks:

- When the process starts (for example by a system reset) p1 contains a token.
- When the process ends p2 contains a token.
- When **condition** is false t1 will fire.
- When **condition** is true t2 will fire if it is enabled by a token in the place labeled **event**.

3.2.3.2. Competition

As mentioned before, resources (processes) can be shared by several other processes. Mutual exclusion of such a resource must be guaranteed.

Consider two processes **proc_1** and **proc_2**. These processes can both call a shared process **proc_s**. This process is not reentrant, so mutual exclusion of executing this process must be guaranteed. In a high level programming language the process descriptions are given below.

```
PROCESS proc_1;
begin
  {internal actions}
  signal(call_1); {service request of proc_s}
  wait(grnt_1); {wait for proc_s ready for service}
  send(proc_s,operands); {pass operands to proc_s}
  {internal actions}
  signal(res_1); {request for receiving results from proc_s}
  wait(rdy_1); {wait until proc_s has ended}
  receive(proc_s,result); {receive result from proc_s}
  {internal actions}
end.

PROCESS proc_2;
begin
  {internal actions}
  signal(call_2); {service request of proc_s}
  wait(grnt_2); {wait for proc_s ready for service}
  send(proc_s,operands); {pass operands to proc_s}
  {internal actions}
  signal(res_2); {request for receiving results from proc_s}
  wait(rdy_2); {wait until proc_s has ended}
  receive(proc_s,result); {receive result from proc_s}
  {internal actions}
end.

PROCESS proc_s;
begin
  repeat
     wait(call_1 | call_2);
     {example priority resolver: }
     if call_1 then req_proc:= proc_1;
                    grant:= grnt_1;
                    ready:= rdy_1;
                    res_rqst:= res_1;
               else req_proc:= proc_2;
                    grant:= grnt_2;
                    ready:= rdy_2;
                    res_rqst:= res_2;
     signal(grant);
     receive(req_proc,operands);
     {internal actions: manipulation of operands}
     signal(ready);
     wait(res_rqst);
     send(req_proc,result);
  until forever;
end.
```

These processes can be modeled by a Petri net graph as shown in figure 17.

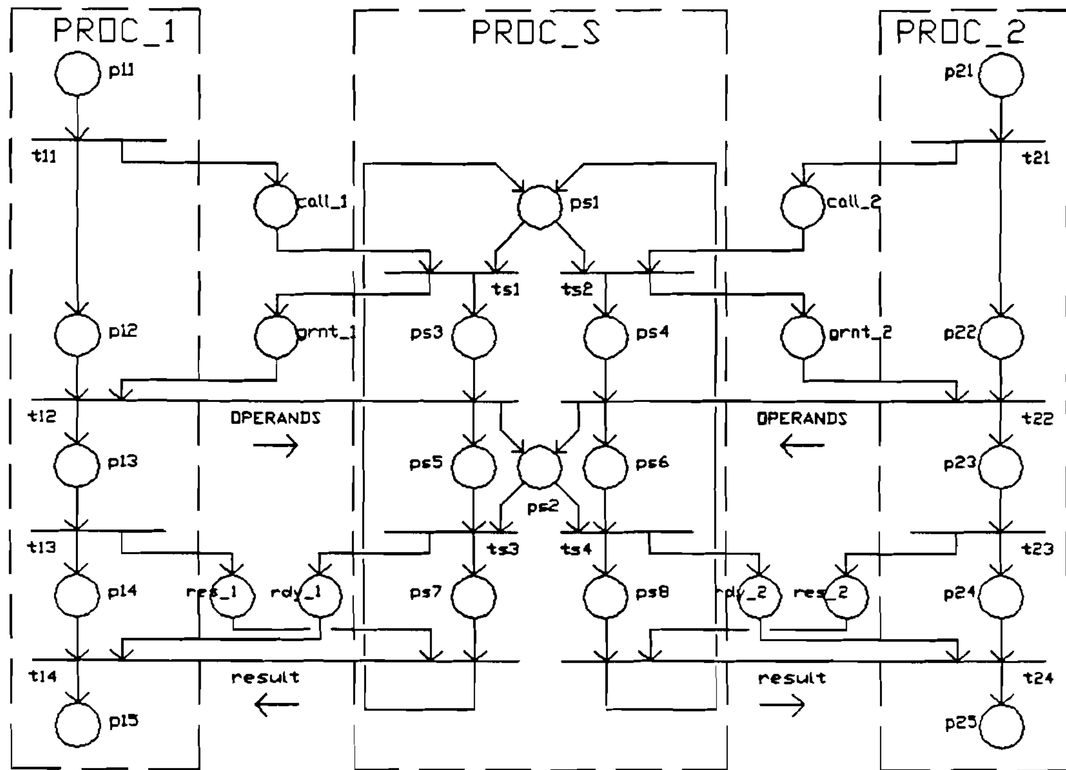


Figure 17. An example of calling a shared process.

Remarks:

- The priority resolver of **proc_s** is modeled by place **ps1**. For an observer outside off **proc_s** this is non-determinism.

- The internal actions of **proc_s** are modeled by place **ps2**.

- The places **ps3** to **ps8** indicate for which process (**proc_1** or **proc_2**) the process **proc_s** is executing. This must be remembered for sending back the **result** to the right process. These places also indicate where the momentary execution of process **proc_s** is. This is for readability reasons only. For example place **ps3** is redundant. It models the same behaviour as place **p1s2**.

- The places **p11**, **p13**, and **p15** represent the internal actions as described in the high level programming language of process **proc_1**. The places **p12** and **p14** indicate where the momentary execution of

process **proc_1** is. This is for readability reasons only. These places are redundant unless process **proc_1** has internal actions at this point, which is not the case in this example. The same holds for process **proc_2** of course.

- The places labeled **call_1**, **call_2**, **grnt_1**, **grnt_2**, **rdy_1**, **rdy_2**, **res_1**, and **res_2** represent the events, used in the system description.

3.2.3.3. Cooperation by sharing

Processes can interact indirectly by using shared variables. The implementation of shared variables, including the controller to guarantee mutual exclusion on handling these variables, result in a shared process. This process is shared by all processes which use these variables. The process exchanges the shared variables with other processes by executing the **send** and **receive** statements. An example of shared variables would result in the same structure as the example in the previous section, so it will not repeated in this section.

3.2.3.4. Cooperation by communication

The third kind of interaction is explicit communication. The Petri net graph which models communication is already discussed in section 3.2.2 of this chapter in which the **send** and **receive** statements were modeled by Petri net graphs.

3.3. Formal system description: structural (de)composition

The result of the first part of the formal system description (functional decomposition) is a description of interacting processes. The description of these processes contains only primitive operations, flow control statements, and communication and synchronization statements. In addition there can be processes that will be realized by using existing (sub)systems (for example IC's that are commercially available). These processes are not described in detail, except for the synchronization and communication behaviour.

Finally, if a hardware implementation is desired, the system description must be transformed into a set of interacting digital circuits. Before implementing the system in hardware, two kinds of problems have to be solved.

1) Processes can be implemented in several ways. For example, a process can be implemented more than once, to avoid bottle necks if the process is shared. The system description does not define **how the processes** will be **implemented**. The designer has to decide this.

2) The system description only described which processes communicate and what data will be exchanged. The description does not define how the data will be exchanged **physically**. For example, data transport can be realized by using one or more busses. The designer must decide how the data transport will be realized on **hardware** level.

When these problems are solved, the system description can be transformed into hardware. This section will discuss briefly how these problems can be solved. It is the second phase in the formal system description process: the **structural (de)composition**.

3.3.1. Implementation of processes

In general processes can be implemented in hardware in two ways.

1) A process can be implemented by transforming the process description into a **digital circuit**.

2) A process description can be substituted, as a **macro**, in the description of the calling process, if there is one.

These two ways of implementing processes will be discussed now more detailed.

3.3.1.1. Digital circuit implementation

A process description can be transformed into a digital circuit (hardware). An advantage of this way is that the parallelism of the system description will also be in the final realization, since the process can execute independently of other processes. A disadvantage is that every process that will be implemented by a digital circuit costs money.

If a particular process can be called by several other processes, then the designer has to think about **how many times** this shared process must be **realized** by digital circuits. The minimum is one, in which case a bottle neck for using this process can occur. The maximum is the number of calls for this process. In this case no bottle necks can occur. The optimal number of process realizations is hard to define. If (statistical) analysis

have been made which forecast the frequency and time density of the use of the process, then these data can be used to define the number of realizations of it. After defining the number of realizations, the next question is how to define an access mechanism for the usage of the process. Two basic principle can be applied.

1) Every realization of the shared process will be shared by a static subset of calling processes. For this solution no additional overhead is required to control the access of the procedure. A disadvantage is that bottle necks still can occur, although other realizations might be idle at that time. This principle is shown in figure 18.

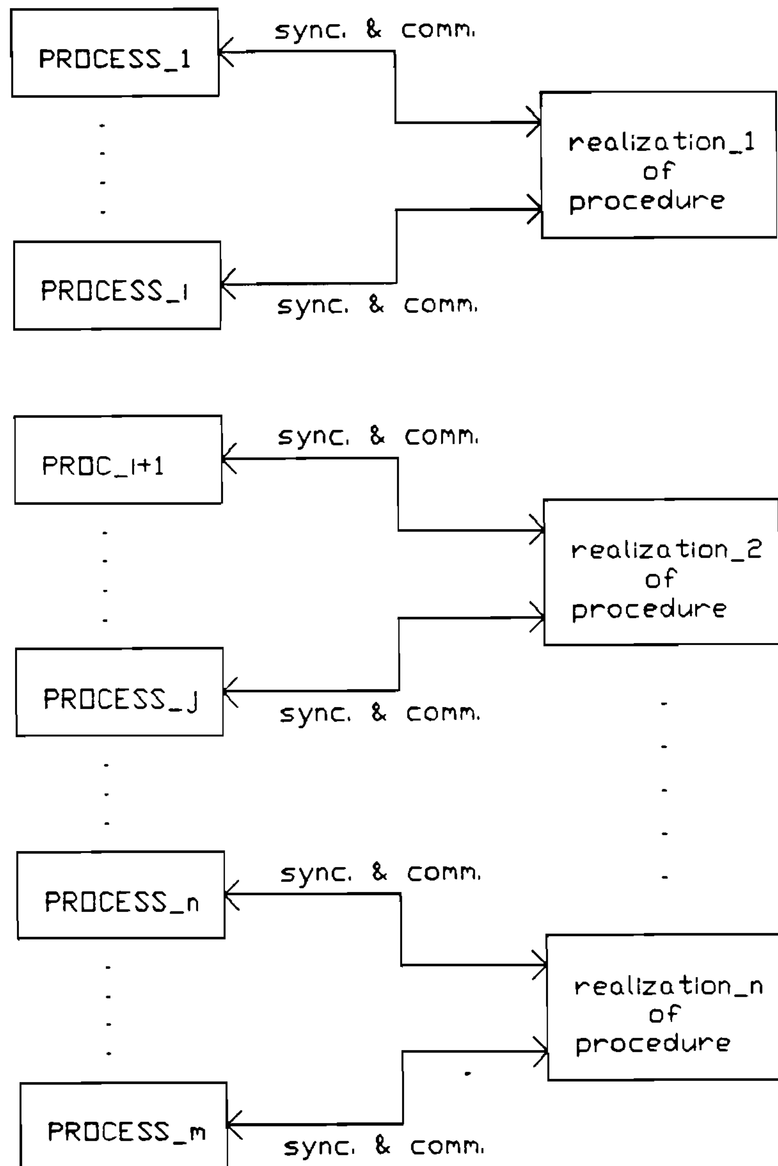


Figure 18. Multiple process realization, shared by a static pool of calling processes

2) A controller can be developed which controls the access to the shared process realizations dynamically. This controller contains an allocation algorithm, including the routing of the data to and from the shared process realizations. The disadvantage of this solution is that extra overhead (hardware) is required, but the chance of bottle necks is smaller. Figure 19 shows this principle.

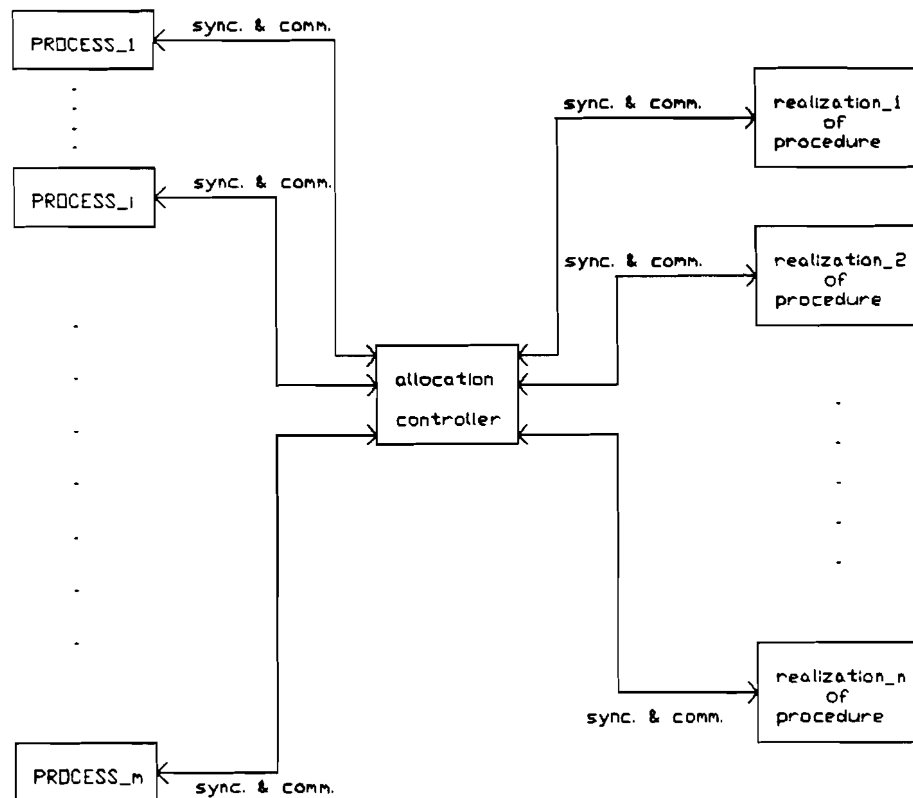


Figure 19. Dynamic allocation of a procedure

3.3.1.2. Macro implementation

A process description can be seen as a macro, so the call for it can be substituted by the process description itself. Of course, at the highest level in the system hierarchy, processes execute autonomous. So at that level there are no calling processes to substitute the description.

The advantage of this principle is that no communication and synchronization actions have to be performed. Bottle necks cannot occur since the process description will be inserted in all calling processes. Depending on the number of calling processes, it can take more or less additional hardware (compared to the previously described principle) when this principle will be applied. A disadvantage of this principle is that the calling and called process cannot execute concurrently, since they are implemented in the same (sequential) circuit. So this kind of realization will execute slower.

3.3.2. Data transport between processes

Two principles can be applied to implement physical connections for data transport between processes.

1) For all communication between processes private connections can be made. This means that if processes want to communicate, then there is always a physical connection available. This connection will be used for the exchange of data between the communicating processes. Figure 20 shows an example of this principle, in which process 1 can communicate with the processes 2, 3 and 4; process 2 can communicate with the processes 1 and 4; process 3 can only communicate with process 1; process 4 can communicate with the processes 1 and 2.

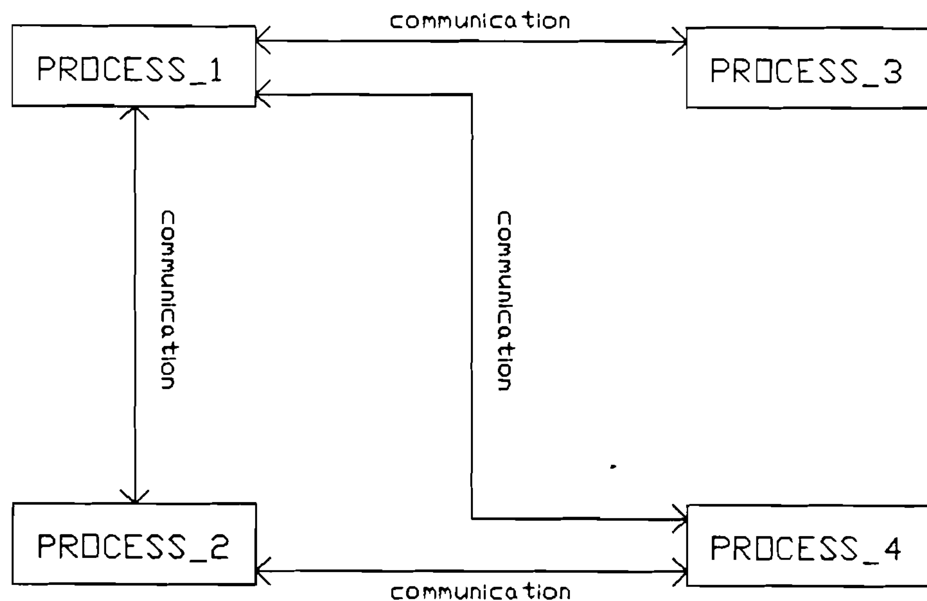


Figure 20. Private connections for all communication actions

2) Shared data interconnection between processes can be defined. This is called a **bus system**. The most extreme case of this principle is only one bus to perform all communication of the entire system. This is shown in figure 21.

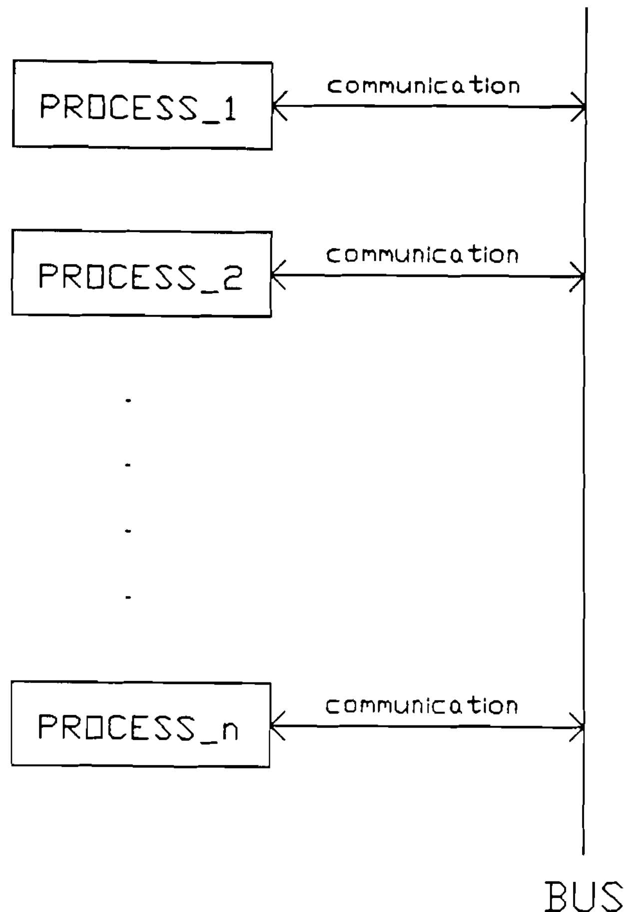


Figure 21. Communication by using a bus system

Of course it is possible to define more than one bus to increase the data transfer capacity of the system.

A bus system needs a control mechanism to guarantee mutual exclusion for usage of the bus. This control mechanism can be described as a shared process in a high level programming language.

An advantage of a bus system, compared to private connections, is a significant lower number of connections between processes. Specially in integrated circuits this is very important since connections need a large

surface on the chip, which costs money. A disadvantage is that a bus can be a bottle neck, since it is a shared process. So the execution speed of the system can be slower this way.

3.4. Conclusion

Functional and structural decomposition leads to a formal description of a digital system. This description has several properties.

- The system is described by a number of interacting processes. Every process description will be transformed into a digital circuit.

- The information transformation statements which are used in the system description are all primitive operations. These operations will be realized in combinatorial logic.

- Besides the information transformation statements the system description contains flow-control, and communication and synchronization statements. These statements will also be transformed to digital logic.

A general system architecture, in which the system description can be transformed, will be the subject of the final chapter. There will be discussed how the primitive operators, the flow control statements, and the synchronization and communication statements can be realized in hardware.

4. A GENERAL SYSTEM ARCHITECTURE

The previous chapter stated that all processes, of which the system description consist, will be realized by interacting digital circuits. In this chapter a general system architecture will be described to implement a digital system in hardware. This means the transformation of the system description into (coupled) state machines.

4.1. System architecture decomposition

To be able to transform systematically a system description into hardware, it is necessary to (de)compose this hardware in the same way as the description is (de)composed. Analyzing the system description, three parts can be distinguished.

1) **Processes.** As a result of the functional and structural decomposition, the system description consists of separate interacting processes. For each process a state machine will be designed.

2) **Event management.** The synchronization of processes is described by applying an event mechanism. This mechanism operates 'between' the processes (events do not belong to one process). So in hardware this mechanism will also be implemented separated from the processes.

3) **Data interconnection.** Communicating processes require data interconnection between these processes. Recalling the previous chapter, this interconnection can be realized by applying one of two principles. First, simple interconnection can be used for interconnecting processes. This is just wiring. Secondly, a bus system can be applied. This can be described as a shared process. In this case a state machine must be designed.

The general system architecture can now be defined as a result of this system decomposition. Figure 22 shows this architecture.

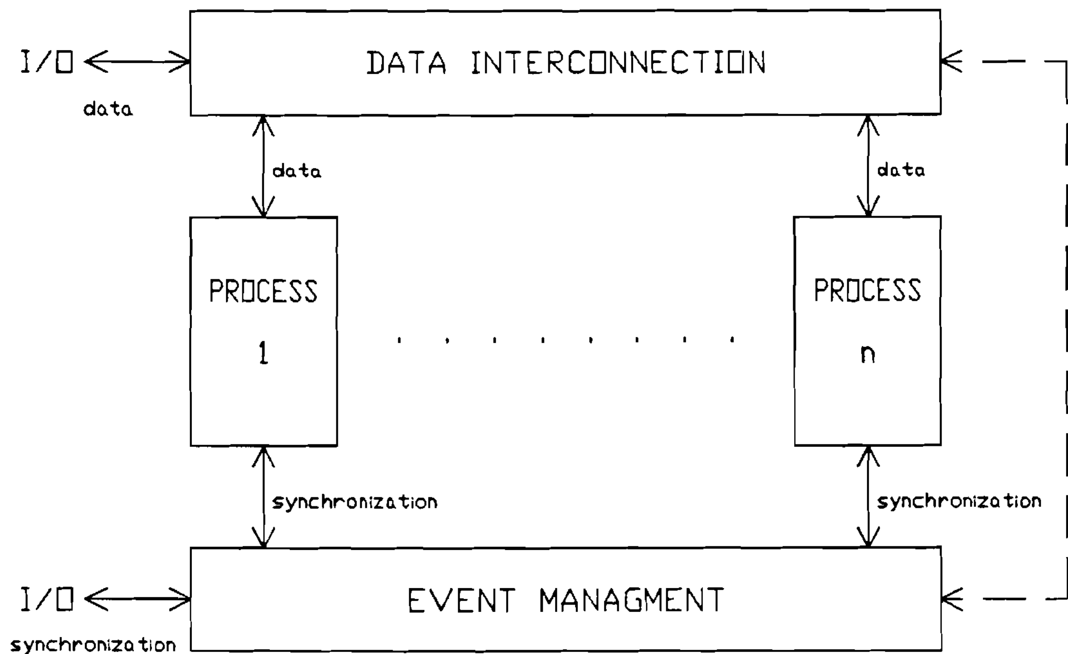


Figure 22. General system architecture

Note that all blocks in this architecture represent state machines, except for the data interconnection. This **might** be only interconnection. In that case no connections exist with the event management.

All state machines will be of the synchronous type, and all storage elements will be clocked by the **same** clock signal. Note that no clock signal has been defined so far in the behaviour description of a digital system.

The following sections will deal with the implementation of **processes** and **event management**. The data interconnection will not be discussed separated, since it is simply wiring, or a shared process.

4.2. Process implementation

The state machine that will be used to implement one process is decomposed into two parts:

1) the **processing unit**. This state machine contains memory elements to store the value of all process variables. It also contains the combinatorial logic that realizes the primitive

information transformation operations. Connections with other processing units via the **data interconnection** (refer to figure 22) may be used for communication.

2) the **control unit**. This state machine contains the algorithm that has to be executed. Control signals going to the processing unit will be generated to perform the operations of the algorithm. Status signals from the processing unit are available to evaluate conditions, which affect the program flow. Connections to other control units via the **event manager** (refer to figure 22) are used to realize the synchronization of processes.

This principle of process decomposition is described in [DAVI]. In figure 23 the brief outlines of the system architecture of one process is shown. In the following sections the processing and control unit will be described more in detail.

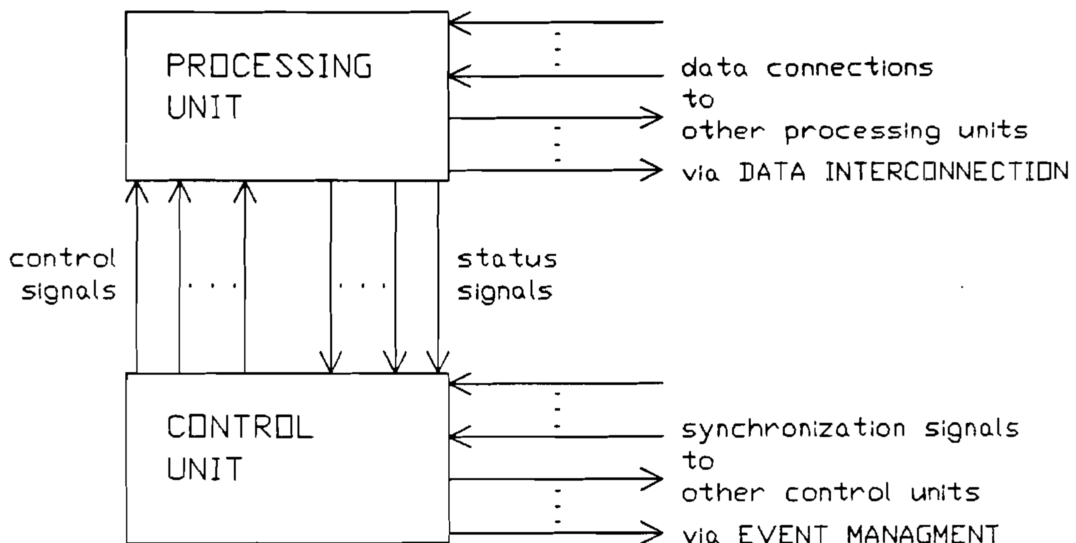


Figure 23. General process architecture

4.2.1. The processing unit

As mentioned before, the processing unit consists of several parts. Storage elements will be used to store the process variables. Combinatorial logic performs the primitive operations and generates status signals. These are used in the control unit for the programs flow control.

The different parts of the processing unit will be discussed now more in detail.

4.2.1.1. Storage elements

The storage elements, or registers, are of the synchronous type. Each memory element has four connections, as shown in figure 24.

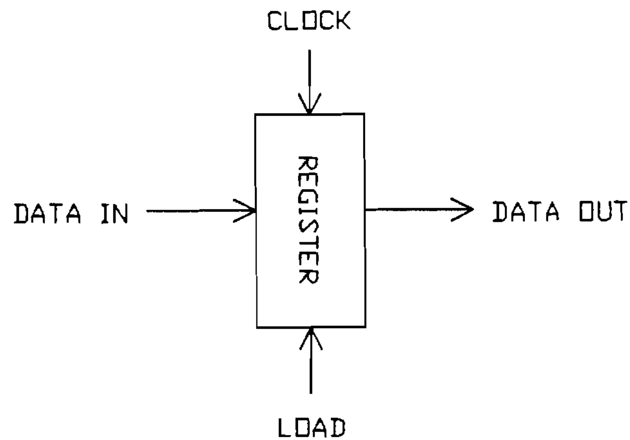


Figure 24. Representation of a storage element

Data in: This is the data that can be stored into the register.

Data out: This is the data that is presently stored in the register.

Load: This signal controls the register. If this signal is active then the **data in** will be clocked in the register. If this signal is not active then the content of the register will not be changed when clocked. The **load** signal comes from the control unit that is connected to the processing unit to which this register belongs.

Clock: This is the synchronous clock signal that is connected to all storage elements in the digital system. It is not interesting at this moment what type of clocking is used. The clock signal can be a single signal, to be used by edge triggered flip-flops. It can also be a two phase, non overlapping clock signal, used for clocking Master-Slave latches.

4.2.1.2. Primitive operations and status

Combinatorial logic must implement the primitive operations, and generate the status signals, that are used as conditions in the process description. These elements will be called computational elements. In figure 25 the general outline of such an element is shown.

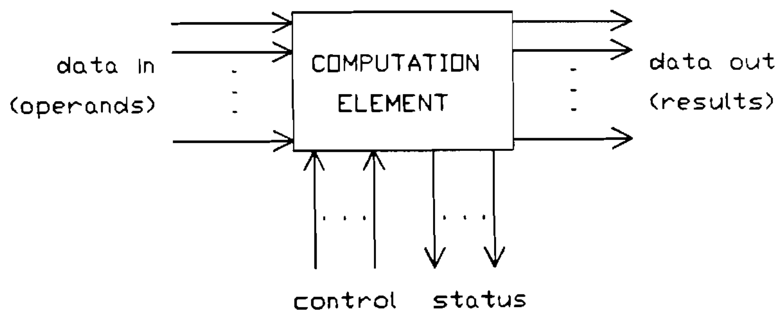


Figure 25. General outline of a computational element

Data in: Input of the computational element is a number of operands on which a particular operation must be performed. The number of operands depends on the definition of the primitive operators.

Data out: Output of the computational element is a number of results of a particular operation. The number of results depends on the definition of the primitive operators.

Control: It is usual to implement one or more primitive operations into one computational element. The reason for this is simply the fact that a process description consists of operations that will be performed sequentially. In that case one needs only one computational element at the time. Although, there may be more computational elements if the process description contains simultaneous performance of operations. If more than one operation is implemented in the computational element, then a **control** signal must indicate which operation must be performed. This signal is generated by the corresponding control unit.

Status: Flow control statements are based on the evaluation of particular conditions. These conditions, or **status signals**, are boolean (true or false) and depend on the result of primitive operations. They are outputs of the computational element. Some examples of status signals (A and B are values in storage elements):

- $A + B > 10$
- $A < B$
- $B = 0$

A special kind of operational element is the **interconnection element**. These elements are used to control the data flow in a processing unit, to perform the assignment statement, and to route the variables between storage elements and computational elements. The interconnection

elements are in general multiplexers and busses, controlled by tri-state gates. The control signals for the multiplexers and tri-state gates are generated by the corresponding control unit.

4.2.1.3. State machine representation

When a processing unit is considered as an ordinary state machine, it can be represented by combinatorial next-state and output decoders, and memory elements that contain the present state. This is shown in figure 26.

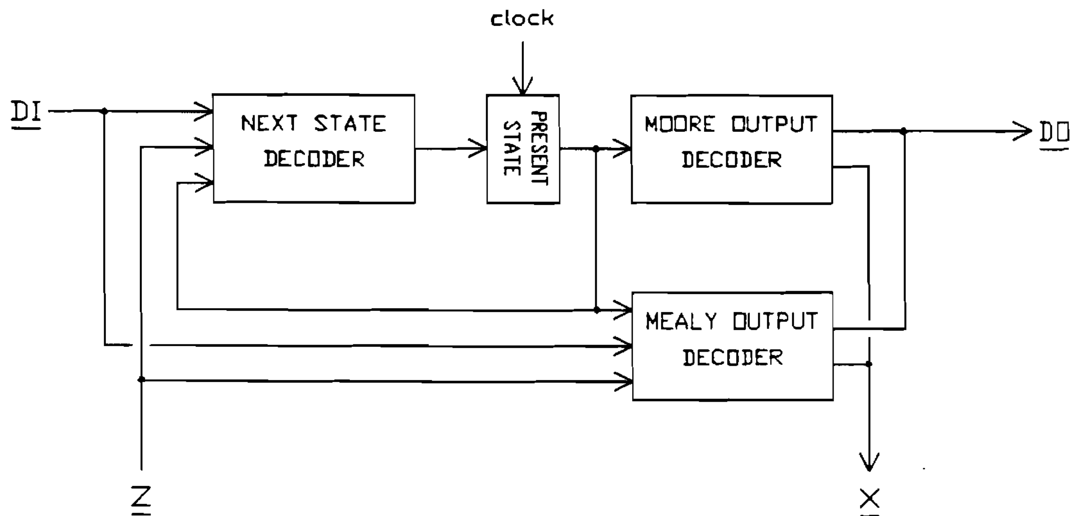


Figure 26. State machine representation of a processing unit

The signal names in this figure have the following meaning.

Z: The **control** signals which are generated by the control-unit and control the registers, computational elements, and interconnection elements will be considered to be a **vector**, named **Z**.

X: The **status** signals that are outputs of computational elements, will be considered to be a **vector**, named **X**.

DI: **Data input** of the processing unit. The data comes, via the data interconnection, from other processes or from the system environment (refer to figure 22).

DO: **Data output** of the processing unit. The data goes, via the data interconnection, to other processes or to the system environment (refer to figure 22).

Like any other state machine, a processing unit can be implemented as a Mealy or a Moore machine. In appendix B the differences in applying these machines are analyzed. As a general conclusion, the choice between Mealy and Moore machines affects the **execution speed**:

1) The number of **operations per clock cycle** that can be performed, so the execution speed, is affected by this choice.

2) The combinatorial delay in a processing unit depends partly on the use of Mealy or Moore machines. That means the maximum **clock frequency**, so the execution speed, is affected by this choice.

4.2.1.4. Example

In figure 27 an example of a processing-unit is given.

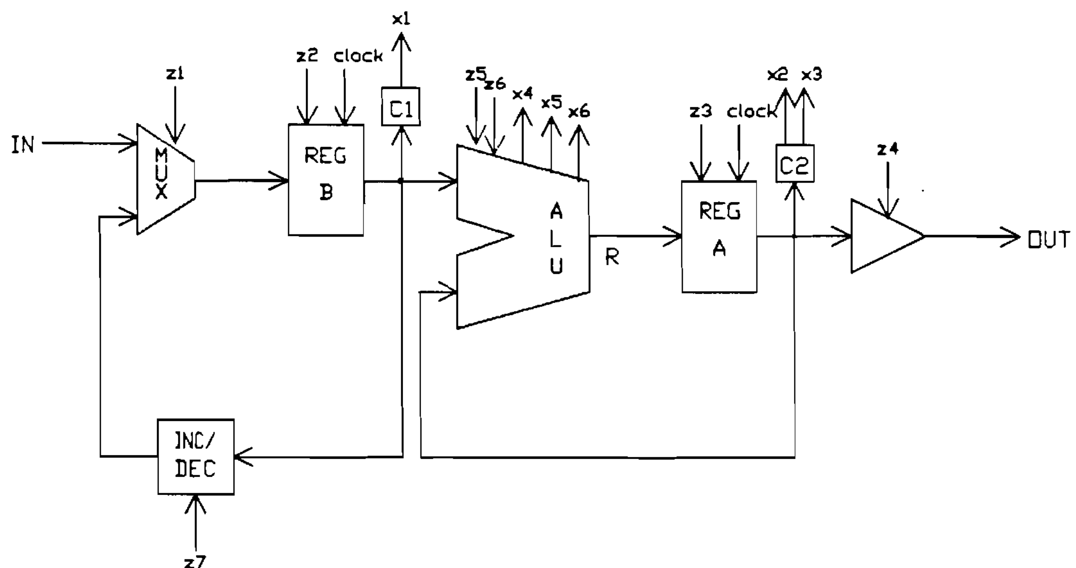


Figure 27. Example of a processing-unit

This processing-unit has the following characteristics.

- A and B represent the registers in which the process variables can be stored.

- IN is a data input of the processing unit. It comes, via the **data interconnection**, from other processes or the system environment.

- The control vector $\underline{Z}$ = (z1,z2,z3,z4,z5,z6,z7). The meaning of the elements:

    - z1 controls the interconnection element that selects the input of register B

    - z2 is the **load** signal for register B

    - z3 is the **load** signal for register A

    - z4 controls the tri-state gate (an interconnection element). The output of this gate goes, via the **data interconnection** to other processes or to the system environment.

    - z5 and z6 control the ALU (a computational element). The ALU can perform four functions (R is the result of the ALU):
        - R = A + B
        - R = A - B
        - R = B
        - R = A + 1

    - z7 controls the increment/decrement computational element.

- The status vector $\underline{X}$ = (x1,x2,x3,x4,x5,x6). The meaning of the elements:

    - x1 is the output of a computational element that computes the condition **B** = 0. This is a zero flag of register B.

    - x2 and x3 are outputs of a computational element that computes resp. the conditions A = 0 (zero flag of A) and A > 0 (sign flag of A).

    - x4, x5, and x6 are outputs of the ALU (computational element) and represent resp. the conditions **carry out**, **overflow**, and R = 0 (zero flag of the result).

When this processing unit will be considered as a state machine, according to figure 26, the data input **IN** and the vector $\underline{Z}$ are the input of the state machine. The content of the registers A and B are the present state, and the data output **OUT** and the vector $\underline{X}$ are the output of the machine. In the given configuration, the outputs x1, x2, and x3 are Moore outputs, since these outputs only depend on the present state. The other outputs x4, x5, x6, and **OUT** depend on the present state and on the control vector $\underline{Z}$, so these are Mealy outputs. In general, a processing-unit is a mixture of a Moore and a Mealy machine.

Note that the conditions x2 and x6 contain the same information: the zero flag of the result of the ALU, that can be clocked in register A.

The difference is that x6 is a Mealy output that will be available before register A is clocked. x2 is a Moore output that will be available after clocking register A.

4.2.2. The control unit

As mentioned before, the control-unit contains the algorithm that has to be executed. This algorithm is described in a high level programming language, as discussed in the previous chapter. There are many ways to implement a control-unit. Refer to [DAVI] for details. In figure 28 the control-unit is represented as a state machine.

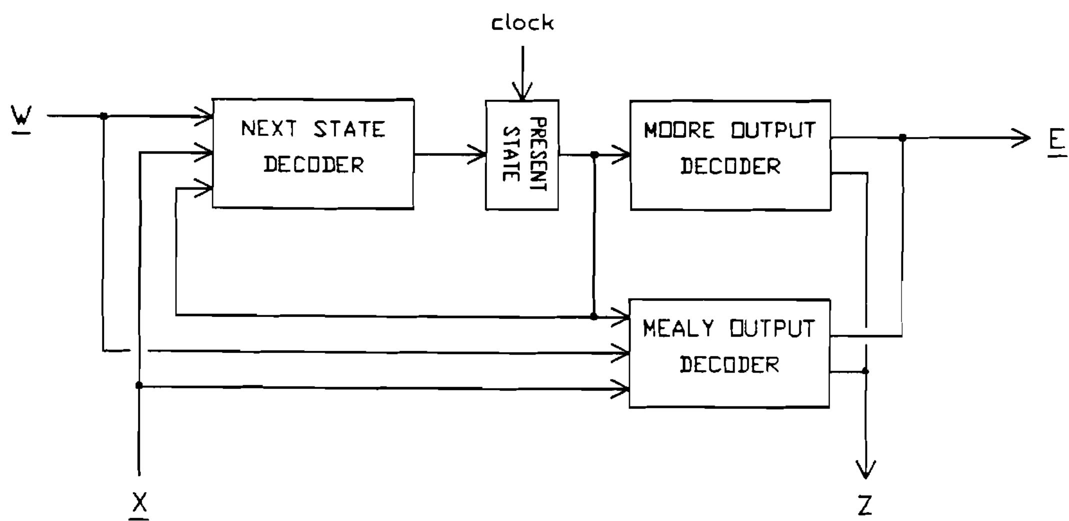


Figure 28. State machine representation of a control-unit

The signal vectors X and Z have the same meaning as described under the previous section. They are connected to the corresponding signals in the processing unit. The other two signal vectors W and E are used for the implementation of the synchronization statements **wait** and **signal**. They are connected with the **event manager**. In the next section the implementation of the synchronization mechanism will be discussed in detail.

In general the control unit is, just like the processing unit, a mixture of a Mealy and a Moore machine. This is shown in figure 28. Refer to appendix B for details about differences between Mealy and Moore machines in system performance. Note that the present state storage elements are clocked by the same clock signal as the registers in the processing unit (synchronous system).

Every clock cycle the control unit can generate a vector Z of control signals to perform activities (primitive operations, routing of data) in the processing unit. On its turn, the processing unit can generate the status vector X every clock cycle. This vector is an input to the control unit and can affect the next state and/or the output vector (flow control statements).

So far, two of the three types of statements have been discussed now. The performance of **primitive operations** is implemented by designing a control unit that generates control signals for a processing unit. In this processing unit the operation is performed by using computational elements, interconnection elements, and registers. The control unit executes **flow control statements** simply by evaluating the status vector X, generated in the processing unit. The next state and/or output vector Z of the control unit now depends on the value of particular status signals (conditions).

The third and last type of statements is the set of synchronization and communication statements **wait**, **signal**, **send**, and **receive**, as defined in the previous chapter.

The communication statements **send** and **receive** only describe what data with what process must be communicated. No synchronization is described by these statements. So the execution of the **send** statement is simply routing data from inside the processing unit of the transmitting process to the **data interconnection** between the communicating processes (refer to figure 22). The execution of the **receive** statement is routing data from the **data interconnection** to inside the processing unit of the receiving process.

The implementation of the synchronization statements **signal** and **wait** is subject of the next section.

4.3. Event management implementation

In figure 13 and 14 in the previous chapter the Petri net representations of the **signal** and **wait** statements are shown. The occurrence of events is represented by **tokens** in **places**. There are two possibilities: an event has occurred or not, so these places can contain zero or one token. That means they can be realized by one single flip-flop. Since the system is synchronous the flip-flop is clocked by the same signal as all other storage elements in the system. The following state assignment of the flip-flop will be defined:

- If the place contains a token, then the flip-flop has the value 1.

- If the place contains no token, then the flip-flop has the value 0.

When this definition is used, the implementation of the **signal** and **wait** statements can be made easily. A clocked Set-Reset flip-flop will be used for the event management.

As an example, figure 29 shows the control units of two processes, and one clocked Set-Reset flip-flop, labeled **event**, for the event management.

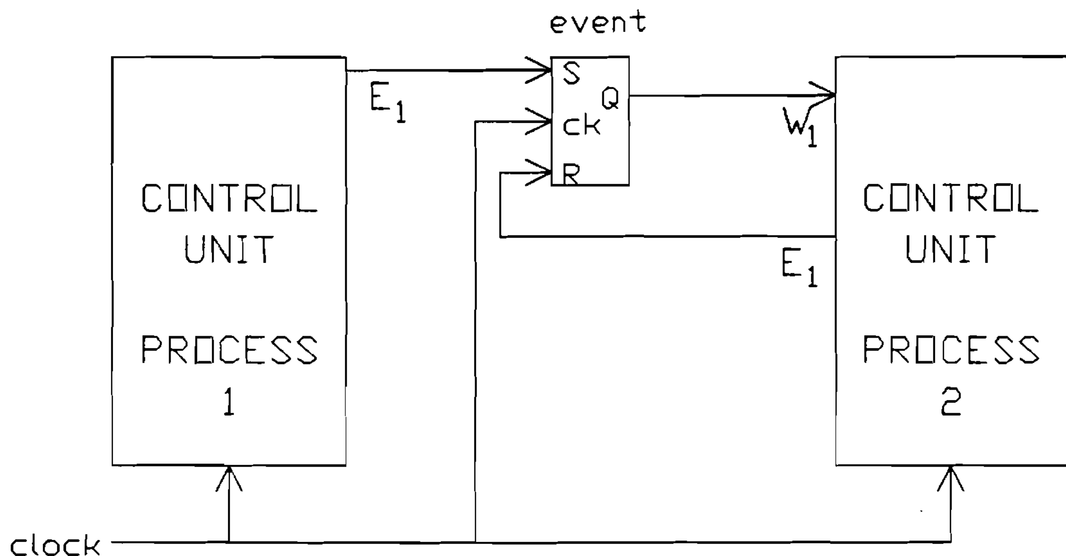


Figure 29. Implementation of synchronizing processes

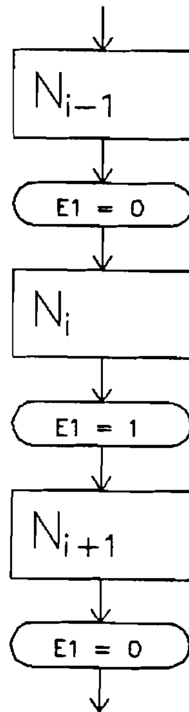
In this configuration process 1 can execute the statement

signal(event);

by setting the **event** flip-flop. This is done by making the signal E_1 of its control unit '1' for one clock cycle. The **event** flip-flop then will be set when clocked. An ASM chart of this **signal** execution is shown in figure 30a. Process 2 can execute the statement

wait(event);

by waiting until the **event** flip-flop contains the value '1'. For this purpose, the flip-flop output is connected with input W_1 of its control unit. After detecting this '1', process 2 will reset the flip-flop by making the signal E_1 of its control unit '1' for one clock cycle. The **event** flip-flop then will be reset when clocked. An ASM chart of this **wait** execution is shown in figure 30b.



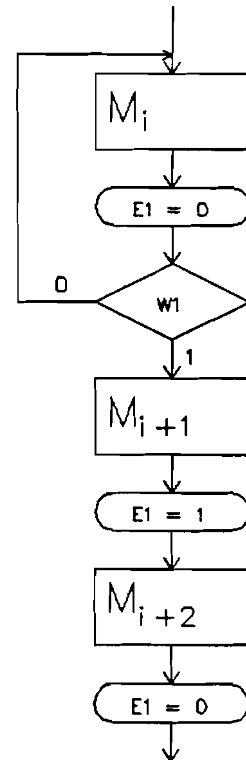


Figure 30a. **signal(event)** Figure 30b. **wait(event)**

In general, the hardware implementation of the statement

signal(ev[1],ev[2],..,ev[n]);

will be according to figure 31. In this figure the control unit of the process that can execute the **signal** statement is connected with the Set inputs of the flip-flops that manage the events. These flip-flops are labeled ev[1] to ev[n]. The Reset inputs and Q outputs of these flip-flops are not shown in this figure. They are connected with processes that can execute the **wait** statement on these events.

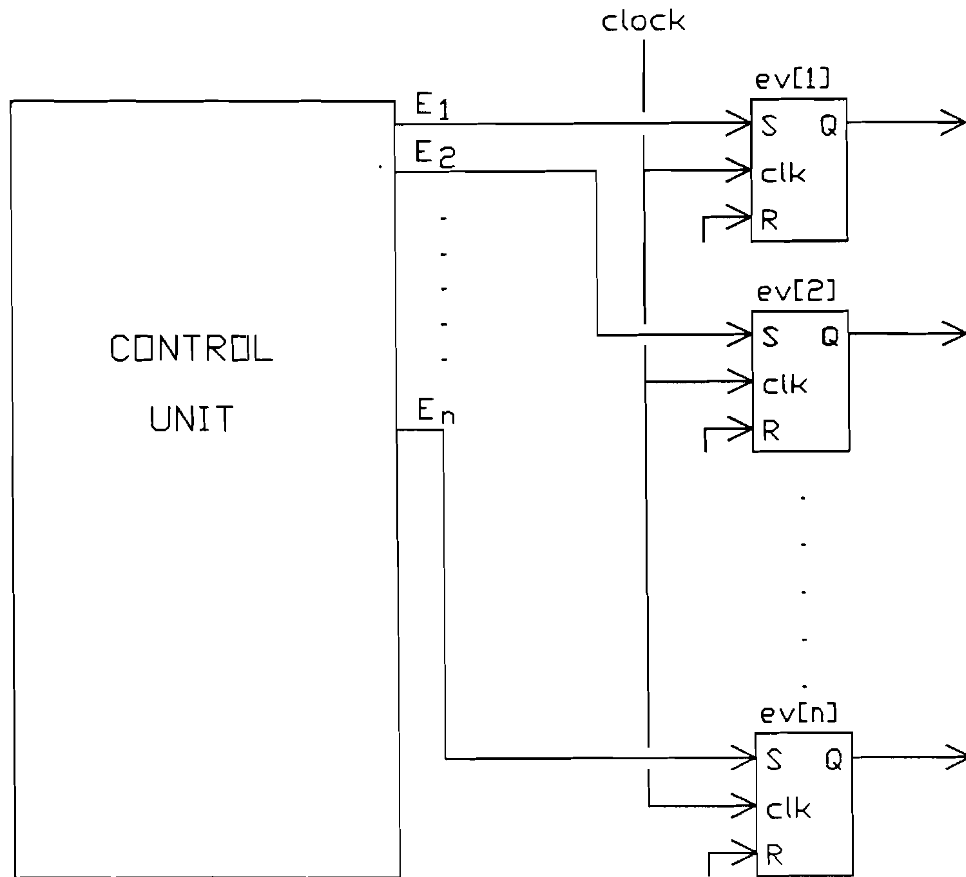


Figure 31. Implementation of **signal** statement

The signals E_1 to E_n are outputs of the control unit. These are elements of the vector **E** (refer to figure 28). The execution of the **signal** statements is very simple. The outputs E_1 to E_n will be made '1' for one clock cycle, so that the **event** flip-flops will be set when clocked. After that, the outputs E_1 to E_n will be made '0' again, and the control unit continues executing the next statements of the process. An ASM chart representation of this **signal** statement is shown in figure 32.

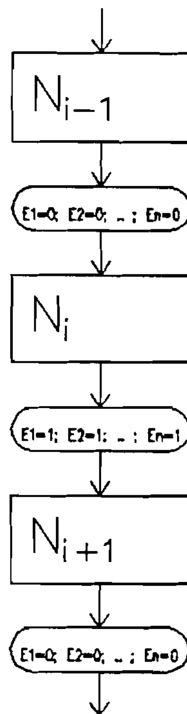


Figure 32. ASM chart of the **signal** implementation

In general, the hardware implementation of the statement

wait(ev[11]&ev[12]&..&ev[1n] | ... | ev[k1]&..&ev[km]);

will be according to figure 33. In this figure the control unit of the process that can execute the **wait** statement is connected with the Reset inputs and the Q outputs of the flip-flops that manage the events. These flip-flops are labeled ev[11] to ev[km]. The Set inputs of these flip-flops are not shown in this figure. They are connected with processes that can execute the **signal** statement on these events.

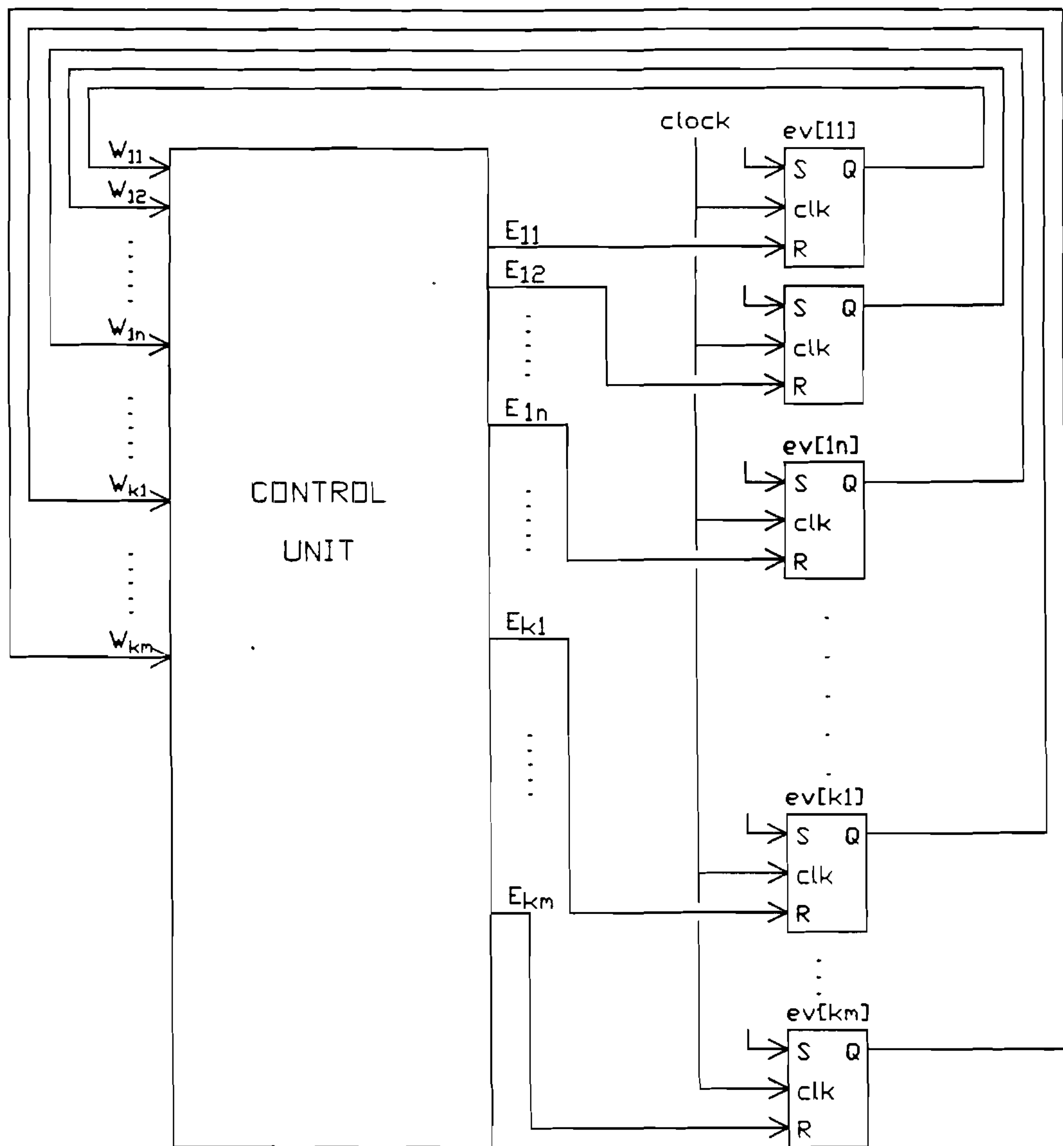


Figure 33. Implementation of wait statement

The signals $E_{11},...,E_{km}$ are outputs of the control unit. These are elements of the vector $\underline{E}$ (refer to figure 28). The signals $W_{11},...,W_{km}$ are inputs of the control unit. These are elements of the vector $\underline{W}$ (refer to figure 28). During execution of the **wait** statement the signals $W_{11},...,W_{km}$ are evaluated until all events in one combination have occurred. Then the corresponding reset signals of the vector $\underline{E}$ will be made '1' during one clock cycle. The flip-flops of the detected events then will be reset when clocked. An ASM chart representation of this **wait** statement is shown in figure 34.

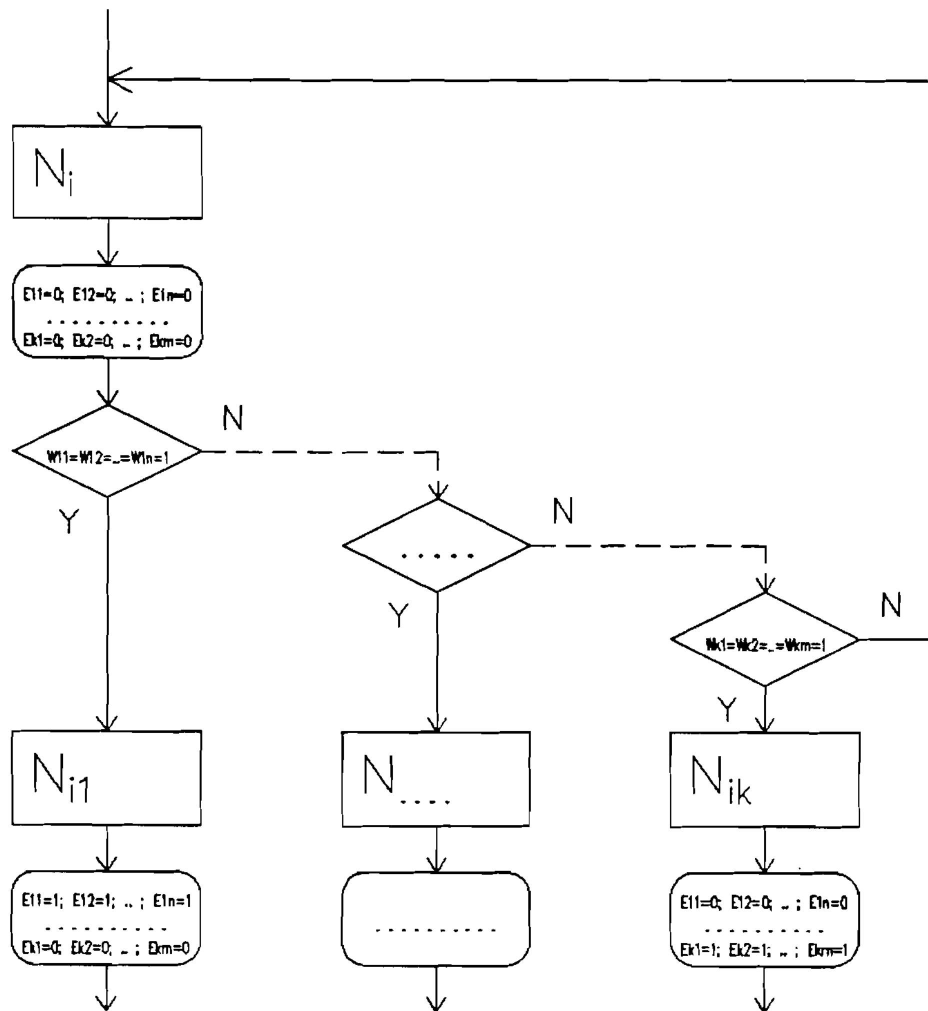


Figure 34. ASM chart of a wait implementation

Note that the ASM chart of figure 34 shows the priority resolving algorithm used in the wait statement, because it is the detailed representation of the control unit. A very simple priority resolver is chosen as an example. The first combination of events has the highest priority, the last has the lowest priority. Of course it is possible to show this priority resolver also in a Petri net graph. The disadvantage is that the graphical representation of the communication and synchronization in the system will be less clear that way.

Event flip-flops never can be set and reset simultaneously. The Reset signal cannot be active before the value '1' at the Q output of the flip-flop has been detected. That will be at least one clock cycle after an active Set signal. Since Set and Reset signals are active for only one clock cycle, they cannot be active simultaneously.

The ASM chart of figure 34 goes with a Moore machine. However, the synchronization statements can also be implemented with Mealy machines. In appendix C the execution speed of various configurations has been analyzed.

4.4. Conclusions

In this chapter a general system architecture is presented. This architecture can be used to transform the system description into hardware. **Event management**, used for implementing the synchronization statements, is discussed in detail. Details about how to implement processing and control units are beyond the scope of this report. However, a lot of literature (for example [DAVI]) can be found about this subject.

In this chapter, the processing and control units were represented by ordinary state machines. In appendix A and B these state machines are analyzed in detail. Appendix A deals with different types of storage elements, embedded in sequential circuits. Testability and switching frequency of various configurations have been analyzed. In appendix B the control unit will be connected with the processing unit. The consequences of doing this will be discussed. Performance analyses has been made concerning testability, and execution speed of information transformation and flow control statements. Finally, in appendix C control units are connected to each other via the **event manager**. For various configurations the execution speed of synchronization and communication statements has been analyzed.

CONCLUSIONS AND FURTHER RESEARCH

Digital systems can be realized both in hardware and in software. The methodology for designing software has been analyzed because in this subject a lot of research effort has been spent. The result is that software description languages (for example Concurrent Pascal) can also be used for describing and designing hardware. In the language a number of primitive operators must be defined. These operations finally will be realized by combinatorial logic. Statements for synchronization and communication must be added to be able to design parallelism in systems. For the transformation from the Pascal-like system description to a digital circuit a special kind of system architecture will be used.

A lot of research effort still has to be spent to develop CAD tools for designing digital systems in a structured way.

- Tools must be developed for simulating the communication and synchronization behaviour of systems. The frequency of process calls can be traced, to detect potential bottle necks. Starvation can be detected by looking at the duration of time processes have to wait for other processes.

- Tools must be developed to calculate the communication and synchronization behaviour of systems. In this way it is possible to detect dead locks.

- Tools must be developed to perform parts of the design process automatically. For example, the transformation from the final Pascal-like description, that contains all details, to a hardware realization can be done automatically.

- Tools must be developed that can be used during the functional decomposition phase. When a designer decides to make a particular decomposition, then it would be very useful to analyze the consequences of this decision for the final system characteristics (speed, costs).

This report can be a basis of further research to structured and computer controlled system design methodologies.

REFERENCES

[ANCE] Francois Ancean
The architecture of microprocessors
Addison-Wesley publishing company, 1986

[ANDR] F. André, D. Herman, J.P. Verjus
Synchronization of parallel programs
North Oxford Academic, 1985

[BENA] M. Ben-Ari
Principles of concurrent programming
Prentice Hall International, London, 1982

[DAVI] M. Davio
Digital systems with algorithm implementation
John Wiley & Sons, 1983

[HARE] David Harel
Statecharts: A visual approach to complex systems
Department of Applied Mathematics,
The Weizmann Institute of Science, Rehovot, 1986

[HARL] David M. Harland
Concurrency and programming languages
Ellis Horwood Limited, Chichester, 1986

[KLIN] E. Klingman
Micro processor systems design, volume II
Prentice Hall, Inc., Englewood Cliffs, 1982

[KOOM] C.J. Koomen
Systeemtechnology I
University of Technology, Eindhoven, 1985

[PETE] James L. Peterson
Petri net theory and the modeling of systems
Prentice Hall, Inc., Englewood Cliffs, 1981

[RAYN] M. Raynal
Algorithms for mutual exclusion
North Oxford Academic, 1986

[STEV] M.P.J. Stevens
Structured VLSI Design Course
University of Technology, Eindhoven, 1987

APPENDIX A: Sequential Circuit Configurations

Sequential circuits can be realized in various ways. Different types of storage elements can be used. Combinatorial logic can be embedded in the circuit in different ways. These matters affect testability and maximum clock frequency of the circuit. In this appendix some configurations will be analyzed. The results can be applied to design the hardware implementation of processes (control and processing units), and analyze its performance (testability, execution speed).

1. Storage elements

Two different kinds of storage elements can be used in sequential circuits.

1) Edge triggered flip-flops. This type of storage element switches on the rising or falling edge (depending on the type of flip-flop) of the clock signal. In figure A-1 a rising edge triggered D flip-flop is shown, including its timing characteristics.

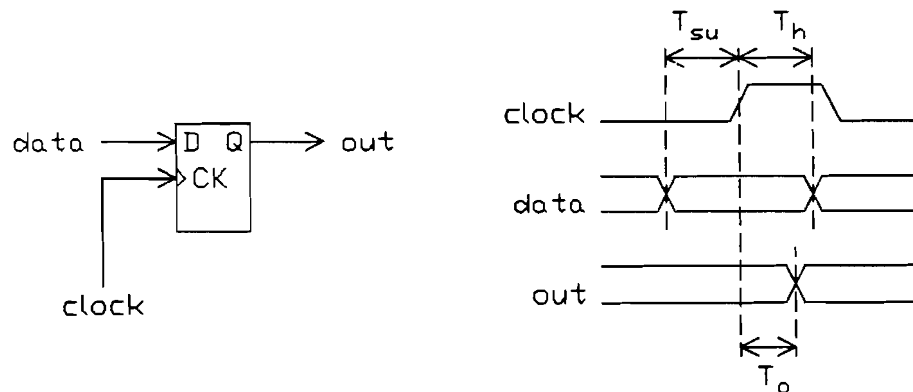


Figure A-1. rising edge triggered D flip-flop

T_{su} Data Set up time. The data signal must be stable T_{su} seconds before the rising edge of the clock signal appears.

T_h Data hold time. The data signal may not change for T_h seconds after the rising edge of the clock signal has appeared.

T_o Data output delay. The clocked data will be available at the output T_o seconds after the rising edge of the clock signal.

2) Transparent latches. This type of storage element transports the input to the output when the clock signal is active. The output cannot change if the clock signal is inactive. In figure A-2 a transparent latch is shown. The clock is high active, so the latch is in the 'see through' mode if the clock signal is high. The timing characteristics of the latch are also given.

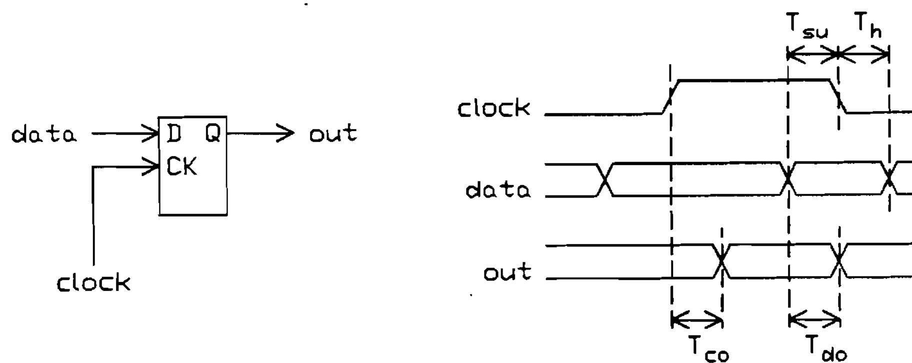


Figure A-2. Transparent latch

T_{su} Data Set up time. The data signal must be stable T_{su} seconds before the falling edge of the clock signal appears.

T_h Data hold time. The data signal may not change for T_h seconds after the falling edge of the clock signal has appeared.

T_{do} Data to output delay. The data output will be available at the output T_{do} seconds after the data input has changed, but only if the latch is in the 'see through' mode.

T_{co} Clock to output delay. The data output will be available at the output T_{co} seconds after the rising edge of the clock signal has appeared.

2. Sequential circuit characteristics

In this section the testability and switching characteristics of some sequential circuits will be analyzed. The results can be used to design state machines, and analyze its performances.

Consider a sequential circuit that contains two combinatorial circuits, named C1 and C2. The previously described types of storage element will

be used. Three possible circuit configurations will be discussed now. The delays of the combinatorial circuits C1 and C2 are resp. T_{c1} and T_{c2} seconds.

2.1. Circuits with edge triggered flip-flops

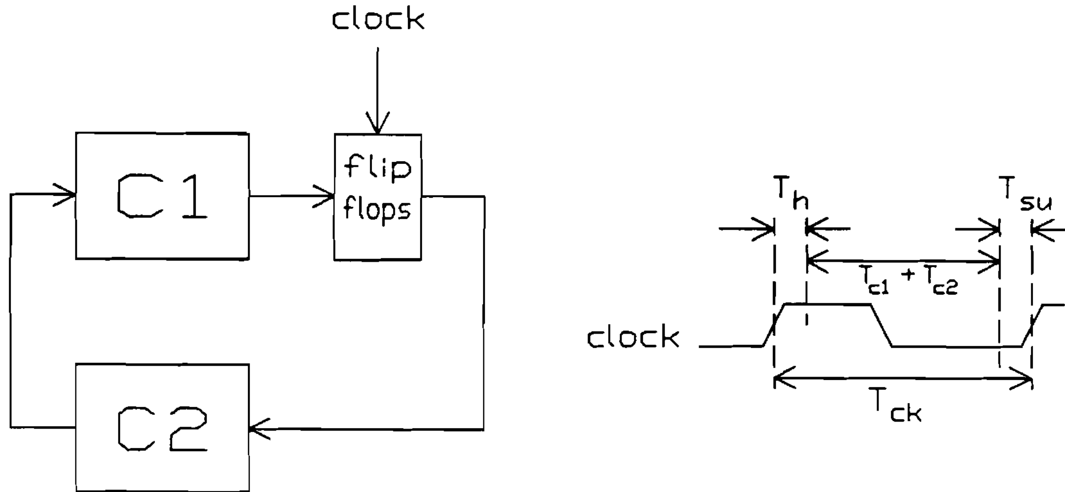


Figure A-3. A sequential circuit with positive edge triggered flip-flops

For the clock cycle time T_{ck} the next relation must hold.

$$T_{ck} >= T_{c1} + T_{c2} + T_{su} + T_h$$

An advantage of this circuit architecture is that a scan path can be created easily. A scan path can be used to test the (combinatorial) circuits between storage elements for production errors. The 'scan test method' is a technique that is applied often these days.

A disadvantage of this circuit architecture is that clock skewing can cause problems if the feedback of some signals is very fast. This is because the outputs of the flip-flops will be immediately available after the rising edge of the clock signal.

2.2. Circuits with Master Slave flip-flops

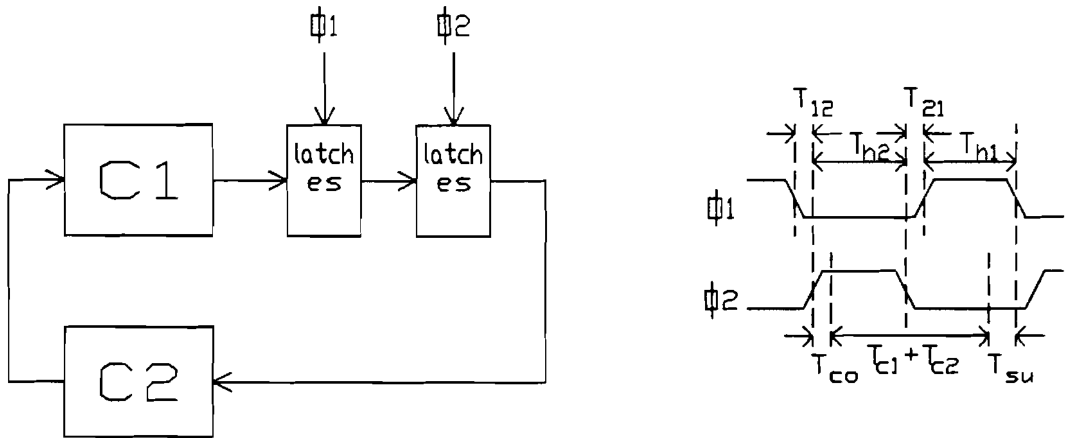


Figure A-4. A sequential circuit with Master Slave flip-flops, built with transparent latches

For the clock cycle time the next relation must hold.

$$T_{h2} + T_{21} + T_{h1} >= T_{co} + T_{c1} + T_{c2} + T_{su}$$

In this circuit architecture it is also very easy to create a scan path for testability reasons. Note that if $T_{12} = 0$, then the minimum clock cycle time is the same as in the architecture of figure A-3, assumed that the set up and hold times of the storage elements are equal. The problem of clock skewing is avoided in this architecture, since Master Slave flip-flops are used. The time intervals T_{12} and T_{21} will be used to avoid that, due to clock skewing, the two latches are both in the 'see through' mode for a particular time.

2.3. Circuits with latches

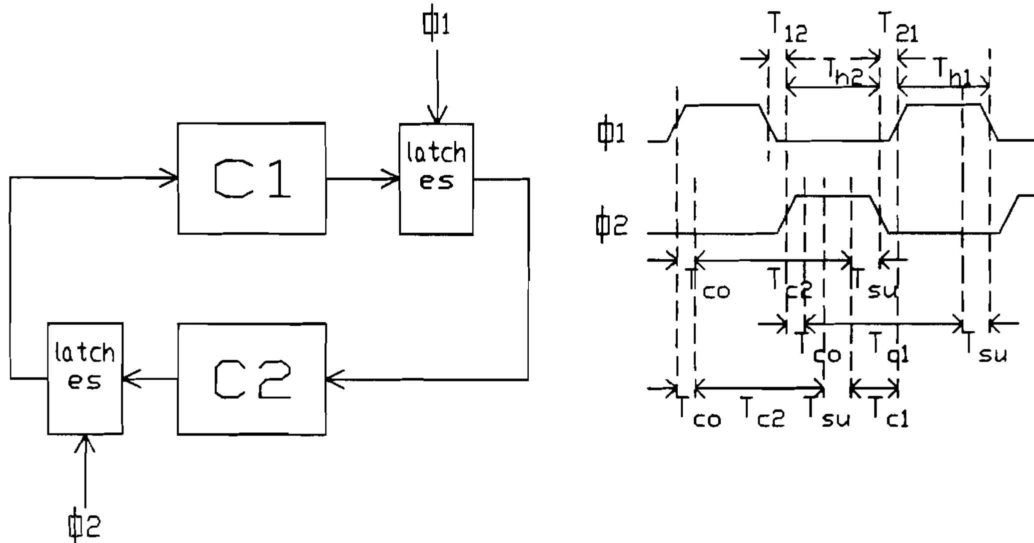


Figure A-5. A sequential circuit with transparent latches

For the clock cycle time the next relations must hold.

$$T_{h1} + T_{12} + T_{h2} >= T_{co} + T_{c2} + T_{su}$$

$$T_{h2} + T_{21} + T_{h1} >= T_{co} + T_{c1} + T_{su}$$

$$T_{h1} + T_{12} + T_{h2} + T_{21} >= T_{co} + T_{c2} + T_{do} + T_{c1}$$

In this circuit architecture it is hard to create a scan path, since the transparent latches are scattered. That is a very big disadvantage. An advantage of this architecture is the density in time of the power dissipation of the circuit. In modern technologies (CMOS) the circuit dissipates much power when it switches. Since the two combinatorial circuits don't switch at the same time, the power wires in the circuit can be smaller. This is specially important in IC technology. Another advantage of this architecture is that the complete clock cycle is used (refer to the final clock cycle relation). This is not the case in the architecture of figure A-4, if T_{12} is not equal to zero. So this architecture can be a little faster than the previous one.

3. Conclusion

Of course there are more types of circuit architectures. For example sequential circuits with three coupled combinatorial circuits can be realized with a three-phase clock signal. However, in most applications the previously described architectures are very usable.

APPENDIX B: Process Configurations

The hardware implementation of processes has been described in chapter 4 of this report. However, different kinds of state machines can be used to realise the processing and control units of a process. The connection between these units can be made in different ways. All these matters affect the testability and maximum clock frequency of the process implementation. In this appendix various process architectures will be discussed.

In the figures B-1 and B-2 the state machine representations of resp. the processing and control unit are repeated.

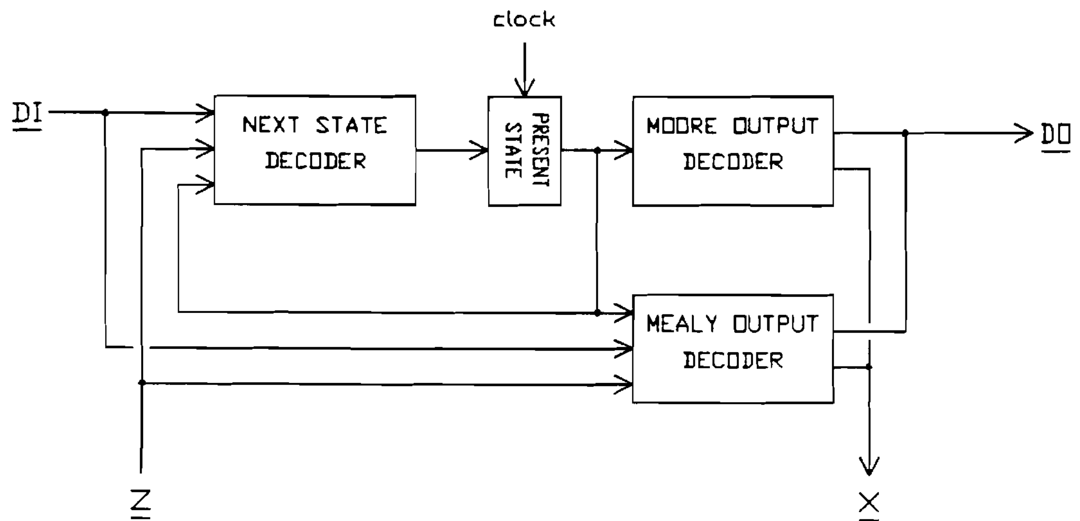


Figure B-1. State machine representation of a processing unit

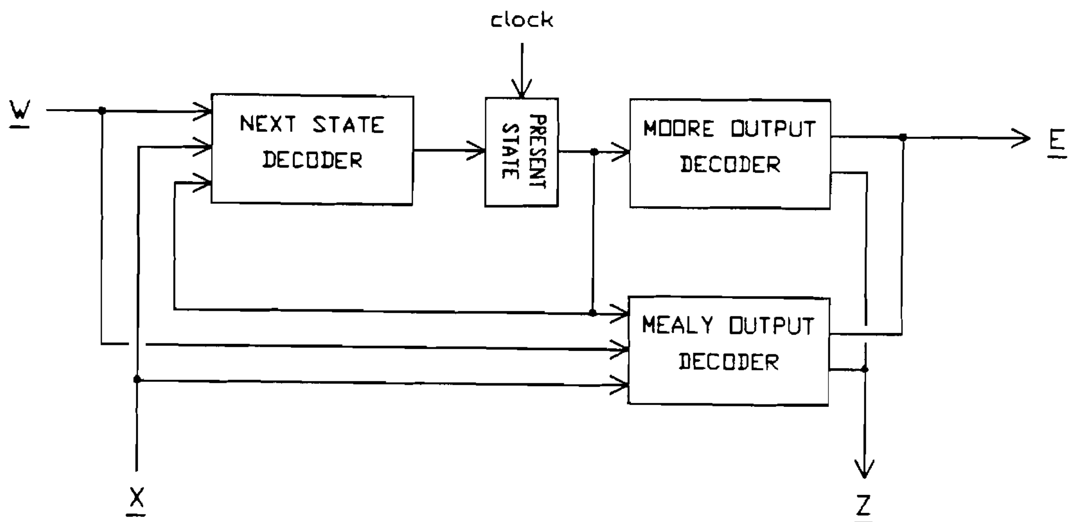


Figure B-2. State machine representation of a control unit

The signal names in these figures have the following meaning.

Z: The **control** signals which control the registers, computational elements, and interconnection elements in the processing unit will be considered to be a **vector**, named Z.

X: The **status** signals that are outputs of computational elements in the processing unit, will be considered to be a **vector**, named X.

DI: **Data input** of the processing unit. The data comes, via the data interconnection, from other processes or from the system environment (refer to figure 22 in this report).

DO: **Data output** of the processing unit. The data goes, via the data interconnection, to other processes or to the system environment (refer to figure 22 in this report).

E: The **control** signals which are connected with the Set and Reset inputs of the **event** flip-flops will be considered to be a **vector**, named E.

W: The **status** signals which are connected with the Q outputs of the **event** flip-flops will be considered to be a **vector**, named W.

The **data input** and **data output** vectors DI and DO can be interpreted in the following way.

- The part of DI that goes into the next state decoder (computational an interconnection elements) is received data. In general it is routed through interconnection elements to registers.

- The part of DI that goes into the Mealy output decoder (computational and interconnection elements) is data that is received and immediately send to another process, or it is received data that immediately generates status signals for the control-unit. Note that data can only be stored in the processing unit if it goes to the next state decoder.

- The part of DO that comes from the Moore output decoder (computational and interconnection elements) is data that is send to other processes. Since no control signals go into the Moore output decoder, this data is permanently available. So this type of data output cannot be applied in a shared bus environment.

- The part of DO that comes from the Mealy output decoder (computational and interconnection elements) is data that will be send to other processes under control of interconnection elements.

Refer to chapter 4 of this report for more details concerning this process decomposition.

1. Process architectures

Due to the danger of clock skewing, only circuit architectures with transparent latches, clocked by a two-phase non-overlapping clock signal, will be applied to realize the processing and control units of a process. These two units will be connected now. This is shown in figure B-3. Master Slave flip-flops are used as storage elements.

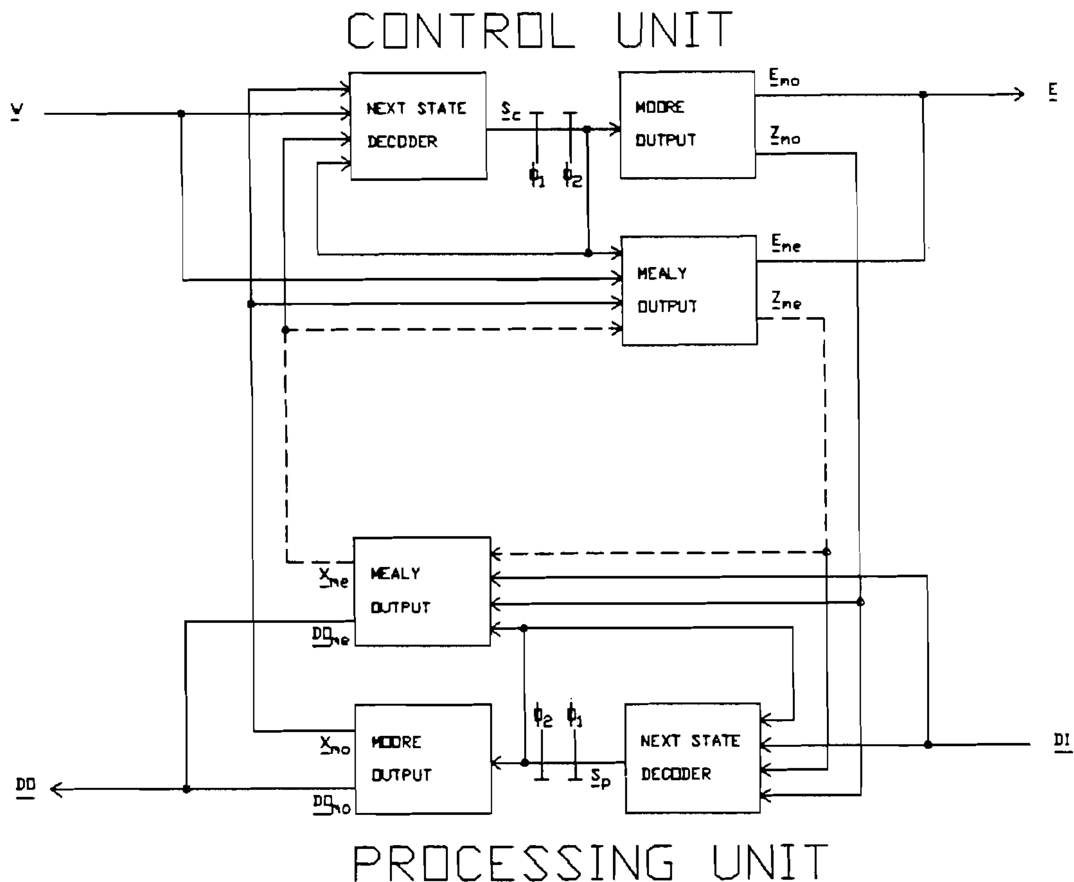


Figure B-3. The state machine representation of a process

In this figure the symbol

is used to represent a transparent latch. The output vectors from the Mealy output decoders have the index **me**, and those from the Moore output decoder have the index **mo**.

Unfortunately this compound state machine can be unstable since it contains a combinatorial loop. In figure B-3 this is indicated by the dashed lines (the signals $\underline{Z}_{me}$ and $\underline{X}_{me}$), which go through the two Mealy output decoders in the circuit. However, the system may operate correct in particular cases. This can be described mathematically.

Suppose $\underline{S}_c$ is the present state of the control unit, and $\underline{S}_p$ is the present state of the processing unit. The outputs of the Mealy output decoders are functions of its inputs:

$$\underline{Z}_{me} = \underline{Z}_{me}(\underline{S}_c, \underline{X}_{me}, \underline{X}_{mo}, \underline{W})$$

$$\underline{X}_{me} = \underline{X}_{me}(\underline{S}_p, \underline{Z}_{me}, \underline{Z}_{mo}, \underline{DI})$$

The following relation must hold for a stable system.

$$\left\{ \forall \underline{Z}_{me}, \forall \underline{S}_p, \forall x \in \underline{X}_{me} : \frac{\partial x(\underline{Z}_{me_0})}{\partial \underline{S}_p} \neq 0 \;\middle|\; \frac{\partial \underline{Z}_{me_0}}{\partial \underline{S}_p} = \underline{0} \right\}$$

In english it means:

> All elements x of $\underline{X}_{me}$ which can be affected by a particular $\underline{Z}_{me}$ may not affect this $\underline{Z}_{me}$. This must hold for all possible $\underline{Z}_{me}$ in the process description and in all possible states of the processing unit.

In general the system will not fulfil this condition. In that case the combinatorial loop must be eliminated by adding storage elements. There are four possibilities for placing these additional storage elements. This results in four different process architectures. In the figures B-4, B-5, B-6, and B-7 these architectures are shown. Later on the characteristics will be discussed.

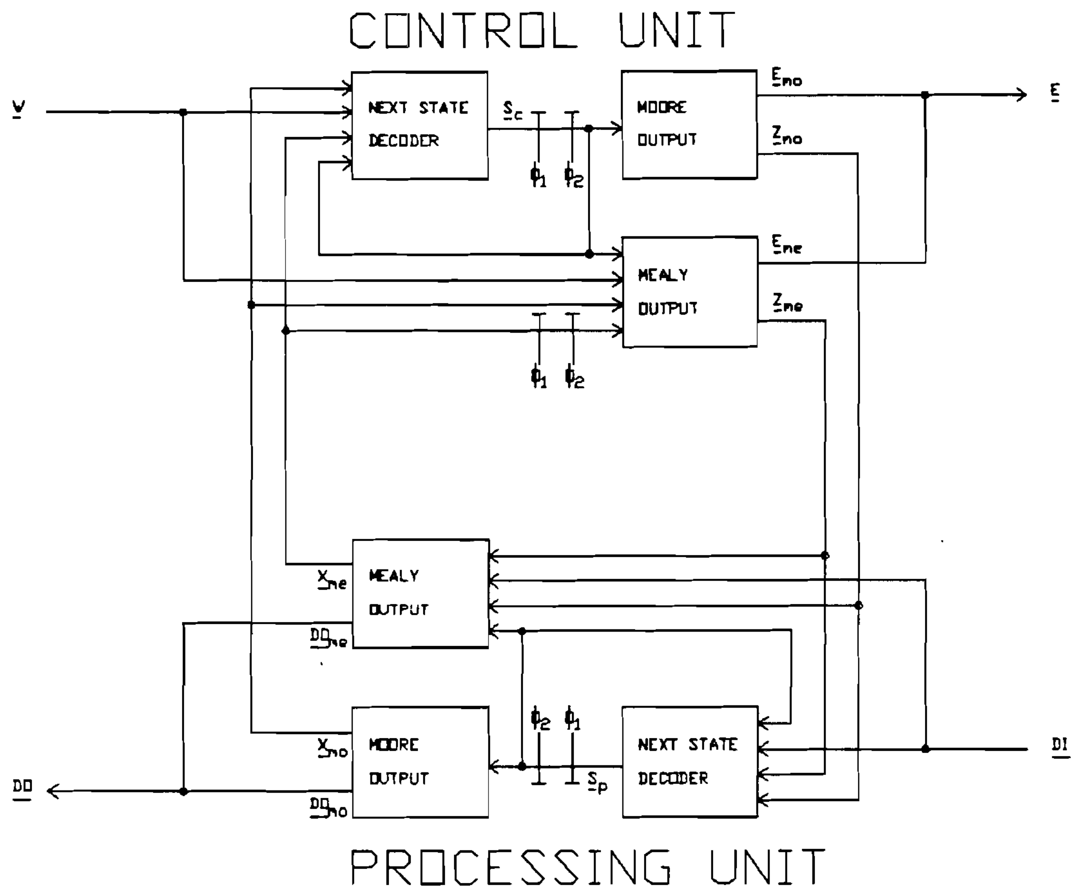


Figure B-4. Additional flip-flops only in the inputs of the Mealy output decoder of the control unit

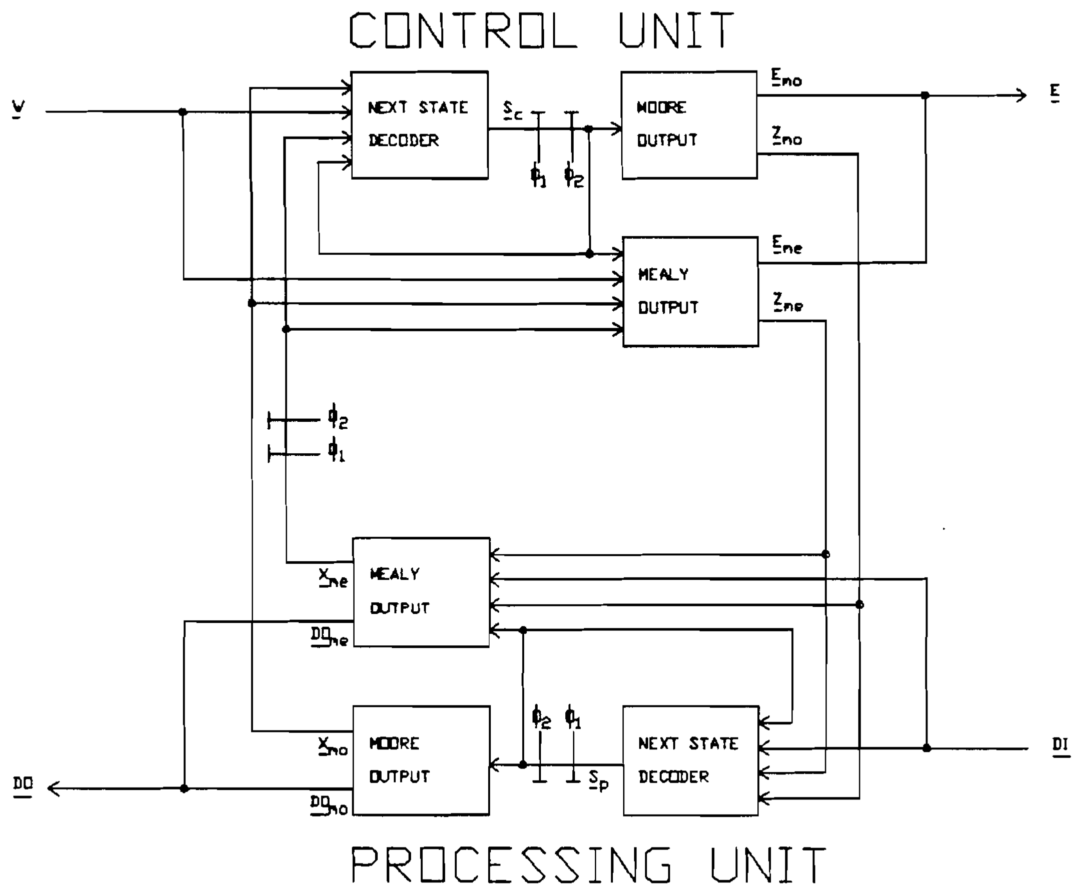


Figure B-5. Additional flip-flops in the Mealy outputs of the processing unit

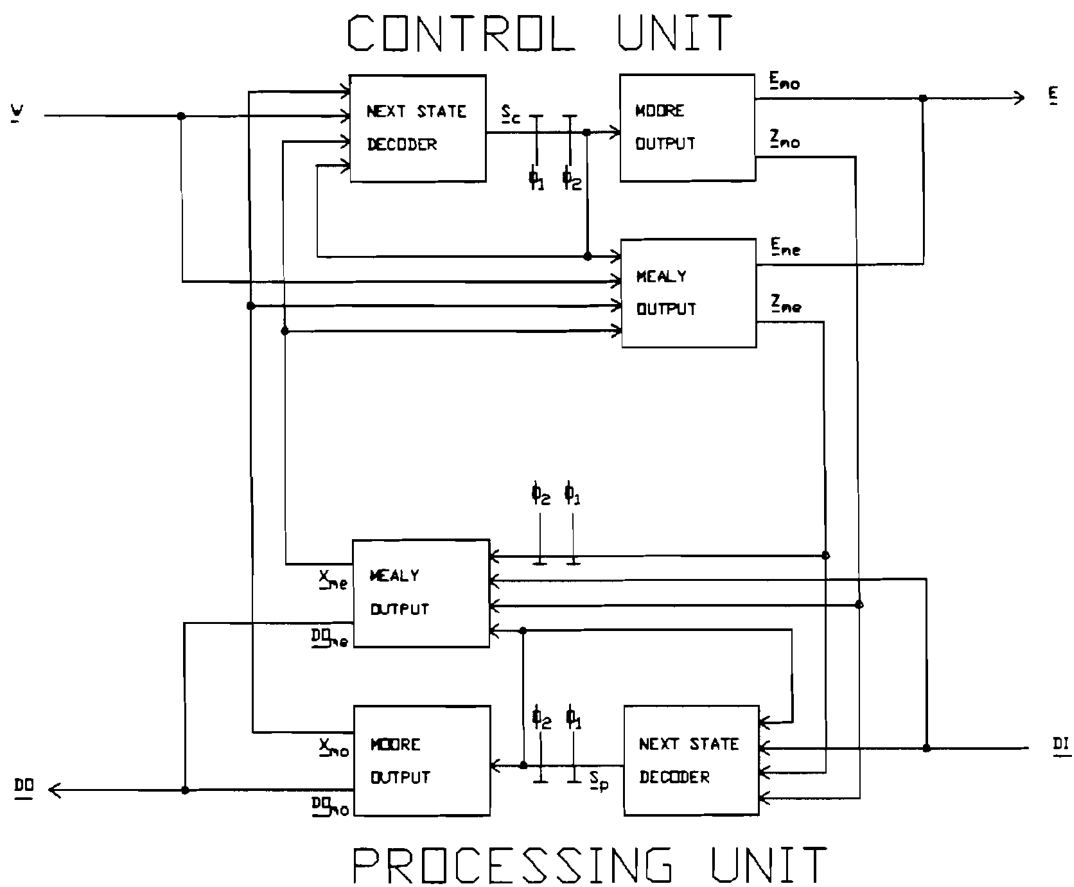


Figure B-6. Additional flip-flops only in the inputs of the Mealy output decoder of the processing unit

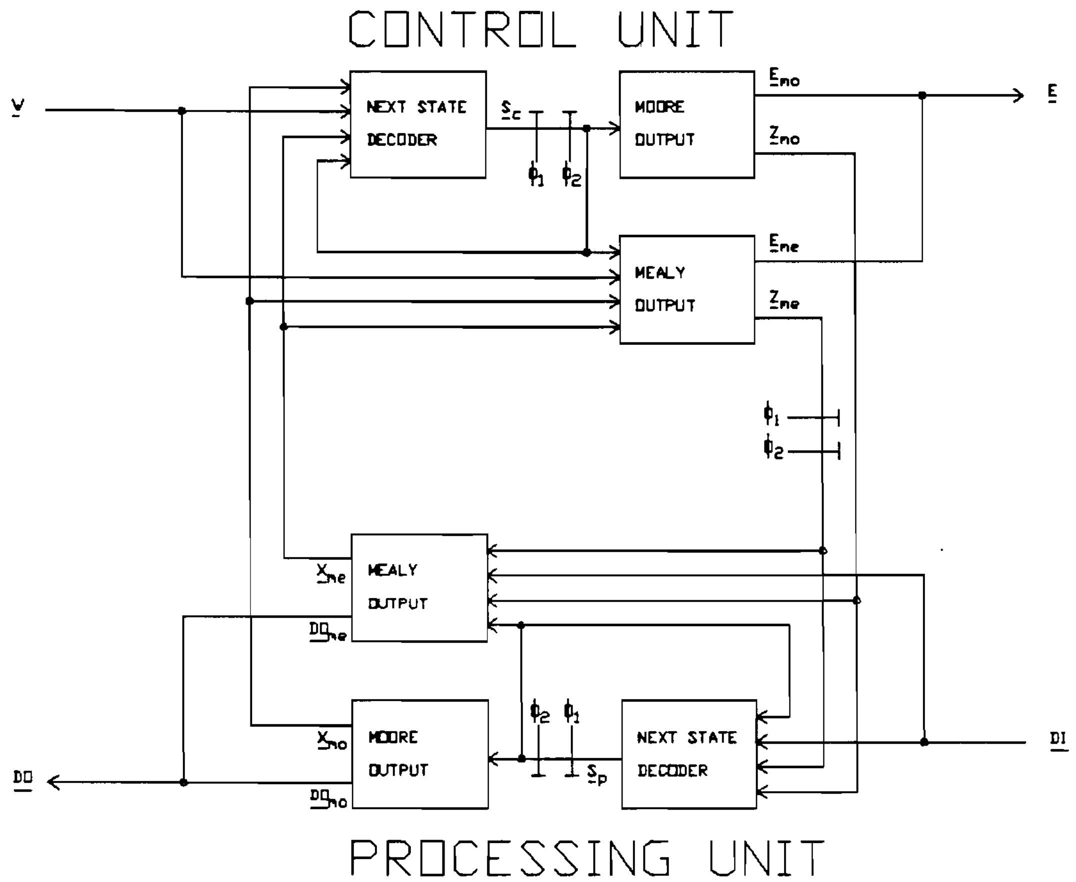


Figure B-7. Additional flip-flops in the Mealy outputs of the control unit

Note that the configuration of figure B-4 and B-6 is not always a physical possibility, since the next state and Mealy output decoder can be in one combinatorial circuit (for example an ALU). In that case the inputs to the Mealy output decoder cannot be isolated from the inputs to the next state decoder.

Besides these four architectures there are other possibilities:

- In practice it can be interesting to move the Moore output decoder of the processing- and/or control unit from the outputs of the present state storage elements to the inputs of these. For the processing unit, the resulting architecture is shown in figure B-8.

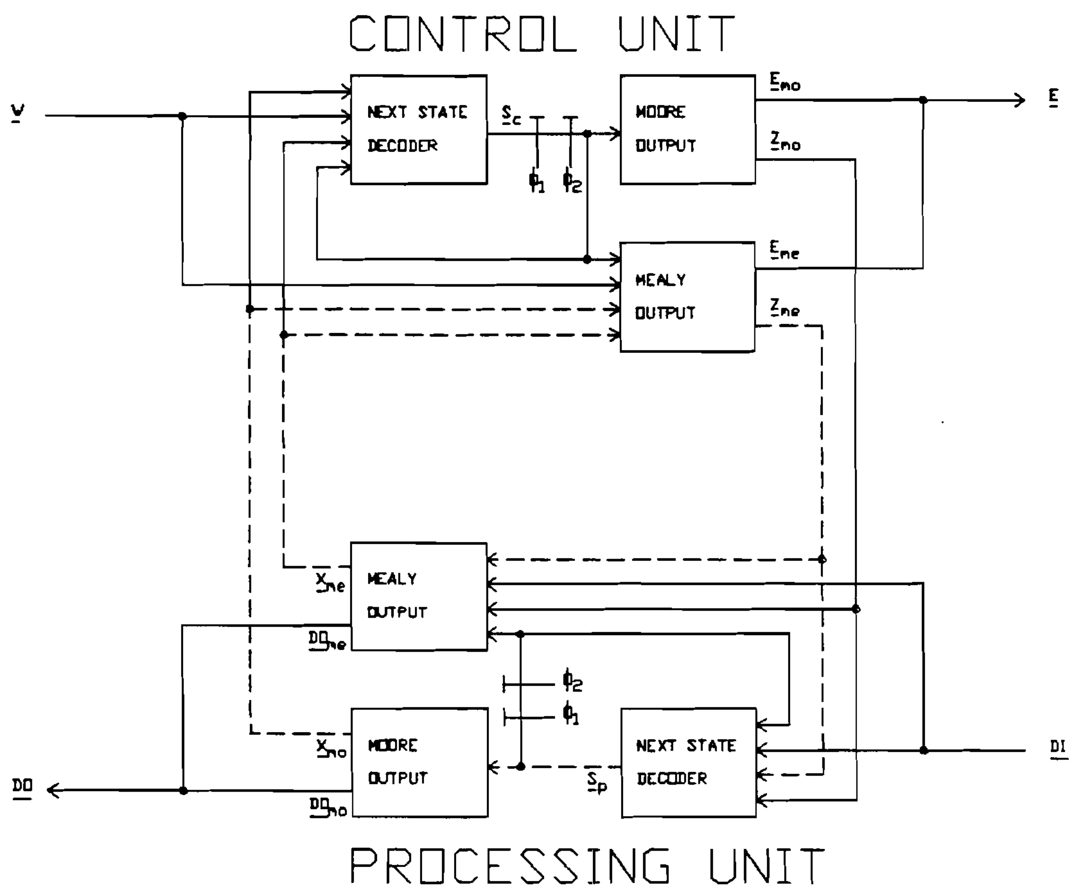


Figure B-8. Processing unit with displaced Moore output decoder

The dashed lines in this figure show more combinatorial loops than in the architecture of figure B-3. These loops must be eliminated to make the circuit stable. Theoretically the outputs of the Moore output decoder have become Mealy outputs, so this architecture shows the same behaviour as the architecture of figure B-3 if only

the Mealy outputs are considered. For this reason this architecture will not be analyzed further. A practical example of such a processing unit can be found in figure 27 of this report. The computational elements **C1** and **C2** will be connected to the inputs (instead of the outputs) of resp. the registers **B** and **A**. In that case there will be no difference between the status outputs x2 and x6 (zero flags). Both are Mealy outputs of the processing unit now.

- It is possible to add more storage elements than needed in the architecture of figure B-3. A reason for this can be to bound the combinatorial delays. The minimum clock cycle time depends on the longest combinatorial delay in the system. Another reason can be the observability and controllability of the combinatorial logic. If more storage elements will be added which can be linked into a scan path, then the combinatorial circuits are better observable and controllable, so better testable. In figure B-9 an example of such an architecture is shown. Both the complete **X** and **Z** vectors are latched.

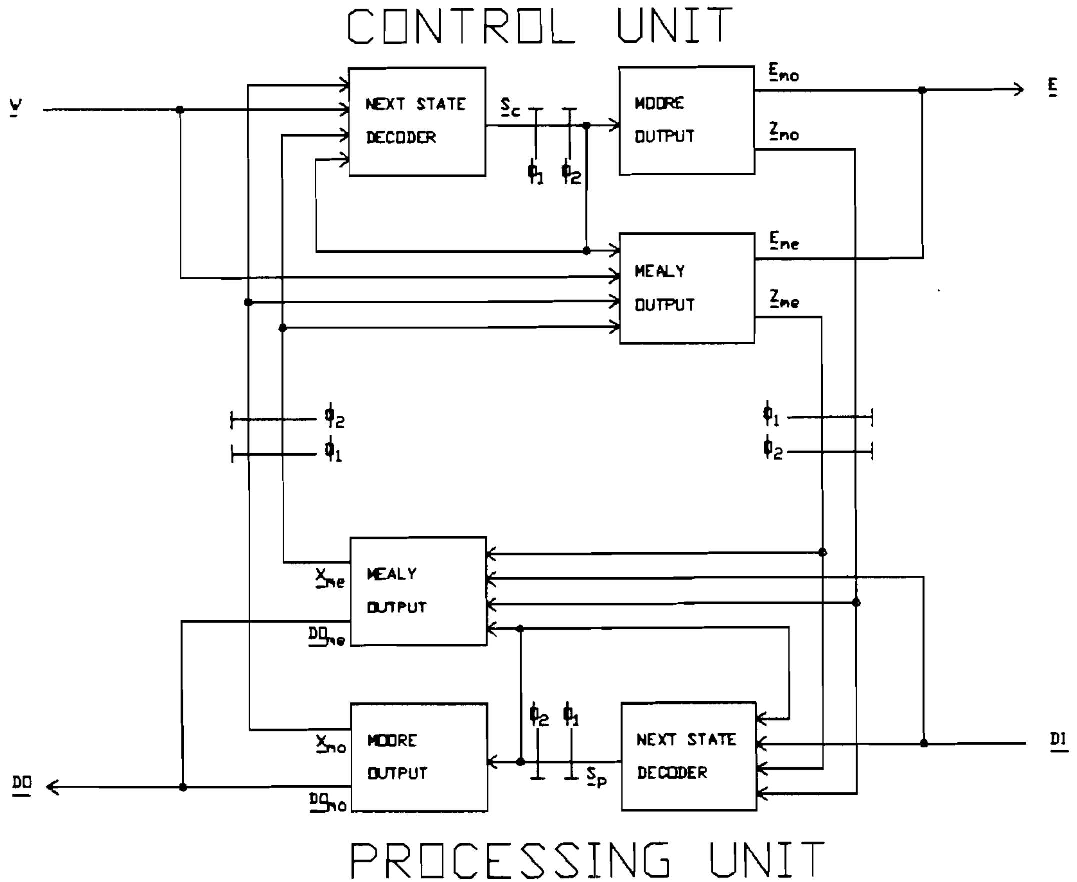


Figure B-9. Process architecture with short combinatorial paths

The characteristics of this architecture will be discussed later on.

- Referring to figure A-5, the latches can be placed in a different way, as shown in figure B-10.

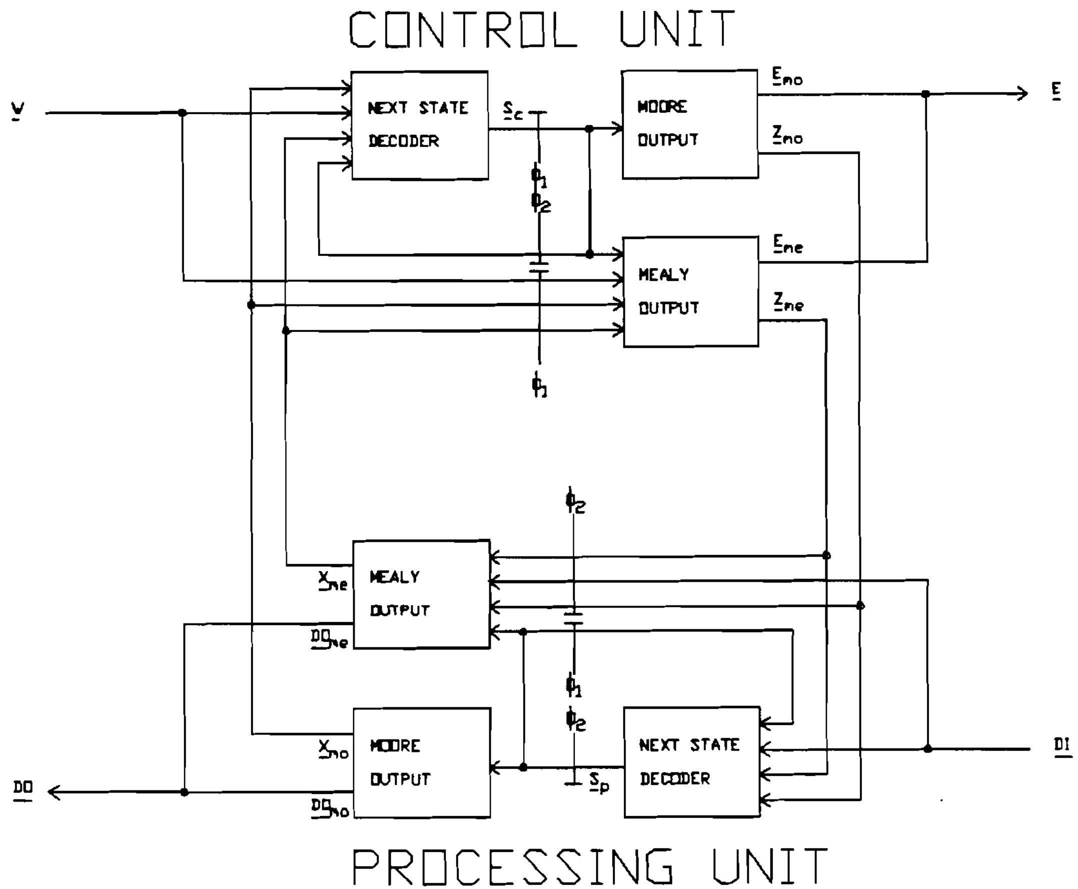


Figure B-10. Process architecture with separated transparent latches

In this architecture the control and processing unit are not switching in phase. In [DAVI] this type of architecture is used in many cases. For this reason its characteristics will be discussed later on.

In the next section the characteristics of the presented process architectures will be discussed.

2. Characteristics of process architectures

The testability and execution speed of the previously presented process architectures will be discussed now. To analyze the execution speed a sequence of information transformation and flow control statements will

be analyzed. The following Pascal like process description will be used for that.

```
PROCESS proc;
begin
  ....
  ....
  op_0(..);
  case condition of
     f1:  begin
              op_1(..);
              ....
          end;
     f2:  begin
              op_2(..);
              ....
          end;

     ....

     fn:  begin
              op_n(..);
              ....
          end;
  ....
  ....
end.
```

In this process description **op_0** to **op_n** are primitive operations. **f1** to **fn** are possible values of **condition**. The execution of **op_0** and the case statement will be analyzed. Assumed is that **op_0** affects **condition**.

The performance analyses deal with the minimum clock cycle, the number of clock cycles to perform an instruction, and the testability characteristics. In the next sections the six architectures of figure B-4, B-5, B-6, B-7, B-9, and B-10 will be analyzed.

Four different cases must be distinguished to analyze the execution speed:

1) The status vector, as a result of the first operation **op_0**, comes from the Mealy output decoder of the processing unit. The control vectors that represent the operations **op_1** to **op_n** come from the Mealy output decoder of the control unit.

2) The status vector, as a result of the first operation **op_0**, comes from the Moore output decoder of the processing unit. The control vectors that represent the operations **op_1** to **op_n** come from the Mealy output decoder of the control unit.

3) The status vector, as a result of the first operation **op_0**, comes from the Mealy output decoder of the processing unit. The control vectors that represent the operations **op_1** to **op_n** come from the Moore output decoder of the control unit.

4) The status vector, as a result of the first operation **op_0**, comes from the Moore output decoder of the processing unit. The control vectors that represent the operations **op_1** to **op_n** come from the Moore output decoder of the control unit.

The machine states and signal vectors will be labeled according to the following definitions.

1) The control vector $\underline{Z}_i$ represents operation **op_i**.

2) The states of the control unit will be named N_{index}. In state N_{i-1} the operation **op_0** will be performed. In state N_i the case statement will be executed. The states N_{i+1} to N_{i+n} are the next states after executing the first operation **op_i** in the case statement.

3) All combinatorial circuits have delay. The value of these delays are as follows (time unit is second).

Next state decoder control unit : **Tcs**
Mealy output decoder control unit : **Tce**
Moore output decoder control unit : **Tco**
Next state decoder processing unit: **Tps**
Mealy output decoder processing unit: **Tpe**
Moore output decoder processing unit: **Tpo**

2.1. Architecture 1

The first architecture is the one of figure B-4.

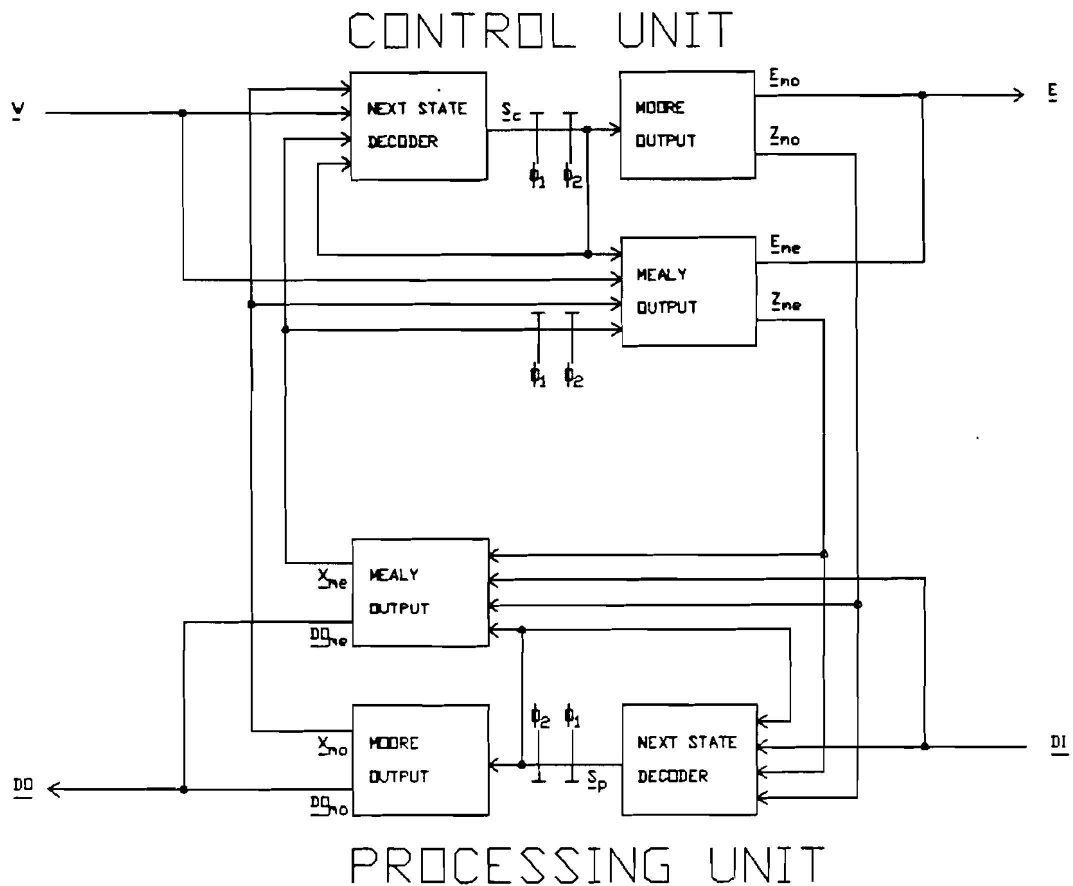


Figure B-4. Process architecture 1

The minimum duration of clock cycle **Tck** depends on the maximum delay of the cascade of combinatorial circuits.

$$
\begin{aligned}
Tck = \max\{ & Tco + Tps, \\
& Tco + Tpe + Tcs, \\
& Tpo + Tce + Tps, \\
& Tpo + Tce + Tpe + Tcs\}
\end{aligned}
$$

The longest combinatorial path contains four combinatorial circuits. This does not mean this path has also the maximum delay.

The Master Slave flip-flops can easily be linked into a scan path, so this architecture is scan testable. However, the controllability and observability of the combinatorial circuits is not optimal since the longest

combinatorial path contains four combinatorial circuits. So it can be hard to find test vectors for all possible faults.

The four different cases for analyzing the execution speed will lead to four different ASM charts for the control unit, as shown in figure B-11. These ASM charts indicate how many clock cycles are needed to perform the operations.

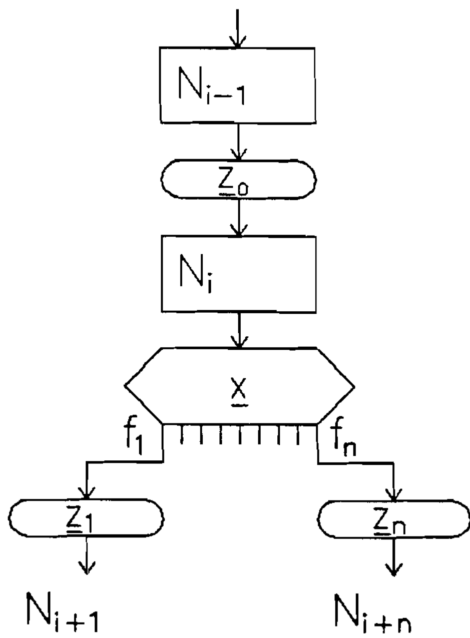
Remarks:

- This architecture can only be realized if the inputs to the next state decoder and to the Mealy output decoder in the control unit can be separated from each other.

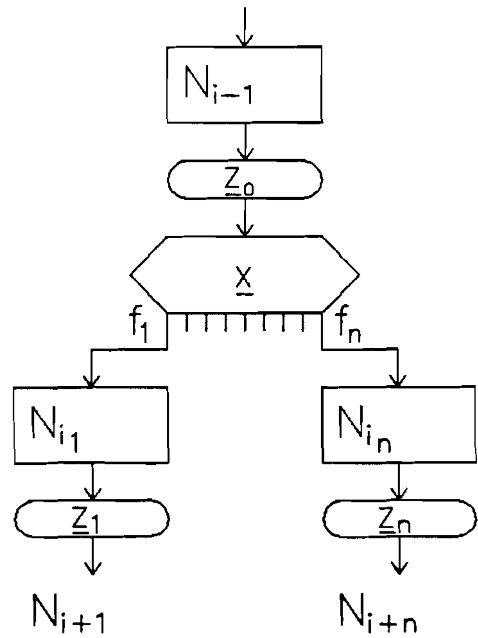
- In the ASM charts it can be seen easily that the control units in the first two cases are Mealy machines (output vectors depend on $\underline{\mathbf{X}}$) and in the last two cases Moore machines (output vectors independent of $\underline{\mathbf{X}}$).

- The ASM charts of the first two cases are identical since $\underline{\mathbf{X}}_{me}$ is latched at the input of the Mealy output decoder of the control unit, so for this decoder the vectors $\underline{\mathbf{X}}_{me}$ and $\underline{\mathbf{X}}_{mo}$ are available at the same time. For the next state decoder of the control unit these vectors are not available at the same time. For this reason the third and fourth cases result in different ASM charts.

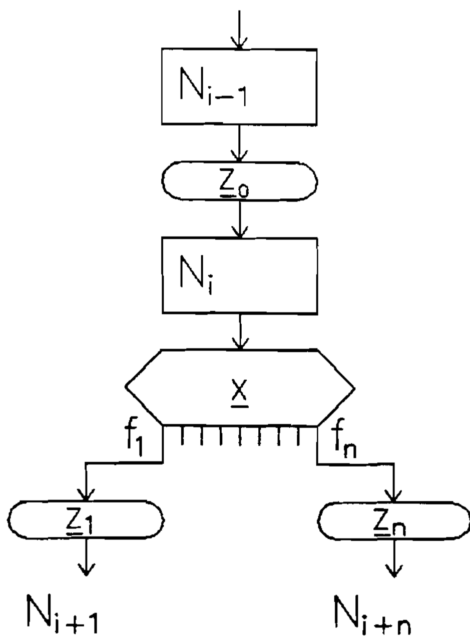
- In the first three cases one operation can be executed per clock cycle. The fourth case is different. An evaluation state in the control unit is necessary to wait for the status of the processing unit. This is because the response $\underline{\mathbf{X}}_{mo}$ to the control vector $\underline{\mathbf{Z}}_{mo}$ has to pass the latches in the processing unit. This takes an extra clock cycle. This is also the case in the second case, but in that case the status vector $\underline{\mathbf{X}}_{mo}$ does not have to pass the latches in the control unit. It goes directly into the Mealy output decoder.



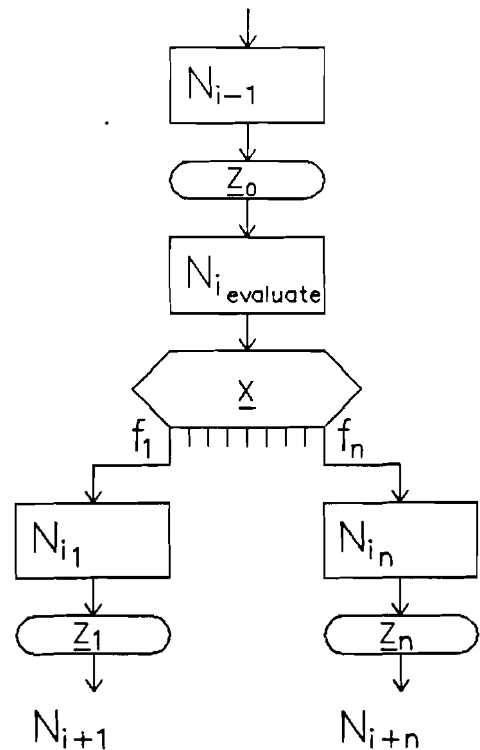
case 1: Mealy/Mealy



case 2: Moore/Mealy



case 3: Mealy/Moore



case 4: Moore/Moore

Figure B-11. ASM charts of architecture 1

The last remark is very interesting. It is possible to analyze how many clock cycles it takes until the response $\underline{X}$ to an operation $\underline{Z}$ is available in the output decoder of the control unit to generate the next control vector. A theorem can be formulated to handle this.

Theorem:

> The signal flow of the response must be traced from the output of the output decoder to the input of the same decoder in the control unit. The number of latches which are passed in this trace must be counted. This is the number of clock cycles that are needed to execute one operation. If this number is greater than one, then additional wait states must be created in the control unit. This is only necessary when the status of the processing unit has to be evaluated. If not, then every clock cycle a control vector can be generated to execute an operation.

To explain this theorem, the response $\underline{X}$ will be traced in the four different cases.

1) $\underline{Z}_{me}$ goes from the Mealy output decoder of the control unit through the Mealy output decoder of the processing unit, then through latches back to the input of the Mealy output decoder in the control unit. One latch has been passed, so one operation per clock cycle can be executed.

2) $\underline{Z}_{me}$ goes from the Mealy output decoder of the control unit through the next state decoder of the processing unit, then through latches, then through the Moore output decoder of the processing unit, and then back to the input of the Mealy output decoder of the control unit. One latch has been passed, so one operation per clock cycle can be executed.

3) $\underline{Z}_{mo}$ goes from the Moore output decoder of the control unit through the Mealy output decoder of the processing unit, then through the next state decoder of the control unit, then through latches back to the input of the Moore output decoder of the control unit. One latch has been passed, so one operation per clock cycle can be executed.

4) $\underline{Z}_{mo}$ goes from the Moore output decoder of the control unit through the next state decoder of the processing unit, then through latches, then through the Moore output decoder of the processing unit, then through the next state decoder of the control unit, and then through latches back to the input of the Moore output decoder of the control unit. This time two latches are passed, so one operation per two clock cycles can be executed. This means that one wait state must be inserted.

2.2. Architecture 2

The second architecture is the one of figure B-5.

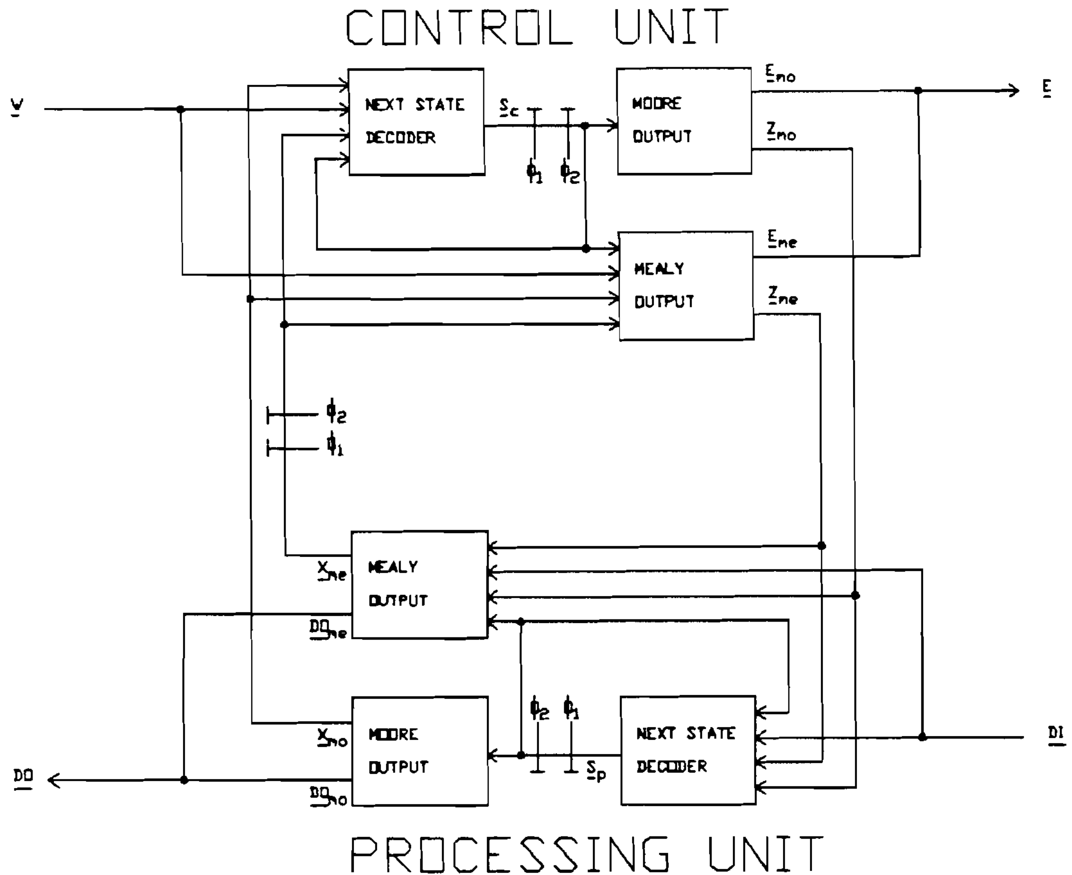


Figure B-5. System architecture 2

The minimum duration of clock cycle Tck depends on the maximum delay of the cascade of combinatorial circuits.

$$
\begin{aligned}
Tck = \max\{ & Tco + Tps, \\
& Tco + Tpe, \\
& Tpo + Tcs, \\
& Tpo + Tce + Tpe, \\
& Tpo + Tce + Tps \ \}
\end{aligned}
$$

The longest combinatorial path contains three combinatorial circuits. This does not mean that this path has also the maximum delay.

The Master Slave flip-flops can easily be linked into a scan path, so this architecture is scan testable. The controllability and observability of the combinatorial circuits is better than in the first architecture since the

longest combinatorial path contains less combinatorial circuits. So it can be easier to find test vectors for all possible faults, although it can still be hard.

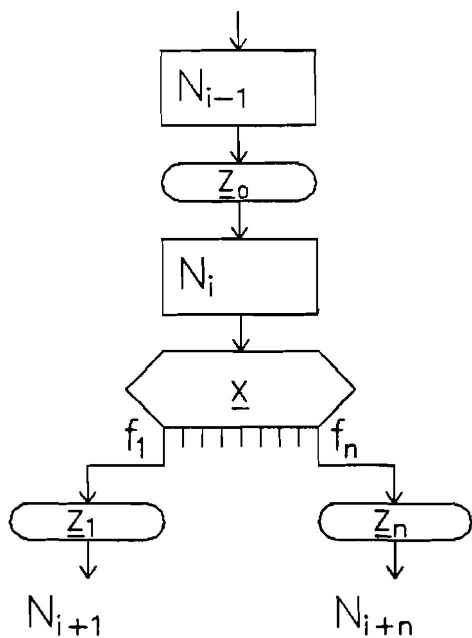
The four different cases for analyzing the execution speed will lead to four different ASM charts for the control unit, as shown in figure B-12. These ASM charts indicate how many clock cycles are needed to perform the operations.

Remarks:

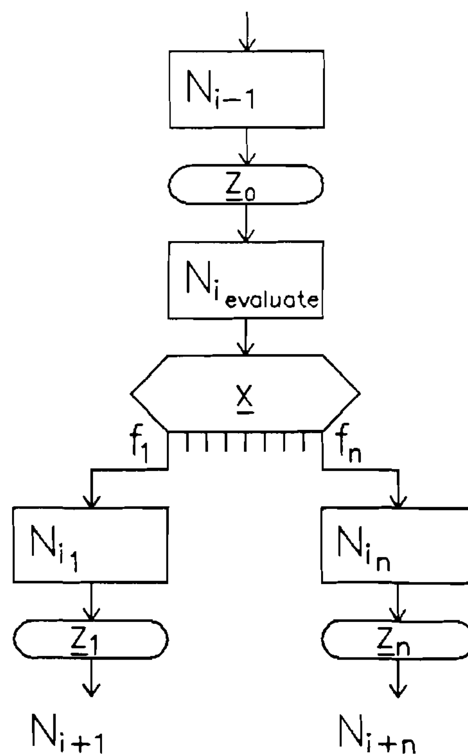
- The ASM charts do not depend on the use of $\underline{X}_{me}$ or $\underline{X}_{mo}$ since $\underline{X}_{me}$ is latched, so the two vectors are available at the same time.

- In the ASM charts it can be seen easily that the control units in the first two cases are Mealy machines (output vectors depend on $\underline{X}$) and in the last two cases Moore machines (output vectors independent on $\underline{X}$).

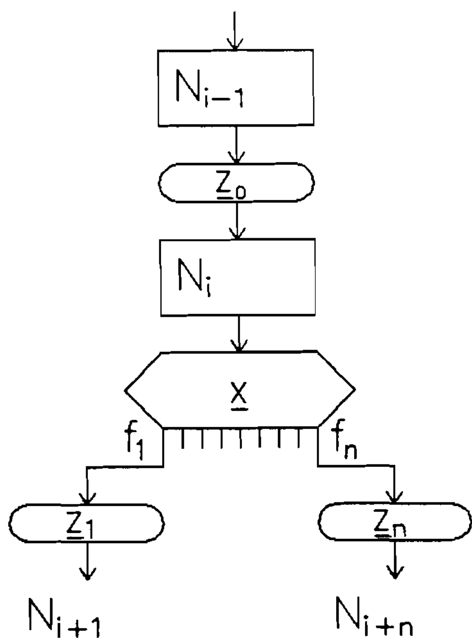
- In the first two cases one operation per clock cycle can be executed. The last two cases are different. An evaluation state in the control unit is necessary to wait for the status of the processing unit. This is because the response $\underline{X}_{mo}$ to the control vector $\underline{Z}_{mo}$ has to pass the latches in the processing unit, and $\underline{X}_{me}$ has to pass the additional latches. This takes an extra clock cycle. This can be checked by tracing the signal flow of the response and applying the previously defined theorem.



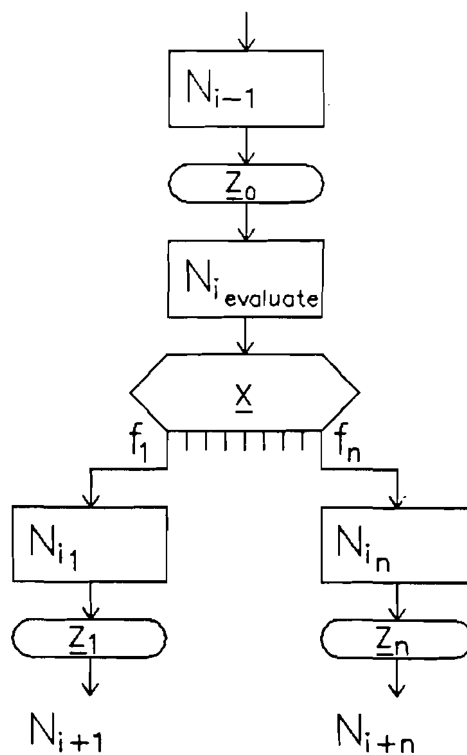
case 1: Mealy/Mealy



case 2: Moore/Mealy



case 3: Mealy/Moore



case 4: Moore/Moore

Figure B-12. ASM charts of architecture 2

2.3. Architecture 3

The third architecture is the one of figure B-6.

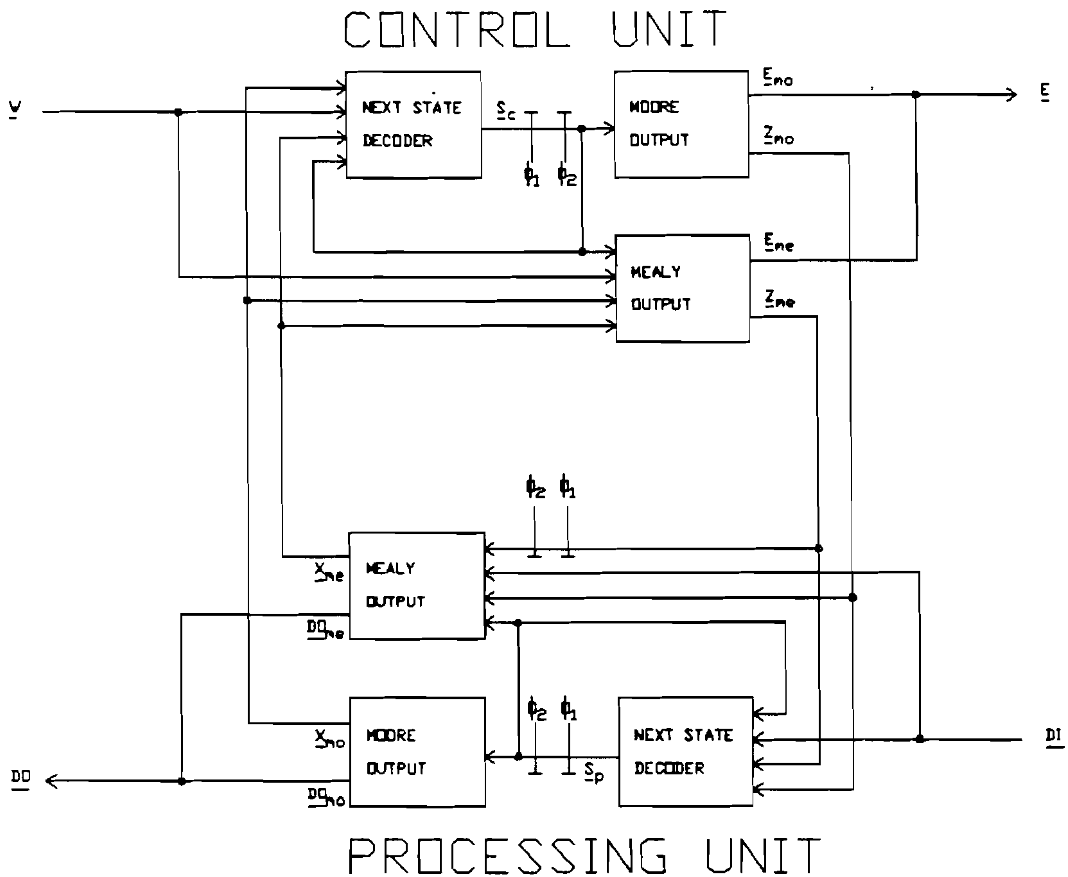


Figure B-6. Process architecture 3

The minimum duration of clock cycle Tck depends on the maximum delay of the cascade of combinatorial circuits.

$$
\begin{aligned}
Tck = \max\{ & Tpo + Tcs, \\
& Tco + Tpe + Tcs, \\
& Tpo + Tce + Tps, \\
& Tco + Tpe + Tce + Tps\}
\end{aligned}
$$

The longest combinatorial path contains four combinatorial circuits. This does not mean that this path has also the maximum delay.

The Master Slave flip-flops can easily be linked into a scan path, so this architecture is scan testable. The controllability and observability of the combinatorial circuits is as good as in the first architecture since the longest combinatorial path contains also four combinatorial circuits. So it

can be hard to find test vectors for all possible faults.

The four different cases for analyzing the execution speed will lead to four different ASM charts for the control unit, as shown in figure B-13. These ASM charts indicate how many clock cycles are needed to perform the operations.

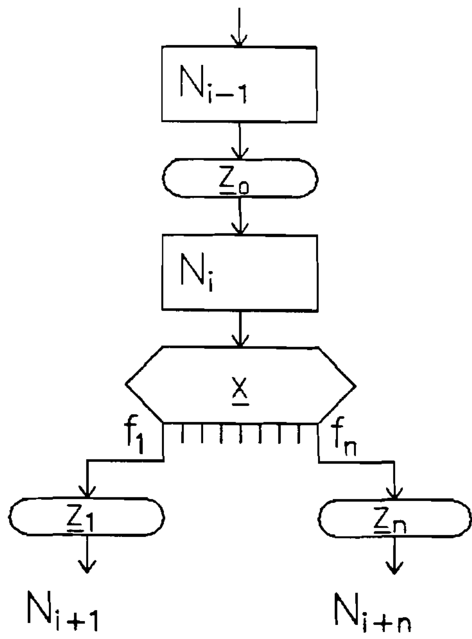
Remarks:

- This architecture can only be realized if the inputs to the next state decoder and to the Mealy output decoder in the processing unit can be separated from each other.

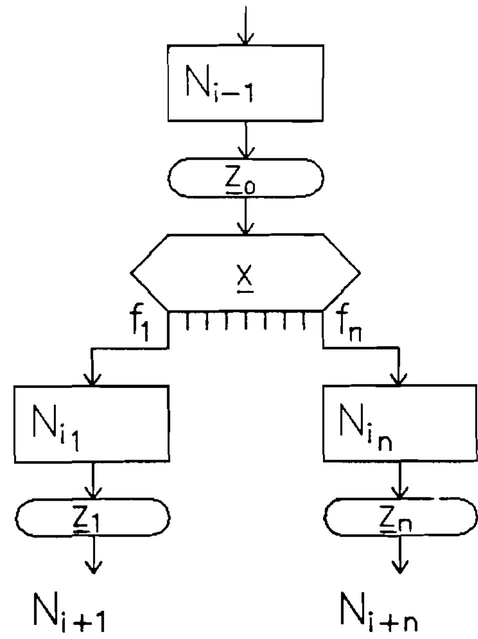
- In the ASM charts it can be seen easily that the control units in the first two cases are Mealy machines (output vectors depend on $\underline{X}$) and in the last two cases Moore machines (output vectors independent on $\underline{X}$).

- The control vector $\underline{Z}_{me}$ that goes to the Mealy output decoder is delayed for one clock cycle. The status vector $\underline{X}_{me}$ will be calculated by using the delayed control vector and the result of this operation in the processing unit which is already clocked at that time.

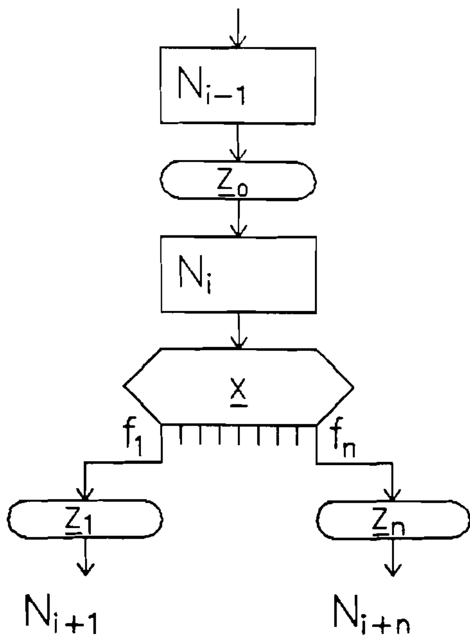
- Applying the previously defined theorem results in the conclusion that only in the fourth situation an additional wait state must be added. In the other situations one operation per clock cycle can be executed.



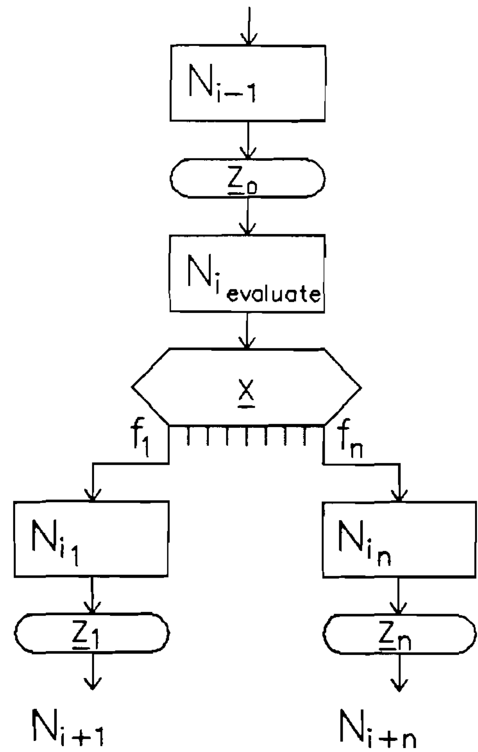
case 1: Mealy/Mealy



case 2: Moore/Mealy



case 3: Mealy/Moore



case 4: Moore/Moore

Figure B-13. ASM charts of architecture 3

2.4. Architecture 4

The fourth architecture is the one of figure B-7.

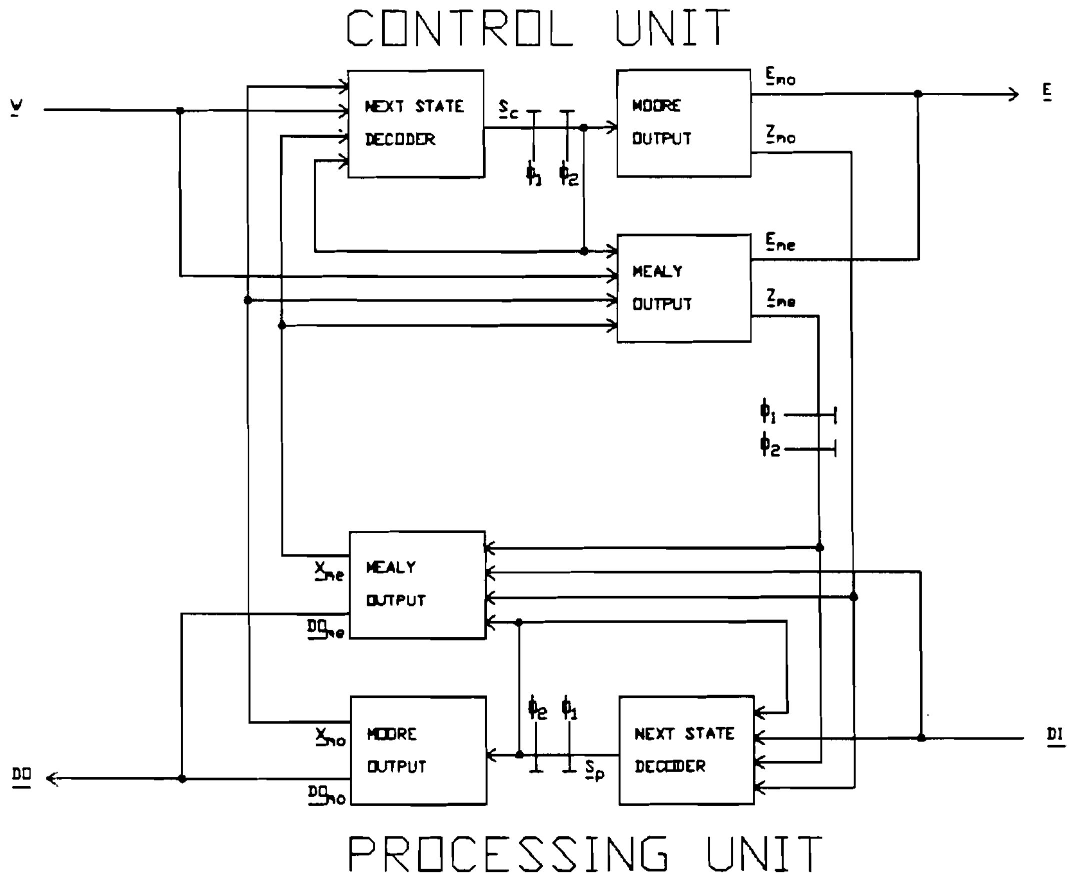


Figure B-7. Process architecture 4

The minimum duration of clock cycle **Tck** depends on the maximum delay of the cascade of combinatorial circuits.

$$Tck = \max\{Tco + Tps, \; Tpo + Tcs, \; Tpo + Tce, \; Tco + Tpe + Tcs, \; Tco + Tpe + Tce\}$$

The longest combinatorial path contains three combinatorial circuits. This does not mean that this path has also the maximum delay.

The Master Slave flip-flops can easily be linked into a scan path, so this architecture is scan testable. The controllability and observability of the combinatorial circuits is as good as in the second architecture since the

longest combinatorial path contains also three combinatorial circuits. So it can be hard to find test vectors for all possible faults, but it might be easier than in the first and third architecture.

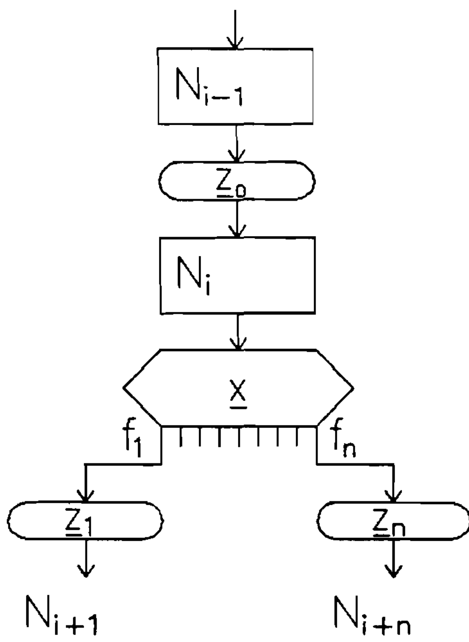
The four different cases for analyzing the execution speed will lead to four different ASM charts for the control unit, as shown in figure B-14. These ASM charts indicate how many clock cycles are needed to perform the operations.

Remarks:

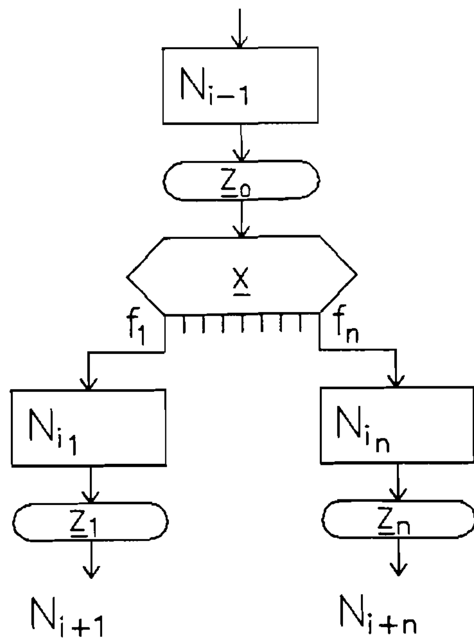
- With respect to the processing unit, there is no difference between the control vectors $\underline{Z}_{me}$ and $\underline{Z}_{mo}$ since both will be available at the same time. This is due to the additional latches.

- In the ASM charts it can be seen easily that the control units in the first two cases are Mealy machines (output vectors depend on $\underline{X}$) and in the last two cases Moore machines (output vectors independent on $\underline{X}$).

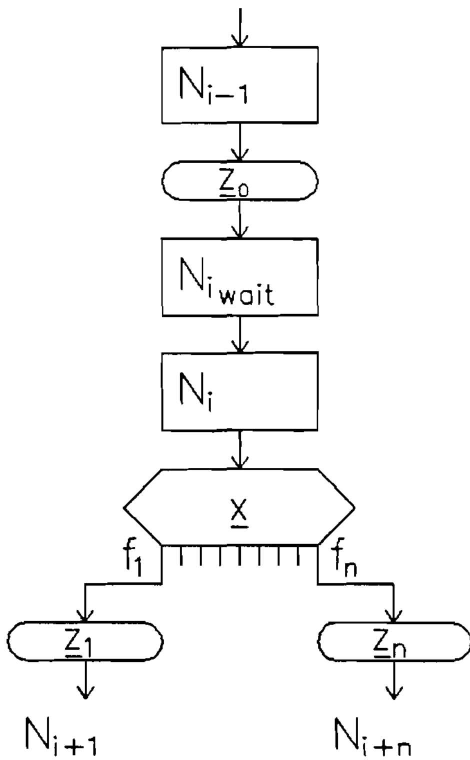
- Applying the previously defined theorem results in the conclusion that in the second and the fourth situation an additional wait state must be added. In the other situations one operation per clock cycle can be executed.



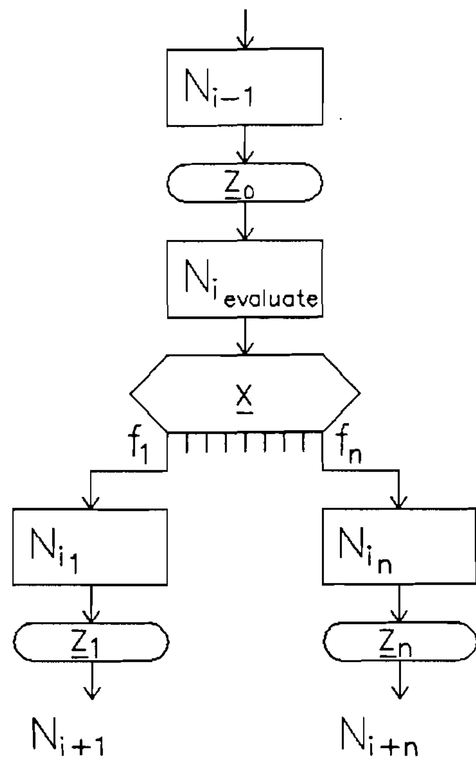
case 1: Mealy/Mealy



case 2: Moore/Mealy



case 3: Mealy/Moore



case 4: Moore/Moore

Figure B-14. ASM charts of architecture 4

2.5. Architecture 5

The fifth architecture is the one of figure B-9.

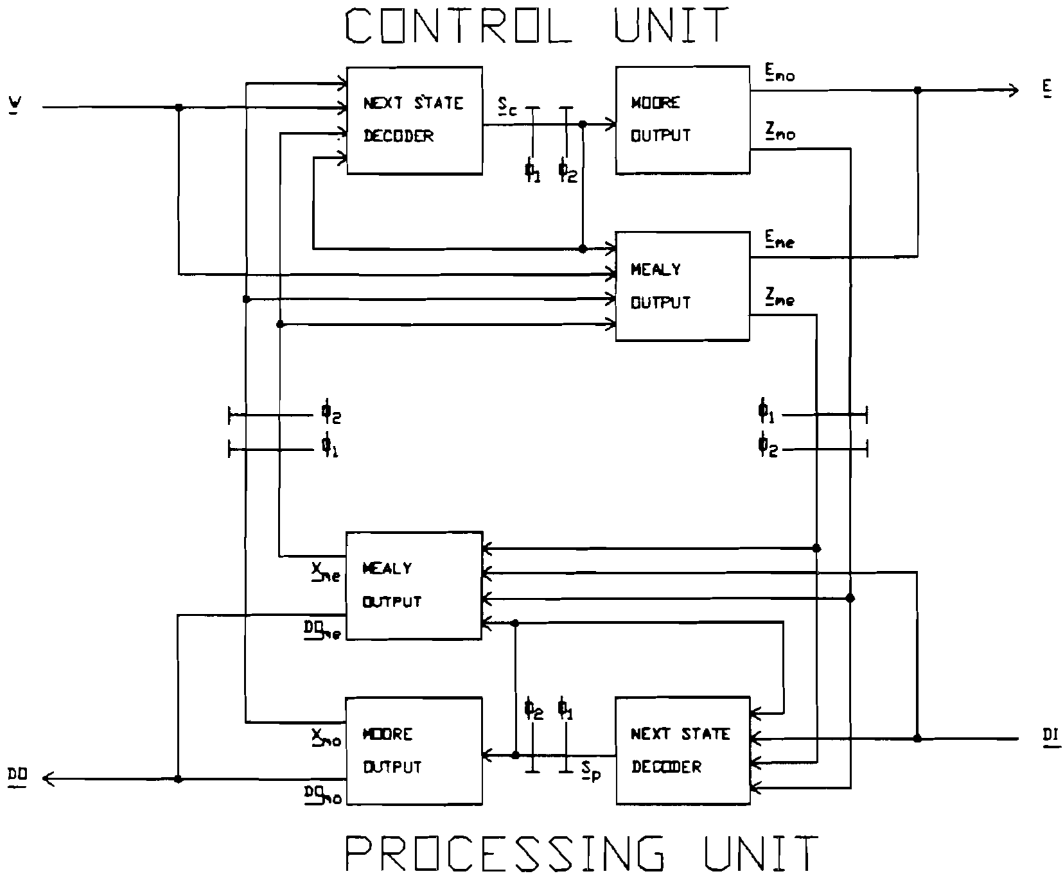


Figure B-9. Process architecture 5

The minimum duration of clock cycle Tck depends on the maximum delay in the combinatorial circuits.

Tck = max{Tcs, Tco, Tce, Tps, Tpo, Tpe}

The longest combinatorial path contains only one combinatorial circuits. This characteristic differs strongly from the previous architectures. It can be stated that this architecture can operate with the smallest clock cycle time.

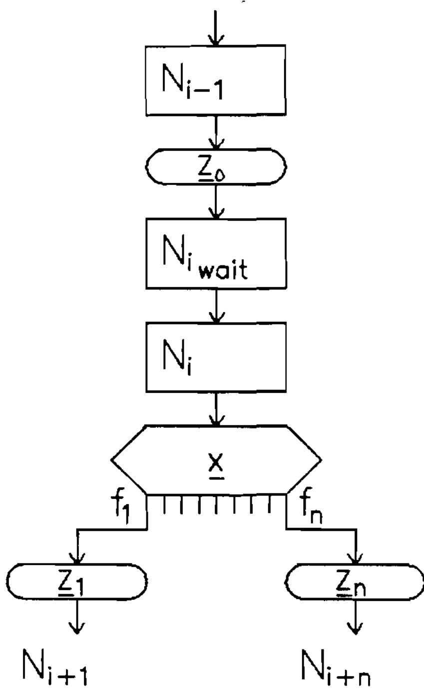
The Master Slave flip-flops can easily be linked into a scan path, so this architecture is scan testable. The controllability and observability of the combinatorial circuits is very good, since every combinatorial circuit can be isolated. This means test vectors for all possible faults may be much easier to find than in the previous four architectures.

The four different cases for analyzing the execution speed will lead to four different ASM charts for the control unit, as shown in figure B-15. These ASM charts indicate how many clock cycles are needed to perform the operations.

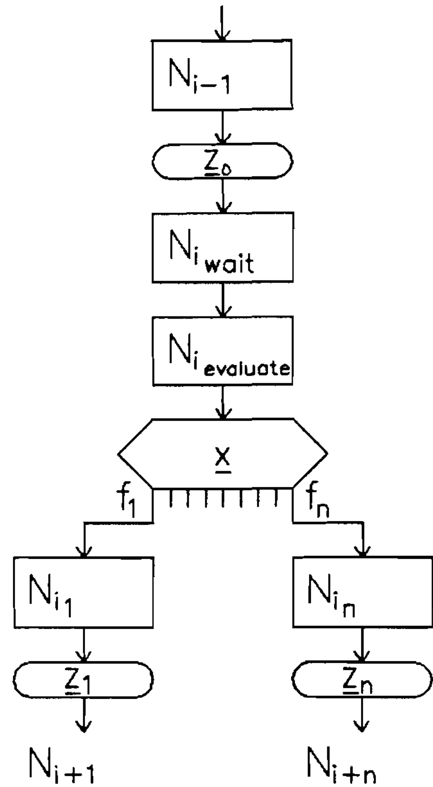
Remarks:

- In the ASM charts it can be seen easily that the control units in the first two cases are Mealy machines (output vectors depend on $\underline{X}$) and in the last two cases Moore machines (output vectors independent on $\underline{X}$).

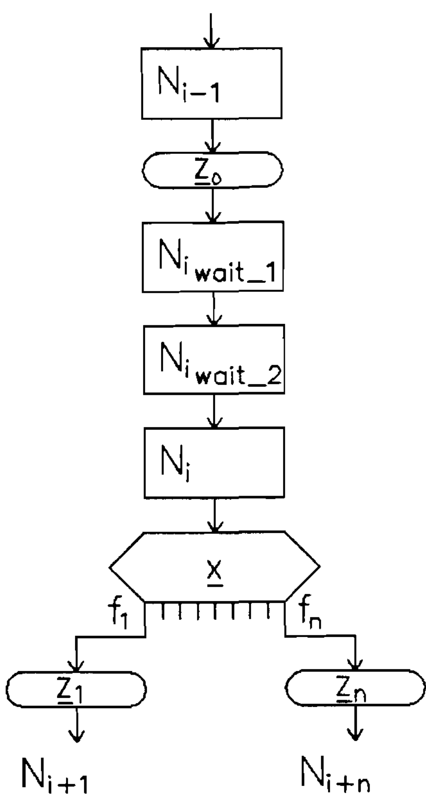
- Applying the previously defined theorem results in the conclusion that in all situations additional wait states must be added. In the first situation one operation per two clock cycles can be executed. In the second and third cases three clock cycles are needed to perform one operation, and in the fourth case this takes even four clock cycles. This is the price that has to be paid for getting a short clock cycle time.



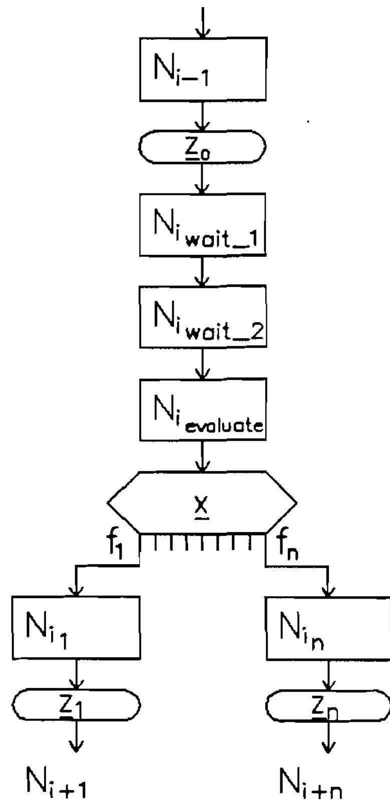
case 1: Mealy/Mealy



case 2: Moore/Mealy



case 3: Mealy/Moore



case 4: Moore/Moore

Figure B-15. ASM charts of architecture 5

2.6. Architecture 6

The sixth and final architecture is the one of figure B-10.

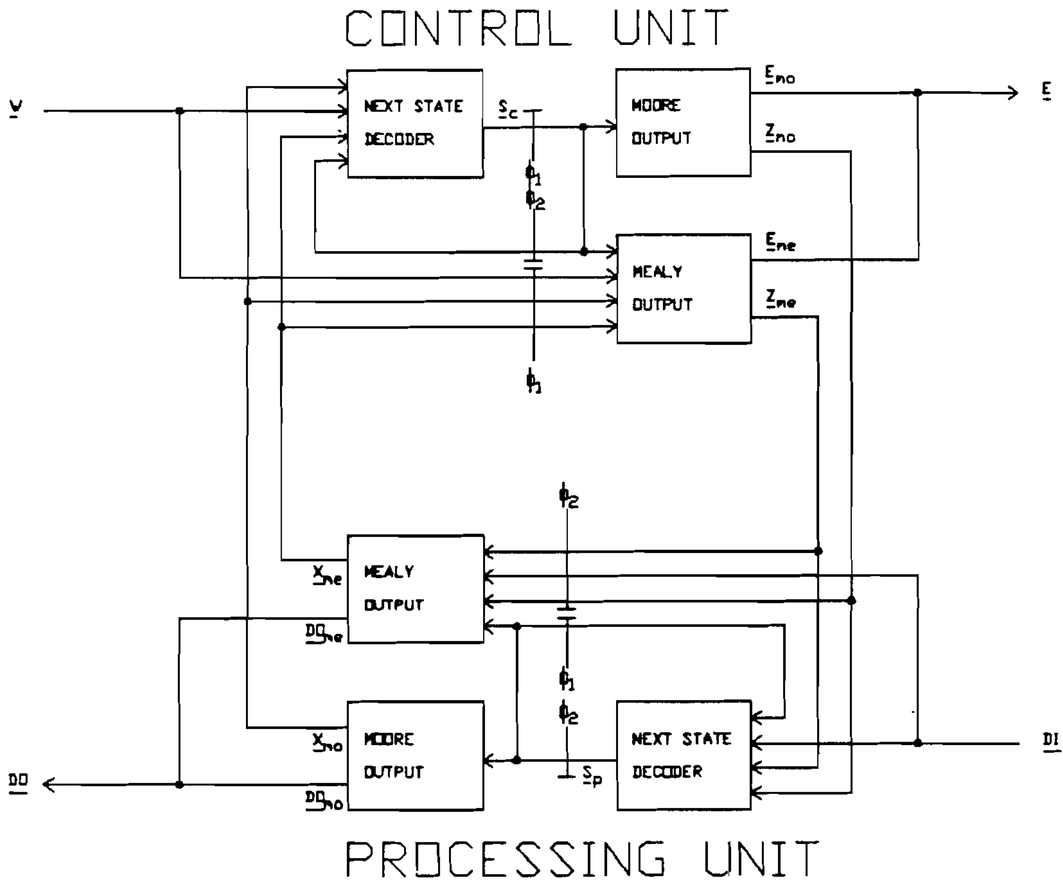


Figure B-10. Process architecture 6

The minimum duration of clock cycle Tck depends on the maximum delay of the cascade of combinatorial circuits.

$$
\begin{aligned}
Tck = \max\{ & Tco + Tps + Tpo + Tcs, \\
& Tco + Tps + Tpe + Tcs, \\
& Tce + Tcs + Tpo + Tcs, \\
& Tce + Tcs + Tpe + Tcs\}
\end{aligned}
$$

The longest combinatorial path contains four combinatorial circuits. Although a system with this architecture has less combinatorial logic between latches, the clock cycle time is certainly not shorter than the previously described architectures.

The latches cannot be linked into a scan path, so this architecture is not scan testable in this way. Additional latches must be placed to be able to

create a scan path. This doubles the number of latches in the system, so that would be an expensive solution.

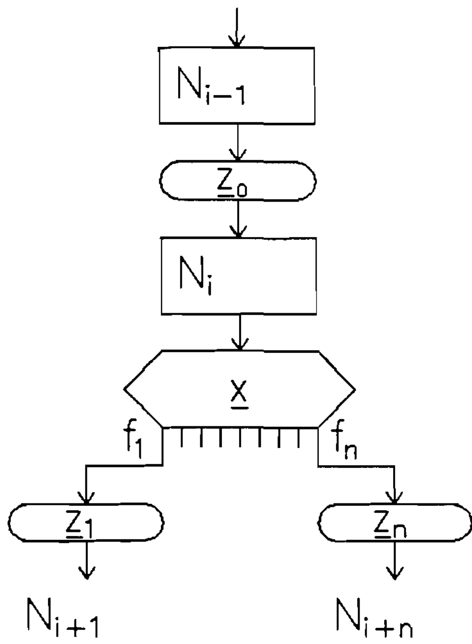
The four different cases for analyzing the execution speed will lead to four different ASM charts for the control unit, as shown in figure B-16. These ASM charts indicate how many clock cycles are needed to perform the operations.

Remarks:

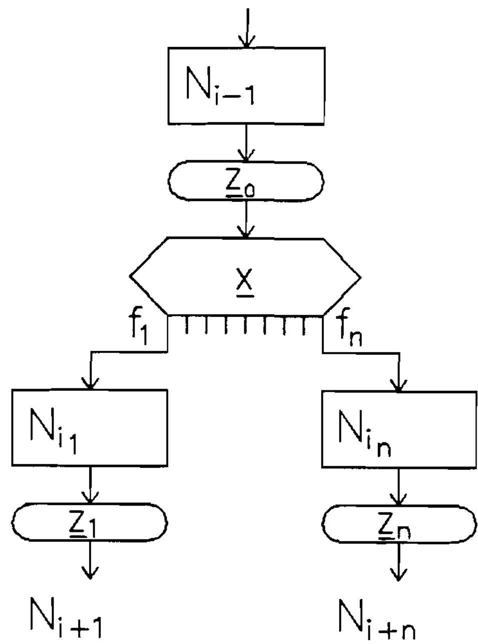
- This architecture can only be realized if the inputs to the next state decoder and to the Mealy output decoder in both the processing and the control unit can be separated from each other.

- In the ASM charts it can be seen easily that the control units in the first two cases are Mealy machines (output vectors depend on $\underline{X}$) and in the last two cases Moore machines (output vectors independent on $\underline{X}$).

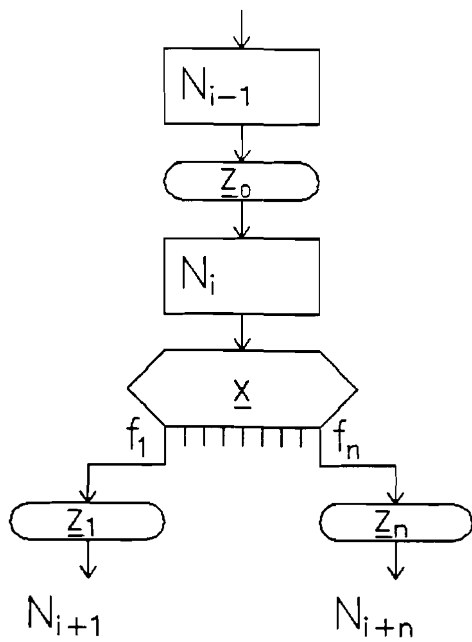
- Applying the previously defined theorem results in the conclusion that in none of the situations an additional wait state has to be added. So this is the only architecture that can execute one operation every clock cycle, even in the fourth situation.



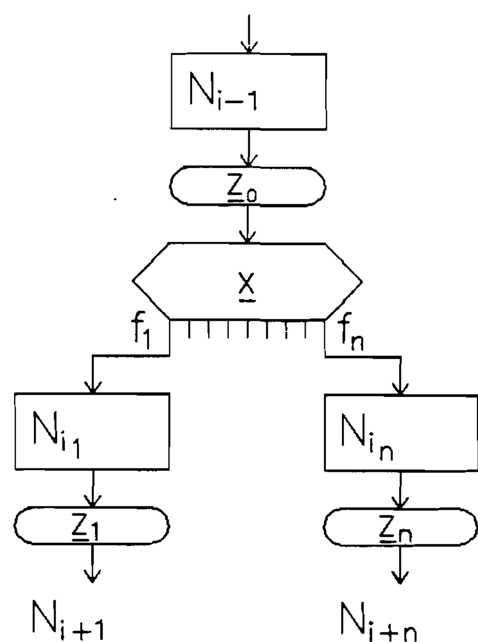
case 1: Mealy/Mealy



case 2: Moore/Mealy



case 3: Mealy/Moore



case 4: Moore/Moore

Figure B-16. ASM charts of architecture 6

3. Conclusions

Some recommendations can be given to implement processes in hardware. These are derived from the analyzed process architectures.

- The architecture of figure B-10 is a bad choice because it is not scan testable without doubling the number of latches.

- If the process description, that must be implemented in hardware, contains many evaluations of conditions, then it is preferable to have less wait states in the control units. One of the process architectures of figure B-4, B-5, B-6, or B-7 will be a good choice.

- If the process description does not contain many evaluations of conditions, then it is preferable to have a short clock cycle time. This can be realized by having short combinatorial paths. Applying the process architecture of figure B-9 can lead to a satisfying performance.

APPENDIX C: Event Management Configurations

In chapter 4 of this report the hardware implementation of the **event management** has been discussed (refer to figure 22 of this report). The control units of processes are connected with **event flip-flops** to implement the **signal** and **wait** statements, so they interact indirectly. Since control units can be Mealy or Moore machines, control units and event management can be connected in various ways. The consequences for testability and execution speed of processes have been analyzed in appendix B. In this section the execution speed of the synchronization and communication statements will be discussed.

1. Description of interacting processes

To analyze the implementation and execution of synchronization and communication statements, the sender and receiver processes of section 3.2.2 of chapter 3 of this report will be used. The Pascal like notation of this system is as follows.

```
PROCESS comm_proc;
cobegin
  proc_i;
  proc_j;
coend.

PROCESS proc_i; {sending process}
begin
  {internal actions}
  signal(trx_rdy); {request for transmitting}
  wait(rec_rdy);   {wait for receiver ready}
  send(proc_j,data); {send data to proc_j}
  {internal actions}
end.

PROCESS proc_j; {receiving process}
begin
  {internal actions}
  signal(rec_rdy); {request for receiving}
  wait(trx_rdy);   {wait for transmitter ready}
  receive(proc_i,data); {receive data from proc_i}
  {internal actions}
end.
```

In figure C-1 the corresponding Petri Net of the process interactions is shown.

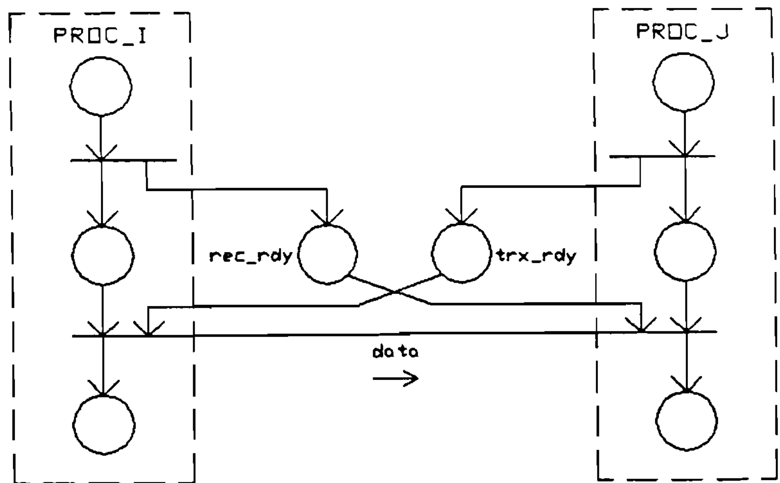


Figure C-1. Petri Net graph representation of the interactions of **proc_i** and **proc_j**

The implementation and execution of the statements **wait**, **signal**, **send**, and **receive** will be analyzed. ASM charts will be presented to show how many clock cycles are needed for performing these statements.

2. Architectures and characteristics of interacting processes

The realization of the interacting control units is important for analyzing the synchronization and communication statements. So both a Mealy and a Moore machine will be used to realize control units. The process architecture of figure B-10 will not be discussed here, because this one causes big problems when connections to other processing and control units must be made. This is due to the use of separated latches, instead of Master Slave flip-flops. It is hardly possible to make a structured design by using this kind of process architecture. So only Master Slave flip-flops will be applied in the presented architectures. In the first case both control-units are Mealy machines. In the second case Moore machines are used to implement the control units. Mixed architectures of Mealy and Moore control units are also possible, but not discussed here. The characteristics can easily be derived from the two presented architectures.

2.1. Architecture 1

The first architecture is the one as shown in figure C-2.

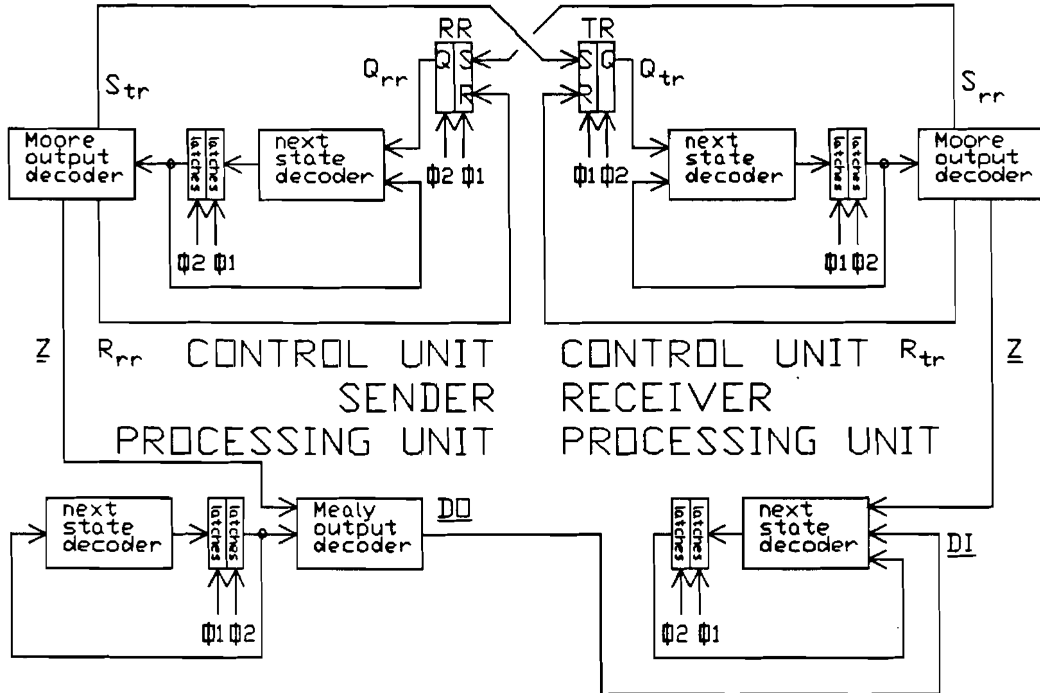


Figure C-2. System architecture 1

In this figure only the important signals and combinatorial circuits are drawn. Some characteristic functions of the combinatorial logic in the processing unit are assumed in this example.

- The output decoder of the processing unit in the sending process is an interconnection element. The data that has to be send will be routed to the output **DO** of this processing unit. A control vector $\underline{Z}_s$ performs this routing.

- The next state decoder of the processing unit in the receiving process is an interconnection element. The input **DI** of this processing unit will be routed to the register in which the received data must be stored. A control vector **Z**r performs this routing.

The **event** flip-flops are labeled as follows.

TR: Transmitter Ready. This flip-flop manages the **Trx_rqst** event.

RR: Receiver Ready. This flip-flop manages the **Rec_rqst** event.

The signals to and from these flip-flops are named as follows.

S_{tr}: Set input of flip-flop **TR**.

R_{tr}: Reset input of flip-flop **TR**.

Q_{tr}: Q output of flip-flop **TR**.

S_{rr}: Set input of flip-flop **RR**.

R_{rr}: Reset input of flip-flop **RR**.

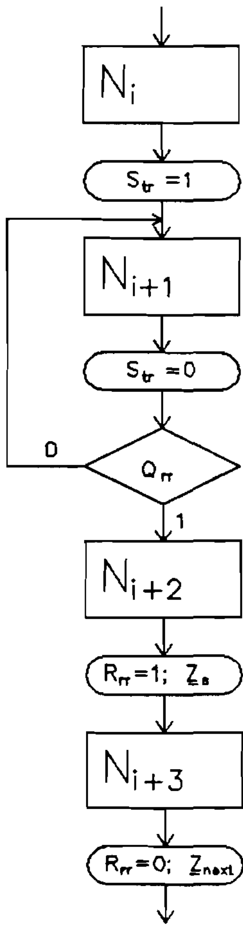
Q_{rr}: Q output of flip-flop **RR**.

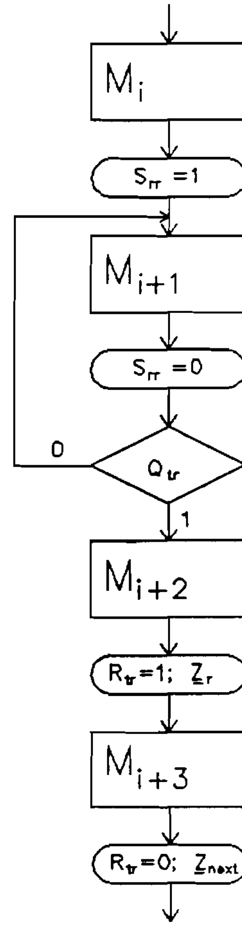
Performing the synchronization and communication statements of the previously described processes will lead to ASM charts for the control units of both the sending and the receiving process, as shown in figure C-3. These ASM charts indicate how many clock cycles are needed to perform these statements.

Remarks:

- The synchronization mechanism takes care for the fact that the sender and receiver enter resp. state N_{i+2} and M_{i+2} at the same time. In that state the data is on the connection (bus) between the processes, since the control vectors $\underline{Z}_s$ and $\underline{Z}_r$ are active simultaneously. During the transition to the next state the transmitted data will be clocked in a register of the receiving process.

- It takes three clock cycles to synchronize the two processes and to exchange the data. In the first one the request is send to the other process by setting a 'synchronization flip-flop'. In the second clock cycle the status of the other process is polled, and in the third cycle the data will be transferred. If more data has to be exchanged (it does not matter in which direction) between these two processes, then every clock cycle data can be transferred, because the processes are synchronized and this situation is maintained during the communication. If no more data has to be exchanged then the next operation can be performed in state N_{i+3}. This is represented by $\underline{Z}_{next}$.





proc_i (sender)

proc_j (receiver)

Figure C-3. ASM charts of architecture 1

2.2. Architecture 2

The second architecture is the one as shown in figure C-4.

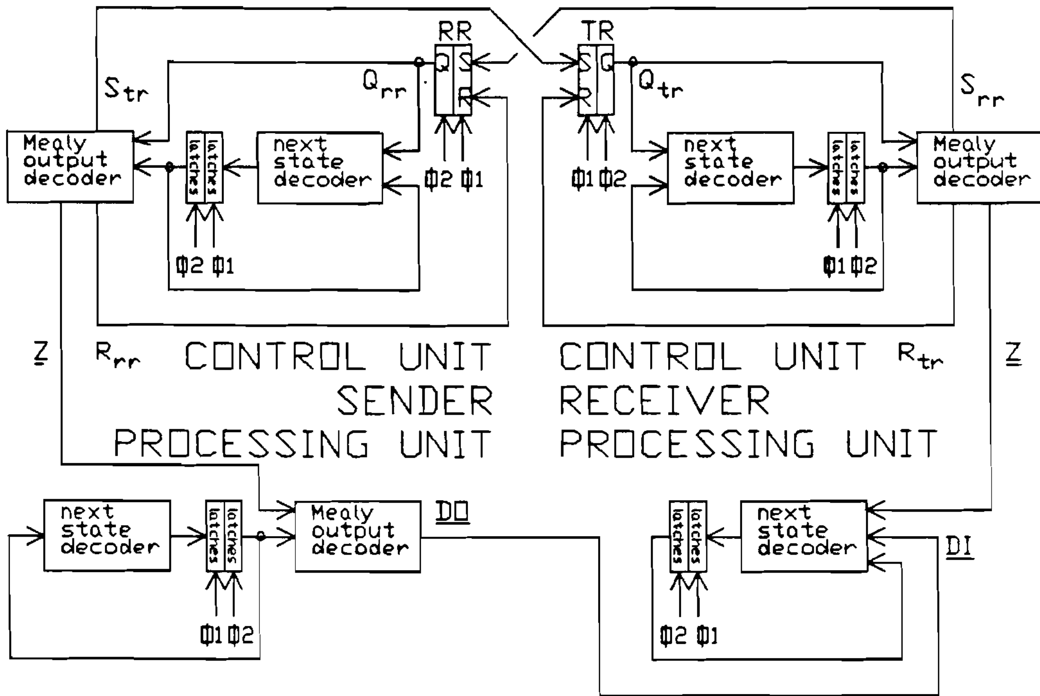


Figure C-4. System architecture 2

Also in this figure only the important signals and combinatorial circuits are drawn. The only difference with figure C-2 is that the Q outputs of the event flip-flops go both to the output decoders and the next state decoders of the control units.

Performing the synchronization and communication statements in the same way as in the previous architecture will lead to ASM charts for the control units of both the sending and the receiving process, as shown in figure C-5. These ASM charts indicate how many clock cycles are needed to perform the statements.

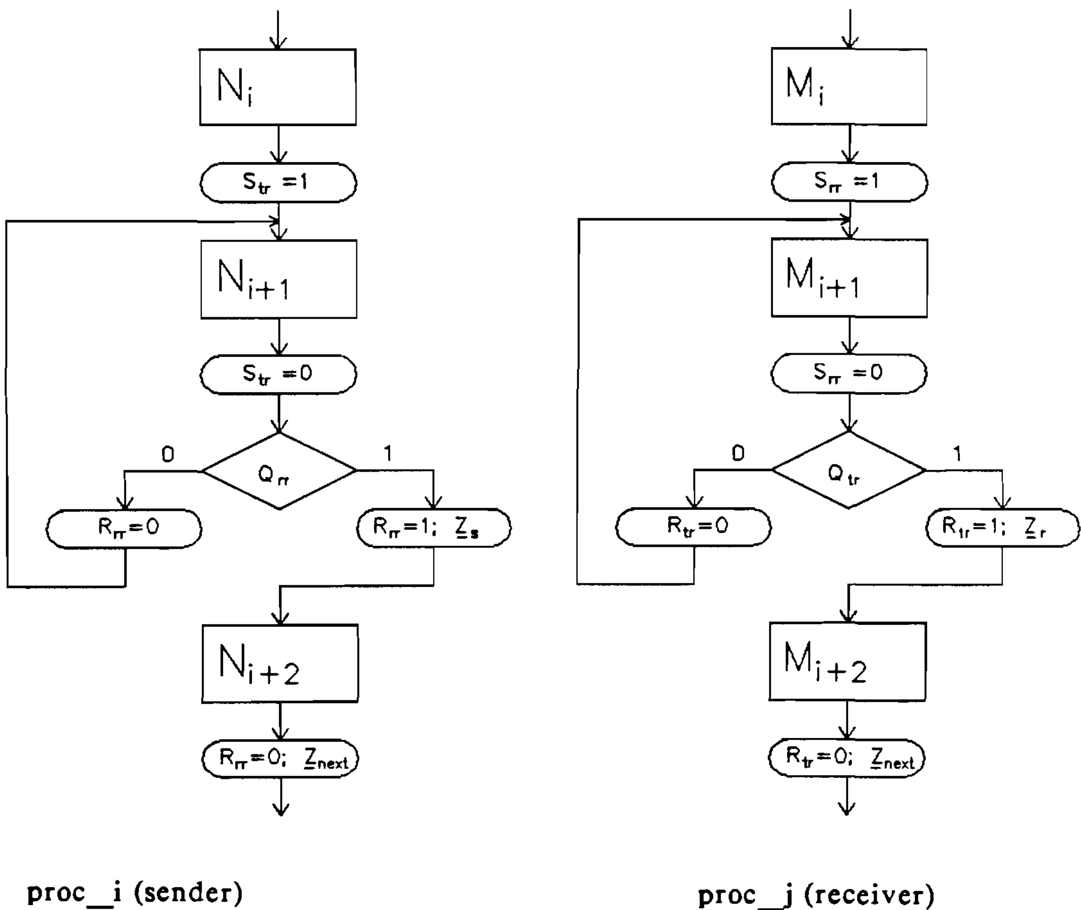
Remarks:

- The synchronization mechanism takes care for the fact that the sender and receiver enter resp. state N_{i+2} and M_{i+2} at the same time. If the sender is in state N_{i+1} and the receiver in state M_{i+1}, then the data is on the connection (bus) between the processes, since the (Mealy) control vectors $\underline{Z}_s$ and $\underline{Z}_r$ are there

simultaneously. During the transition to the next state the transmitted data will be clocked in a register of the receiving process.

- It takes two clock cycles to synchronize the two processes and to exchange the data. In the first one the request is send to the other process by setting the event flip-flop. In the second clock cycle the status of the other process is polled, and the data will be transferred if the other process is also ready for it. If more data has to be exchanged (it does not matter in which direction) between these two processes, then every clock cycle data can be transferred, because the processes are synchronized and this remains during the communication. If no more data has to be exchanged then the next operation can be performed in state N_{i+2} and M_{i+2}. This is represented by $\underline{Z}_{next}$.

- With this architecture processes can synchronize faster than with the previous one, because the control units are Mealy machines.



proc_i (sender) proc_j (receiver)

Figure C-5. ASM charts of architecture 2

3. Conclusions

As a result of the analyzed interacting processes, a few recommendations can be given.

- The architecture of figure B-10 is a bad choice because it is very hard to connect interacting processes (without adding interfacing latches) in a structured way.

- Mealy machines are preferable to realize the control units of processes. The reason for this is that processes can synchronize one clock cycle faster, compared to Moore machine realizations.