Requirements and architecture modeling tool specifying synchronization and communication for implementing concurrent transformations in VLSI

Benders, L.P.M.

Award date:
1988

EB 131

5320

Eindhoven University of Technology
Department of electrical engineering
Digital systems group (EB)

REQUIREMENTS AND ARCHITECTURE MODELING TOOL SPECIFYING SYNCHRONIZATION AND COMMUNICATION FOR IMPLEMENTING CONCURRENT TRANSFORMATIONS IN VLSI

Leon Benders

software techniques translated into
hardware

August, 1988

Master thesis report.
Coach: Prof. ir. M.P.J. Stevens

ABSTRACT

In application specific VLSI designs e.g. real-time and complex data transformation applications, it is essential to implement concurrent processes, which access shared data, share resources, communicate with each other and are synchronized to execute their functions. The number of transformations and events is limited and known. But even in this class of problems deadlock and starvation can occur.

This master thesis report presents a solution for modeling, locating, implementing parallel processes and their synchronization and communication in one VLSI circuit. Only process synchronization, scheduling, mutual exclusion, request/acknowledge synchronization and data communication primitives are required to specify parallel processes in a VLSI design. These primitives based on simplified software techniques are implemented in hardware using architecture modules to execute the control.

A digital system design starts with a global functional description. The concurrent processes have to be recognized to fullfil that function. A tool consisting of two models (requirements and architecture model) locates these parallel processes focussing on the data flow and transformation's data processing. The function of the digital system is defined and further specified in the requirements model. The transformation's description specifies the data transformation, but also contains the necessary primitives for the process interaction. The method avoids an implementation bias and sequentialization of processes. The description should prevent deadlock and starvation, but this can't be guaranteed.

In the next phase all transformations are allocated to parallel operating architecture modules in the architecture model. The allocation may sequence the transformations, when the timing requirements allow this. It redefines process descriptions especially the required synchronization and mutual exclusion and introduces new transformations to implement the allocation. Data transformations are allocated to processing architecture modules (PAM); the control transformations for synchronization, mutual exclusion to resource management (RMAM) and to scheduling architecture modules (SAM). To simplify implementation and to avoid too much synchronization overhead process knowledge may be implemented in these modules.

Often encountered identical transformations in the requirements model are allocated to special modules redefining the model. The transformation is started by other modules (shared processing). The requests are scheduled/served by the server architecture module (SVAM), which connects the caller with the executer, so that both can exchange data (parameters and result) and synchronization signals. Instead of direct information exchange connections, a buffer for the parameters and results can be used reducing the waiting time of the partners and increasing flexibility.

To prevent busy-waiting for synchronization and shared processing events a simple dispatching scheme can be implemented in the PAM using interrupts to react immediately to events. The contents of the datapath is not effected by the interrupt handling. Primitives divide the transformation into tasks, which are made runnable by the occurrence of an interrupt event or by the SAM.

"Er bestaat geen mogelijkheid om na te gaan welke beslissing beter is, want er is geen vergelijking. Wij maken alles zomaar voor het eerst en onvoorbereid mee, net als een acteur die voor de vuist weg een stuk speelt. Maar wat kan het leven waard zijn, als de eerste repetitie voor het leven al het leven zelf is? ... Het leven lijkt daarom altijd op een schets. Hoewel het woord 'schets' evenmin juist is, want een schets is altijd een ontwerp voor iets, de voorbereiding van een schilderij, terwijl de schets van ons leven een schets is voor niets, het ontwerp zonder een schilderij."

Milan Kundra
"De ondraaglijke lichtheid van het bestaan"

CONTENTS

1. INTRODUCTION

A digital system, a VLSI circuit is a combination of hardware and software with a number of programmable parallel processing modules. The applications are real-time systems, multiprocessor systems, distributed operating systems, signal processing and data oriented processors.

Since the early days of digital system design a trend has been seen in which the complexity of digital system design constantly increases. At the same time as the lifetime of products is getting shorter, the product development cycle is lengthening. So today we are trying to manage the ever growing complexity of digital systems in such a way that the period needed to specify and design those systems can be reduced.

The digital system group of the Eindhoven University of Technology is working on the development of a methology for capturing the complexity. To get control of the complexity we should make use of the intrinsic hierarchical character of the system and the design cycle should be structured so that computer tools can be used. Common design steps should be automated. This requires a formal language to describe systems and a formal method to translate the system description into a digital design.

In this master thesis report we will discuss the implementation problems of parallel processes in a VLSI circuit using a structured design methology. [Slenders,1987] describes a structured approach for the functional decomposition of a digital system.

The development process of a digital system consists of several phases. The system design is the conversion of the algorithmic design into an architecure with concurrent operating modules (functional level). The logic design is the conversion of the architecture into a structure. We concentrate on a dedicated VLSI design implementing parallel processing modules.

In all the development phases a designer will encounter problems concerning concurrency. In this report the main subject is a model for locating and implementing parallel processing in one VLSI circuit. We will tackle the following problems:

- how can we find implementable parallel processes?
- how can we divide a process into more processes without losing any potential concurrency?
- how can we describe those processes and their concurrency?
- how can we model the architecture?
- can we introduce hierarchy and leveling of processes into that architecture?
- what are the required synchronization and communication primitives for parallel operating processes in hardware?
- how should we model shared processing?
- what are the required synchronization and communication primitives for shared processing?

The formal description of parallel processes and the system is no part of this project. Other members of the project group are working on this subject. We will use the structured methology and focus on the parallel functional blocks and primitives to implement parallel operating modules.

We will discuss a tool to locate parallel processes and to find implementable functions. We will see how synchronization and communication is introduced in translating the requirements into an architecture. In every phase we use an algorithmic description for the design. Several phases of the design will be mixed. Sometimes a next phase will require changes in an earlier model or phase. After the discussion of parallelism we focus on the necessary primitives.

After discussing the usefulness of software techniques for synchronization and mutual exclusion in hardware we introduce primitives for synchronization and communication, which are used in the process description. An implementation of the software techniques is presented in an appendix. The use and consequences of the techniques are illustrated in another appendix.

Chapter two (concurrent process modeling) is based on [Hatley,1987] and [Ward,1985]. All examples and pictures are from these references unless stated otherwise. We use the terminology of Hatley. The goal of that chapter is to translate their methods to application specific VLSI design. Chapter three is a comprehensive discussion of operating system techniques based on [Janson,1985]. All examples, nomenclature are taken from his work. We focus on the dedicated hardware requirements and the usefulness of the operating systems features in VLSI design.

Another important item is shared processing. A processing module is used by other modules to execute a special process, which requires dedicated hardware. We will talk about the shared processing architecture. Finally we will discuss the implementation of the scheduling primitives in the processing module.

2. IDENTIFICATION OF PARALLEL PROCESSES AND SYNCHRONIZATION USING A MODELING TOOL

2.1. introduction to system development and specification

A modern chip requires a system approach. A digital system consists of synchronized processing modules (datapath, control logic and a data structure). The development process of a VLSI circuit is top-down with a strong feedback while designing. The complexity of digital systems can be controlled by decomposition into a layered structure of systems plus inter-subsystem interactions. [Stevens,1987]

The current VLSI development process consists of three phases:
- requirement and specification phase
- design phase
- implementation phase

and a validation during all phases. The validation process checks completeness, consistency, correctness and invariance.

We have four development levels: behavior, functional, structural and physical. The behavior level is the specification of the requirements and behavior description of functions, which should be unambiguous. The functional level defines the architecture and the decomposition in functional modules. The structured level translates the subfunctions into logic circuits, functional cycles, microprogramming, finite state machines, boolean expressions. The physical level is the translation of submodules into physical devices, placement and interconnection of these physical modules.

The first phase (specification) has four subphases: requirement elaboration and specification, system attributes, process definition and verification. First we transform the user needs into a form that is precise, consistent, verifiable and analyzable. We define functional needs, trade-off criteria and the required performance. The functional and performance system requirements are the objectives of the design and also the constraints for the design. The system attributes are evaluation criteria for quality comparison (reliability, availability, flexibility, reconfigurability, modularity, cost etc).

The next subphase is the process definition: the characterization of input stimuli and required responses. The functional requirements decompose into data processing, communication requirements and precedence constraints. The performance requirements decompose into resource requirements and scheduling. The information and control flow is modeled and the major operations and their locations of occurrence are identified. The results of the process definition are: definitions of processes, resource requirement of processes and interactions among processes. The last phase is to verify specified process definitions against users requirements. We need an analyzable specification language. Until now we weren't interested in the implementation.

The second phase (design) has three subphases: decomposition, functional specification and verification of the design. The defined required processes are converted into implementable specifications. The resulting functional models should satisfy requirements specification. The top-down approach uses hierarchy of abstraction levels and stepwise refinement. In the decomposition phase we identify the tightly coupled processes and divide the system into subsystems corresponding to these processes. The decomposed system must hide local information, be consistent, complete, unambiguous, testable, satisfy corresponding requirements, satisfy the parametric logic specifications (minor changes in requirements may not require a redesign of the entire system) and be expandable.

We group the submodules or subprocesss into different sets (partitioning), which become functional modules. So we increase the modularity and testability, reduce complexity of interfaces and reduce the inter-subsystem communications. The optimal partitioning is difficult and heuristic.

The subphase functional specification of the partitioned processes defines the characteristics of the functions to enable minimization. Characteristics of the functions are types of operations to be performed (matrix, floating point, integer), resource requirements (processing power) and speed requirements (frequency/execution speed of the function).

The last subphase is the verification of the design. The behavior of the models is checked (verification of correctness, evaluation of the effectiveness) by analytical and simulation modeling. In this phase the design is described in blocks.

The last major phase has also three subphases: implementation of architecure, mapping of functions in hardware/software and validation. The implementation of the architecure has several levels: architectural/functional, register transfer, logic, circuit and physical level. Regular structures simplify implementation. The mapping depends on physical constraints, the architecture chosen and the available technology. Validation is the verification of a design by simulation. Testability is a limiting factor in VLSI development.

2.2. the model and the modeling techniques

Modeling and modeling techniques are used in all development phases. We will discuss these techniques and the characteristics of the models. Before we can design a dedicated hardware system, we have to define the problem. This problem definition should be implementation independent. We need to know the data flow, the data transformations and the control of the data transformations. It is essential to define parallel processes. A model should focus on these aspects.

To simplify the implementation and redesign we try to locate parts that are independent. We organize our model for maximum independence and try to partition the problem into parts with minimum interfaces. We will present a model and a modeling tool based on [DeMarco,1987], [Ward,1985] and [Hatley,1987]. The method is called structured analysis and design.

For the modeling it is vital to provide a notation that facilitates a conception of the problem with minimal artificial restrictions. The model is capable of formulating problems and expressing solutions in terms of parallelism and data structures as well as in terms of sequences and operations.

The modeling language is capable of separating the data needed by a process from the control that actually makes the process operate. It distinguishes the time-continuous and time-discrete behavior and is able to model the interactions between the two (real time problems!).

The final model is produced by a structured method, consisting of modeling tools and techniques that illuminate certain aspects of the desired system during the specification process. The model is useful for real-time processing, data processing and process control.

The system specification process must define the system as a whole as well as its partitioning into processes. The system development process has three phases:

- definition of what problem the system is to solve (requirement model)
- deciding how that system is to be structured (architectural or design structure, architecture model)
- actual implementation of the design structure.

Specially how to structure the system is a difficult trade off process, which can't be executed using rules. To state the problem and to capture its solution, we build two models of the system on paper: the requirements model and the architecture model. The requirements model (essential model) is built as a technology independent model of the system's essential requirements and the architecture model is a technology dependent model of the system's structure.

The requirements model describes what the system must do (essential activities) and what data it must store, so that the description is true regardless of the technology used to implement the system. First the high level essential details are determined, the corresponding details are created at a next lower level (information hiding!).

Each successive model is derived from the previous model by incorporating additional information that was suppressed or ignored in the previous model. The requirements model consists of two parts (models): a model, which focusses on defining what the system must interact with and a model describing the required behavior of the system (transformation schemes, data flows). The first model is a description of the boundary between the system and the environment showing the interfaces between the two parts and a description of the events to which the system must respond. The most important task in developing the environmental model is to specify the events to which the system must respond. This is the basis for the creation of the behavioral model.

Our tools use rules, techniques and heuristics to check the consistency of the models (specially the levels). Heuristics are used to establish and to check the completeness of the model, the partitioning and the usefulness of the interfaces [Ward,1985]. These techniques, rules, heuristics won't be discussed in detail, because we are mainly interested in the model description to identify the necessary primitives for synchronization and communication of parallel processes and in the mapping of these processes into parallel operating hardware units.

After discussing the method and its benefits, we need to address the reasons for selecting this tool. The main reason is that the tool focusses on the data flow and the external events. This removes a possible implementation bias. Because of this data approach we are able to find

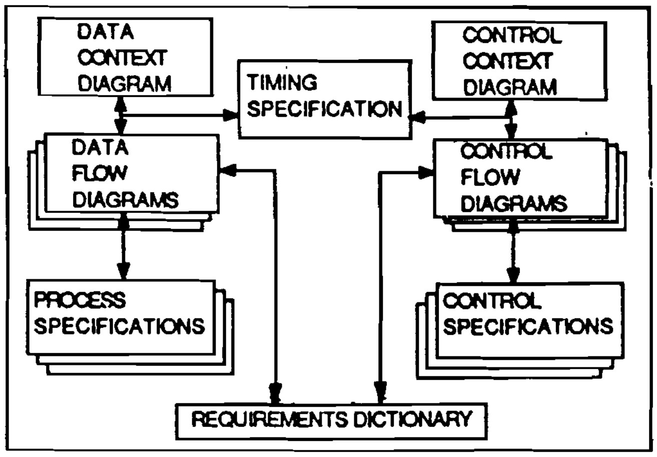


figure 1: components of the requirements model

intrinsic concurrency in the data processing. The composition of the data flow indicates a functional decomposition of the system. The use of graphics to display the system and to indicate potential concurrency makes the system easy to understand and makes the decomposition easy to grab. The method clearly differentiates between logical and physical considerations, but still addresses a number of aspects (processing, control, timing, architectures specifications, hierarchical and iterative nature of the system) in an integrated manner.

2.3. the requirements model

2.3.1. THE STRUCTURE OF THE REQUIREMENTS MODEL

We use the requirements model to specify the system. It is a tool to decompose the system description into functional blocks. The model consists of several components (figure 1). The whole system requirements model describes the system's data flow (shown in the data flow diagrams; **DFDs**), data processing (shown in the process specifications; **PSPECs**) and the control processing (shown in the control specifications; **CSPECs**). These components interconnect with each other as shown in figure 2.

The components are:

- **DFDs**; data flow diagrams decompose the system and its functions
- **PSPECs**; process specifications specify in concise terms each detail of the system's functional requirement. Control flows generated inside **PSPECs** through test on data flows are called **data conditions**. They flow to **CFDs** (control flow diagram), where they are treated as any other control flow.
- **CFDs**; control flow diagrams map control flows along the same paths as the data flow may travel. **CFDs** mirror the processes and the stores of the **DFDs**, but do not show data flows
- **CSPECs**; control specifications specify control processing controls for the processes on the **DFD**.
- the **response time specification** defines the limits on response time allowed between events at the system input terminals and the resulting events at the system output terminals
- the **requirements dictionary** completes the model. It contains an alphabetical listing of all the data and control flows in the **DFDs** and **CDFs** along with their definitions.

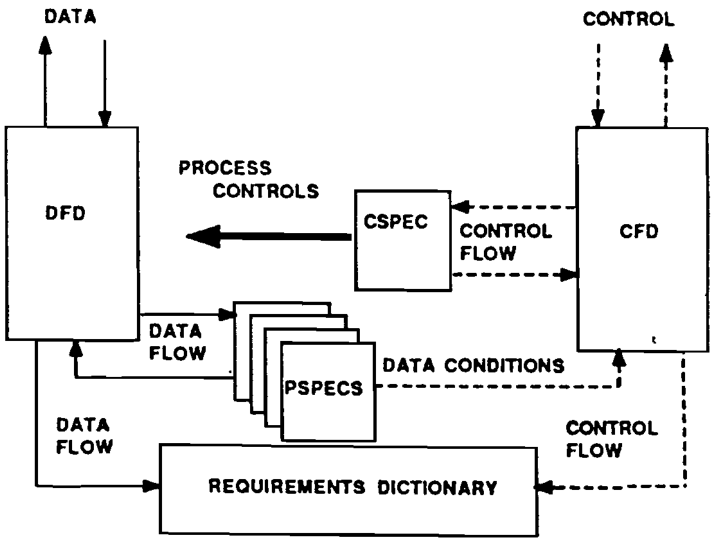


figure 2: the structure of the requirements model

The requirements model is built as a layered set of **DFDs** and **CDFs** with associated **PSPECs** and **CSPECs**. Each successive level of diagrams and specifications expresses a refinement of the higher level diagrams (see figure 3). Only on a new level a process of the upper level is further specified. The goal of building a leveled set of diagrams is to make a clear concise requirement statement at each level of detail and to partition the requirements into mutually exclusive subsets with precisely defined relationships. All redundancy has to be removed.

The context diagrams (see figure 3) establish the boundary between the system under specification and the environment. It is used to show communication between the system and the environment and the entities in the environment with which these systems communicates. They are the highest level diagram. We have process control and data context diagrams. ([Ward,1985] volume 2) discusses the development of these diagrams. Most important is the modeling of the external events, because the system must react to these events. An external event is something arising in the systems's environment at a specific point in time and requiring a preplanned response.

Naming issues of flow and process, specifying data relationship, modeling stored data, execution schemes and consistency checking are all discusses in [DeMacro,1987], [Ward,1985] and [Hatley,1987].

2.3.2. THE DATA PROCESSING MODEL

The processing model focusses on the data flow of the requirements model. A data flow is a pipeline through which data of known composition flows. It may consist of a single element or a group of elements. A flow can be time-continuous or time-discrete. The data flow diagram (**DFD**) is a primary tool depicting functional requirements. It partitions these requirements into component functions, processes and represents them in a network interconnected by data flows. Its main purpose is to show how each process transforms its input data flows into output data flows, and to show the relationship between those processes (transformations). A process is represented by a bubble, a data flow by an arc.

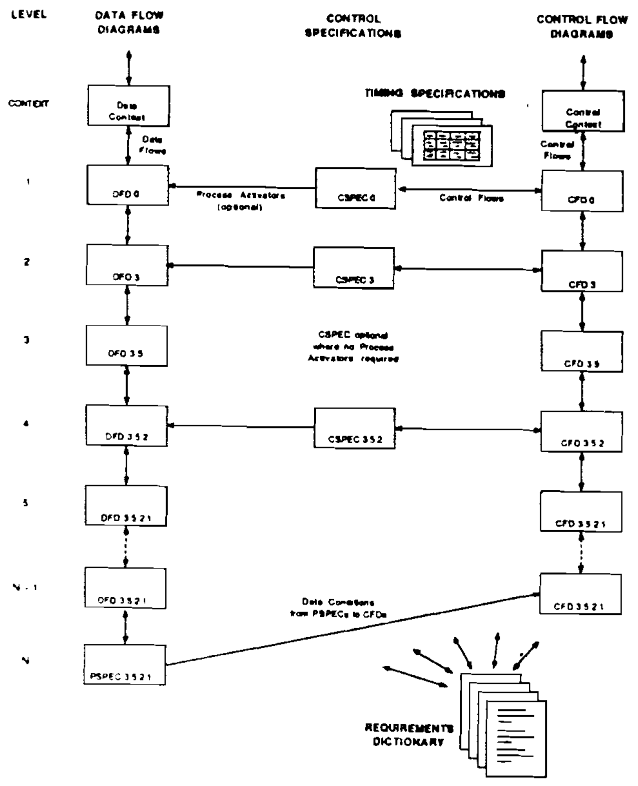


figure 3: composite chart of the requirements model

Besides the bubbles and the data flow we use data stores to define the system. A data store is a data flow frozen in time. The data information it contains may be used any time after storing information and in any order. An arrow head pointing from a store to a transformation means that some output flow of the transformation uses something from the store. An arrow head pointing from a transformation to a store means that the store is changed in value. A data store is represented by a pair of parallel lines.

Starting with the context process, processes are progressively decomposed into more detailed diagrams: a procedure called leveling. The data flow decomposes in parallel with the levels of detail of the **DFD**s.

However, partitioning a transformation into a set of lower level transformations carries the risk of introducing an implementation bias into the model. A transformation can be partitioned without introducing implementation dependence, if the partitioning makes use of the structure of the transformed data. Usually if we find a good name to cover the function, we have reached the bottom level in the requirements model. In the implementation phase such a function can be so complex, that it should be divided into smaller functions.

The processes without children are called functional primitives. A primitive process and its **PSPEC** share the same inputs and outputs. Every input to the process is used within some expression in the specification, which shows how every output from the process is generated. Essential is that a (parent) process can be broken down into smaller (children) processes on a lower level. The interfaces of these groups of processes are already defined in the upper level. So we look only at the children of the process that is split up and neglect the other processes defined on the higher level. The inputs and outputs of a parent and its child **DFD** must match. Verifying that is called balancing and is a principle consistency check.

Each primitive process has a process specification (**PSPEC**) and a requirements dictionary (**RD**) that specifies every data flow. We describe a primitive process in a pascal like language using "**cobegin**" and "**coend**" to indicate parallel actions and some primitives defined for the events handling. Processes that are not primitive, are broken down into more detailed **DFDs** and have global **PSPECs**. **PSPEC** use a pseudo programming language: structured English. The intention of structured English is to combine the rigor of a programming language with the readability of english. The structures are simple single-entry, single-exit constructs:

- **concurrency**: more than one activity takes place simultaneously;
- **sequence**: activities occur in a specified time sequence;
- **decision**: a branch in the flow of activities is made based on the results of a test on an input;
- **repetition**: the same activity is repeated until some specified limit or result is reached.

It is not the final description, but it is just a functional description using input and output flows. At a high level the specification can also be in a pre- and postconditions specification.

One use of the data structure is to expose potential parallelism in the required processing. Parallelism may be indicated when a transformation has a composite input flow whose elements can be acted on independently. Parallelism may also be indicated when a response requires the production of two or more independent output flows. *The partitioning shown in* **DFD***s should not be read as a prescription for parallel processing. It is merely a statement that the transformations are required in any implementation and have no necessary sequence.* ([Ward,1985], volume 2).

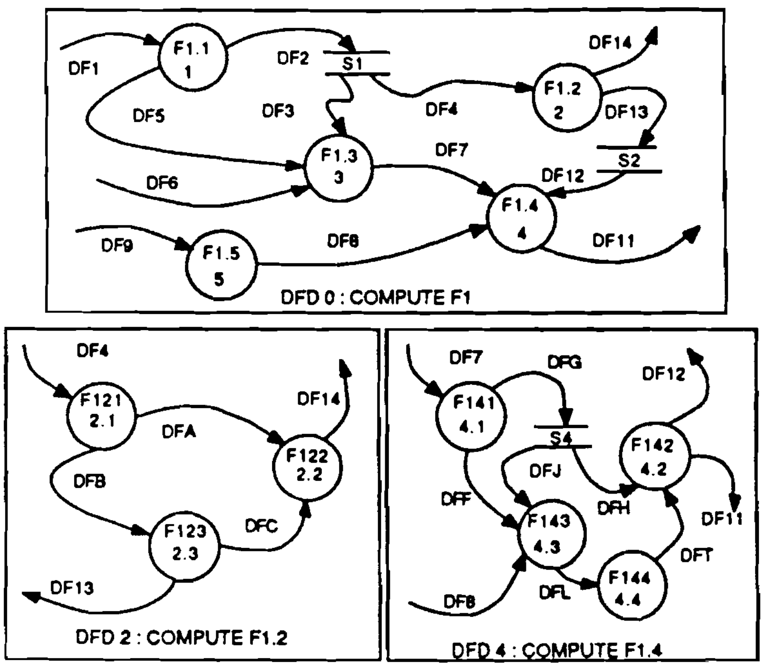


figure 4: example of leveling process 2 and 4 are decomposed

2.3.3. THE CONTROL MODEL

The control model focusses on the control flow between the processes defined in the data flow diagrams. These DFDs illustrate what functions the system is to perform, but don't tell under what circumstances, it will perform them. We need to enable and to disable the processes on the DFDs under some specific circumstance. We even need processes to transform events into control signals. Some transformations need data about the previous operation of other event responses for correct operation. Data transformations may need to be synchronized via a control transformation. The control transformation enables and disables the data transformation to enforce the constraint.

Control flow diagrams (CFDs) show the flow of control signals (events) in the system. Control specifications (CSPECs) indicate how the control processing takes place. The processes in CFD are the same as on the DFDs, but the CFD shows the control signals associated with each process. Control signals are represented as dashed arcs. The CFD contains control stores, which don't differ from the data store. A given store may be unique to a DFD, a CFD or used in both. The control structure is integrated with the process structure in such a way that it inherits all the leveling, numbering and balancing properties.

CSPECs describe the control process/transformations. We define a CSPEC bar as a symbol used on a CFD to indicate the interface between the CFD and its CSPEC. The inputs of CSPECs are control flows from the CFDs and the outputs are process activators and control flows entering the CFDs. Process controls are generated from the logic in the CSPECs and activate and deactivate process in the corresponding DFD. Information from the PSPEC flows to the control structure. This information consists of control flows that are derived from data. They provide the link between **DFDs** and **CFDs** and are called **data conditions.** A **requirement dictionary** contains the definition of all the control signals in the system. **CSPECs** contain diagrammatic and tabular representations of finite state machines. The control signals (always discrete) flowing to and from the **CSPEC** bars on the **CFDs** are the inputs and output of these finite state machines. The control transformation must combine the flow with internal memory to produce the selected output flow.

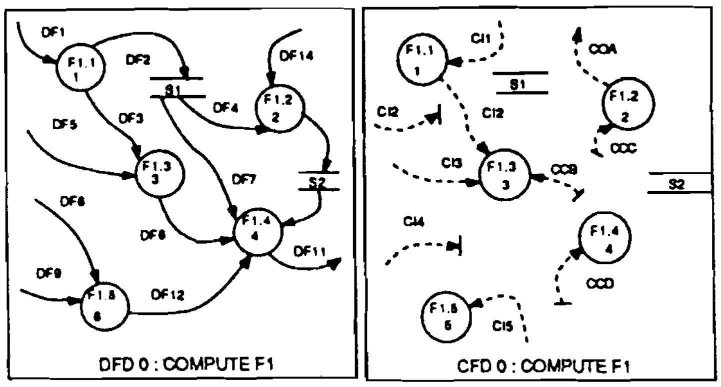


figure 5: DFD and CFD

Possible diagrams for the **CSPEC** are a state transition diagram (STD) or a process activation table (PAT). STDs show the states of the system and how they are influenced by control signals. They respond to events represented by control flows and show the corresponding actions that the system must take. Events and actions are represented on STDs as event/action labels on each of the transitions between the states. PATs show the circumstances under which the process on a **DFD** are enabled and disabled.

The actions from an STD enter a PAT, which enables and disables the appropriate processes. We are just describing control. You should not confuse this with the implementation. The description is a tool to specify the function of the control. A finite state machine is usually easier to read then a pascal language, if we are describing a control transformation. Figure 6 contains an example from [Hatley,1987].

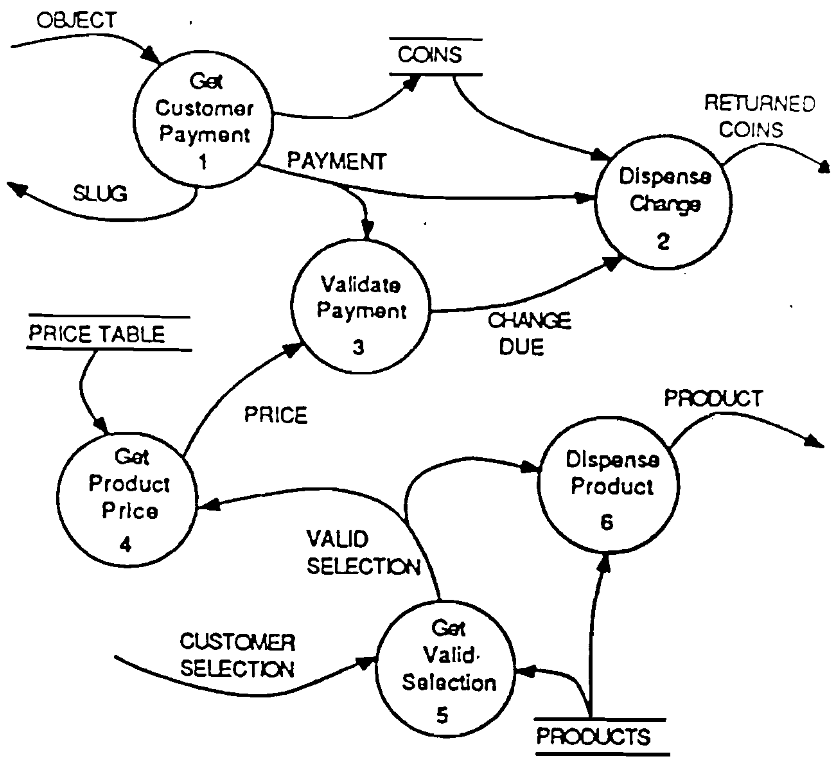
Processes that are not controlled from a **CSPEC** are data triggered. They are enabled each time there is sufficient data at their inputs to preform the specified function.

The control flows labeled enable and disable from the control transformations to the data transformation have a special function in the model. They are not truly inputs to the data transformations, but simple switch the data transformations on and off. For a control transformation only event flows are allowed as inputs and outputs. Incoming event flows for data transformations labeled enable, disable, trigger are interpreted as prompts not as inputs. Only control transformations may prompt other transformations.

We have three types of events:

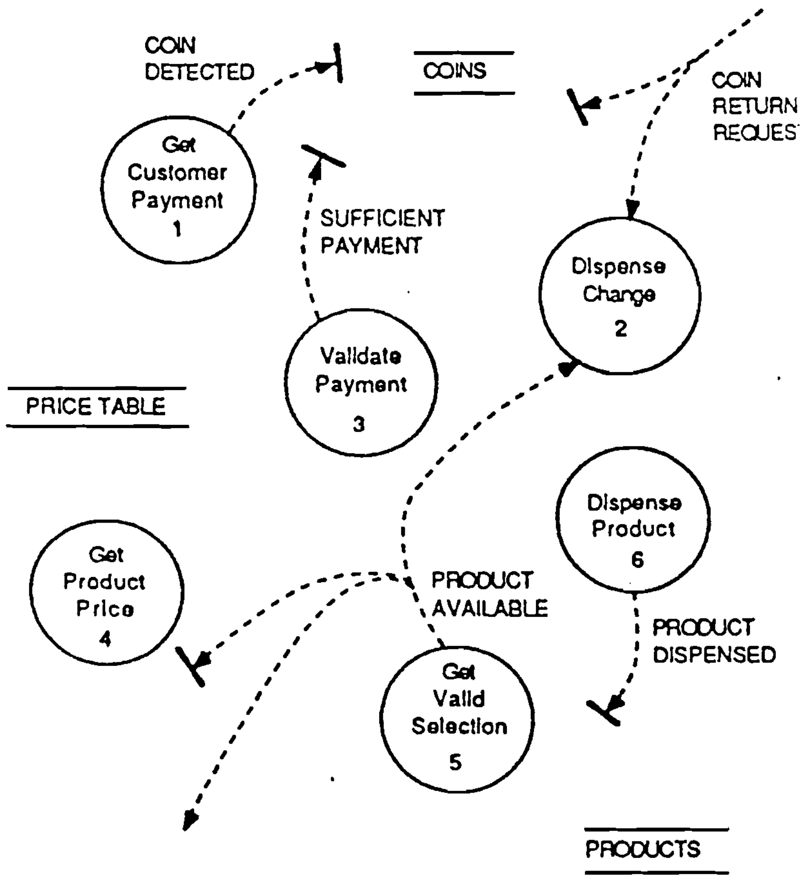
- **flow direct**: the arrival of the control flow signals the event
- **flow indirect**: a control flow may require an event recognition mechanism. A flow triggers an event recognition mechanism that in turn triggers a response.
- **temporal**: the event is indirectly recognized by the passage of time

A strategy for constructing state transition diagrams from an event list is describe in [Ward,1985].



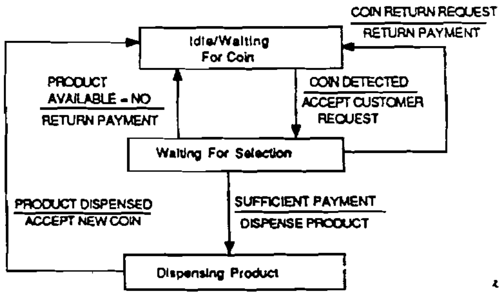
Vending machine DFD.

figure 6a: example



Vending machine CFD.

figure 6b: example



CSPEC 0: VEND PRODUCT: Sheet 1 of 2

Process Activated / Control Action	Dispense Change 2	Dispense Product 6	Get Valid Selection 5
Accept Customer Request	0	0	1
Return Payment	1	0	0
Accept New Coin	0	0	0
Dispense Product	1	1	0

CSPEC 0: VEND PRODUCT: Sheet 2 of 2

COIN RETURN REQUEST = \ User request for coin return\
2 Values: True, False.

PRODUCT AVAILABLE = \ Product availability status\
2 Values: Yes, No.

Figure 2.6. Primitive control entries from vending machine dictionary.

OBJECTS = COINS + SLUGS

COINS = QUARTERS + NICKELS + DIMES

PRODUCTS = [SODA | GUM | CANDY]

Figure 2.3. Composite data entries from vending machine dictionary.

```
PSPEC 3 . VALIDATE PAYMENT
INPUTS :      PRICE, PAYMENT
OUTPUTS :     SUFFICIENT PAYMENT, CHANGE DUE
If PAYMENT ≥ PRICE then
        Issue SUFFICIENT PAYMENT = Yes
        Issue CHANGE DUE = PAYMENT - PRICE
Otherwise
        Issue SUFFICIENT PAYMENT = No
```

Figure 2.2. Vending machine PSPEC.

figure 6c: an example

The control models and **CSPEC** close the loop of the system. The whole requirements model is a feedback control loop.

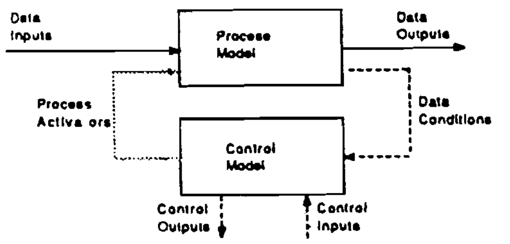


Diagram 2: The Requirements Model as a Feedback Controller

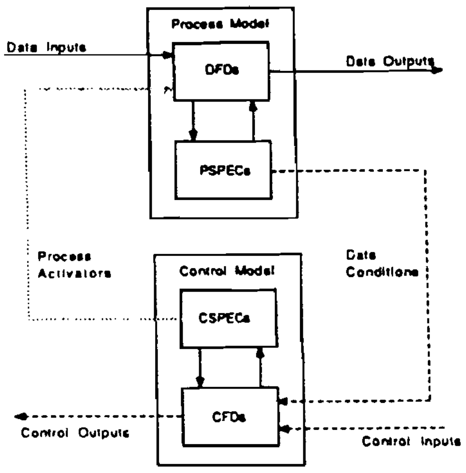


Diagram 3: The Symmetry of the Requirements Model

figure 7: feedback control

Data flows enter the model through the **DFDs**. Some processes in the data structure produce control signals (data conditions), which feed into the control structure; signals from the **CFDs** drive the **CSPEC** and the **CSPECs** control some of the processes on the **DFDs**. (Note the analogy with the operating unit and control units of the basic model for hardware implementation of a transformation)

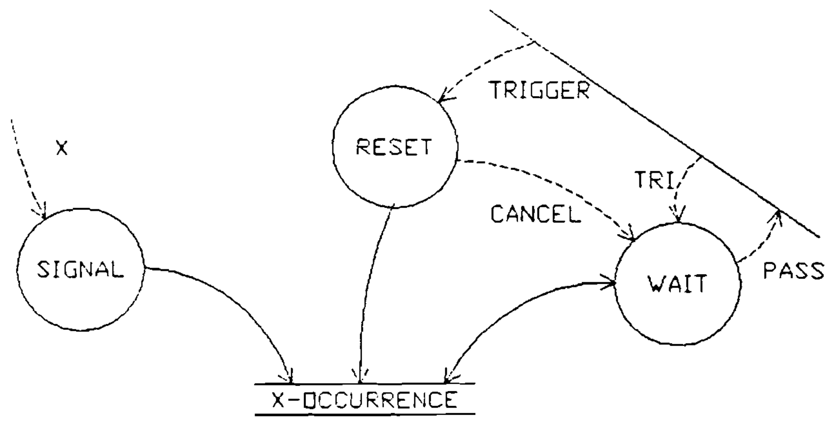
We try to minimize interfaces in our model, so the control transformations (**CSPEC**) of the transformation's scheme may need to be reorganized to achieve the best grouping. Incorporating a control hierarchy into the leveling is useful, because it allows grouping together processes that are activated and deactivate at once.

A few remaining remarks. A transaction center is a set of primitive processes of which one is selected by the control. Multiple instances of transformation centers are denoted by a double circle. The control flow can be connected directly to a control store with no intervening transformation. It is not necessary that a data transformation has a control flow.

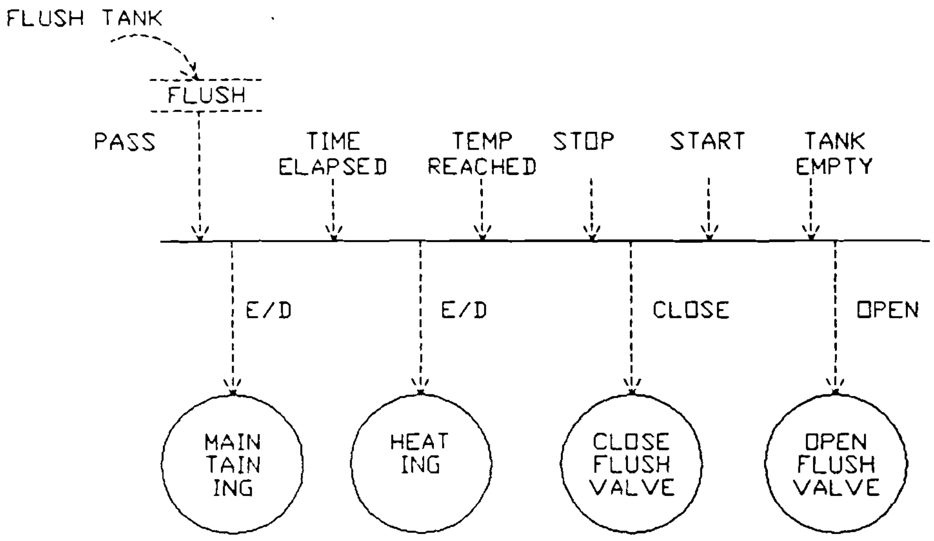
Certain situations that commonly occur in systems, can greatly increase the complexity of a state transition diagram. (e.g. a control signal can arrive in all states, but cause only action in one state). The control store will be used to avoid complex state models. It is used to remember information about events that have occurred, but are not yet to be used. Three operations are defined that effect the control store (see fig 8) The reset operation changes the value of the store to its initial value and also cancels any process "wait" in progress. The signal operation increases the value of the store by one. The process wait operation carries out the following logic:

```
decrement value of store by 1
if value of store >= 0 then
    issue PASS
    else
        do until(SIGNAL) wait
        issue PASS
```

The control store (dashed lines) with these processes is equivalent to a semaphore.



CONTROL STORE SEMAPHORE IMPLEMENTATION



CFD WITH SEMAPHORE STORE

figure 8: semaphore store structure

2.3.4. TIMING REQUIREMENTS

Beside the data and control transformations we need to specify the timing of the system. Timing requirements relate only to external timing. Internal timing is a design issue. Three types of timing are specified: external response timing; reception rates and recomputation rates of external inputs and outputs respectively; and timing required as part of the functionality of a user-required process.

Starting the response time specification early helps to understand the system as an event driven model. The response time specification is likely to be subject to re-examination during the design phase. These specifications must balance with the dictionary: all inputs and output signals must be defined in the dictionary.

2.3.5. REQUIREMENT DICTIONARY

To understand the system specifications we described all transformations and data flows. The dictionary defines all control and data flows in the diagrams. Flows or signals are either primitive or non primitives, the later consisting of groups of the former. Definitions of primitives describe their physical form or information content, including a list of attributes.

A special set of symbols is used to define the way primitives are structured into a group flow. A possible set of construction symbols is:

=	composed of
+	together with
{}	iteration of
[|]	select one of
()	optional
" "	literal
**	comment field
\\	description of primitive elements

2.4. the architecture model

2.4.1. THE STRUCTURE OF THE MODEL

The functional requirements at one level must be allocated to a physical structure with non functional requirements added. After describing the systems requirements we have to specify how the system will fulfill those requirements.

The tool for capturing this process is the **architecture model**, whose principal purposes are:
- to show the physical entities that make up the system;
- to define the information flow between these physical entities;
- to specify the channels in which the information flows;

These purposes are fulfilled using diagrams supported by specifications and a dictionary.

The requirements are mapped into an architecture model taking all the design constraints into account. These constraints include performance requirements, growth and expansion capability, testability and the implementation technology.

The architecture model is equally applicable to any level of system definitions. The upper levels show partitioning of the system into architecture modules regardless of hardware and software allocations. Lower levels show the hardware and software modules derived from that system partitioning. The architecture modules operate in parallel. All processes allocated to an architecture module have to be executed sequentially.

The physical model consists of parallel operating (processing) units called architecture modules. The requirements model, data processes and **PSPECs** and **CSPECs** are divided into groups, that are allocated to architecture modules. For the description of the architecture model we introduce several components (see figure 9). The **architecture flow diagram** (**AFD**) depicts the system architecture. It shows the physical partitioning of the system into its component pieces or modules and the information flow between them. Its main purpose is to allocate the functional processes of the requirements model to physical units of the system and to add more processes as needed to support the new physical interfaces.

The **AFDs** are supported by an **architecture module specifications** (AMSs) and an **architecture dictionary** (AD). Module specifications define the inputs, outputs and processes allocated from the requirements model for each architecture module. The architecture dictionary contains the data and control flow definitions of the requirements dictionary plus the allocation of these flows to architecture models.

In addition to the physical entities and the information flow between them, we need to capture how the information flows from one architecture module to another. For this we use the **architecure interconnections diagram** (**AID**). The **AID** shows the actual physical information channels between the **AFDs** architecture modules and to and from the environment. The **AID** is supported by **architecture interconnect specifications** (AISs), textual specifications that specify the characteristics of the communication channels. In VLSI design the AIS is just an

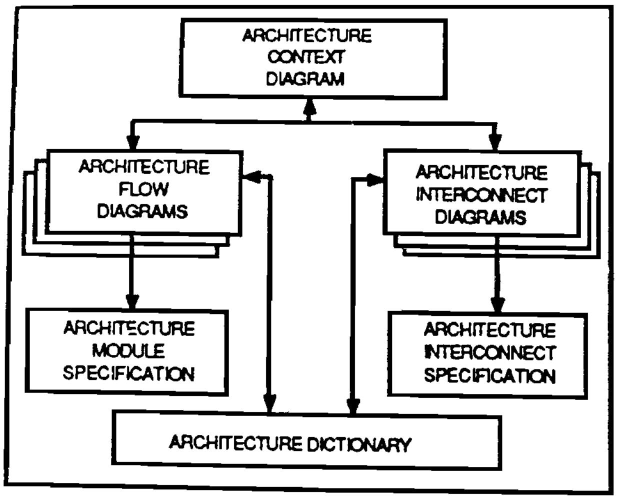

ARCHITECTURE CONTEXT DIAGRAM
ARCHITECTURE FLOW DIAGRAMS
ARCHITECTURE INTERCONNECT DIAGRAMS
ARCHITECTURE MODULE SPECIFICATION
ARCHITECTURE INTERCONNECT SPECIFICATION
ARCHITECTURE DICTIONARY

figure 9: architecture model components

electrical bus/signal description. (The method can be used for the design of several kinds of systems). The **architecture dictionary** also captures the allocation of data and control flows to specific interconnect channels (flow name, compose of, origin, destination, channel).

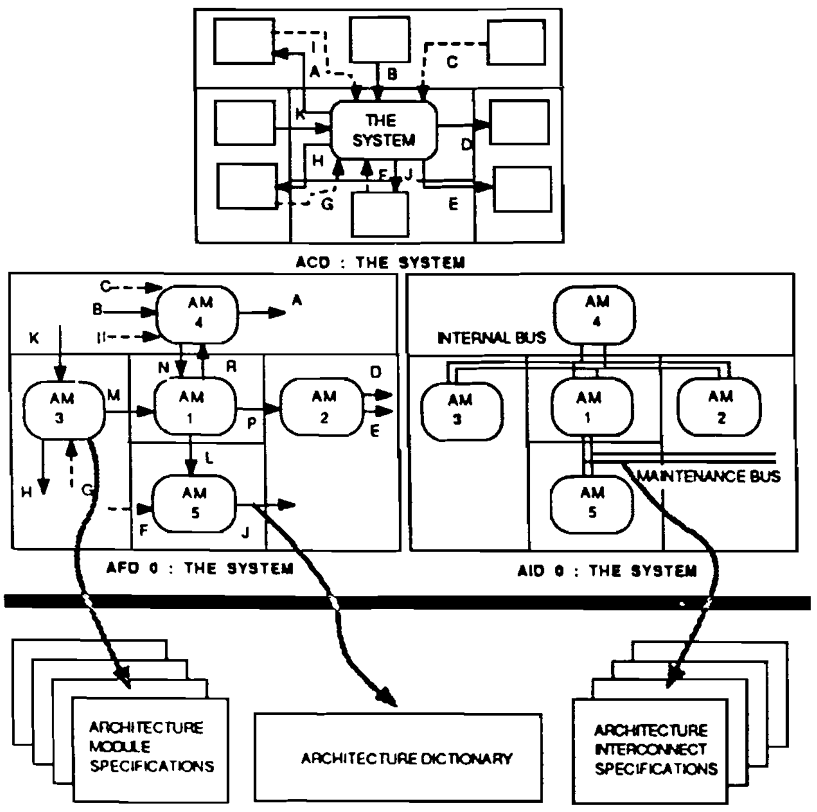


figure 10: the structure of the architecture model

The system architecture model is a statement of the system's physical modules, the information flow between them, and their interconnections (see figure 10). The architecture modules are in our context parallel operating circuits in a VLSI design, but a physical module could also be a computer or a micro.

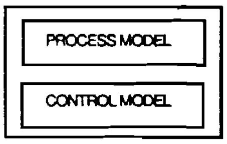
An information flow vector represents all the information that flows between any two architecture modules. These flows may be either single elements or grouping of elements and may contain data flows, control flows or both. In our model only one type of information flow channel exists (the electrical bus), but in other applications a mechanical or optical link can be used.

The system architecture model consist of a layered set of AFDs and AIDs and their associated AMSs and AISs. Each successive layer refines the configurations defined by the higher level diagram. To describe the architecture we need three new symbols to represent the architecture modules, the information that these physical processes communicate and the channels by which this communication takes place.

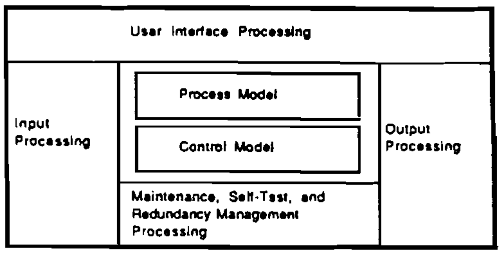
An architecture module is either a fundamental entity or a grouping. A fundamental entity corresponds to an individual physical module. A grouping of fundamental entities represents a subsystem, that contains several physical modules. The mapping between the architecture modules and the data and control transformations in the requirements model can be one-to-one or one-to-many in either direction. The reason for the latter is that several hardware and software technology decisions might have to be made. What may have been a primitive process from requirements perspective may turn out to be non primitive from the architecture.

2.4.2. THE STRUCTURING OF ARCHITECTURE MODULES

The system requirements and architecture are interrelated and must be developed in parallel. A given set of requirements can have many possible solutions depending on the decisions and trade offs made when transforming a technology-independent requirements statement into a technology-dependent architecture model. These decisions and trade offs cause iterations between the requirements and architecture models.



. Requirements template.


User Interface Processing
Input Processing
Process Model
Control Model
Output Processing
Maintenance, Self-Test, and Redundancy Management Processing

Architecture template.

figure 11: requirements and architecture template

In developing the architecture model, we allocate the requirements model to architecture modules and add the following:

- user interface processing
- input processing
- output processing
- maintenance, self-test, initialization and redundancy processing

The architecture module doesn't always contain these blocks. It depends on the problem and its implementation form. A problem that would require distributed systems with several micros, will have many blocks. In our problem field only input and output processing are of interest.
The input and output processing blocks represent the additional processing beyond that of the requirements model, needed for each architecture module to communicate with the other module and to transform the information to and from an internally usable form.

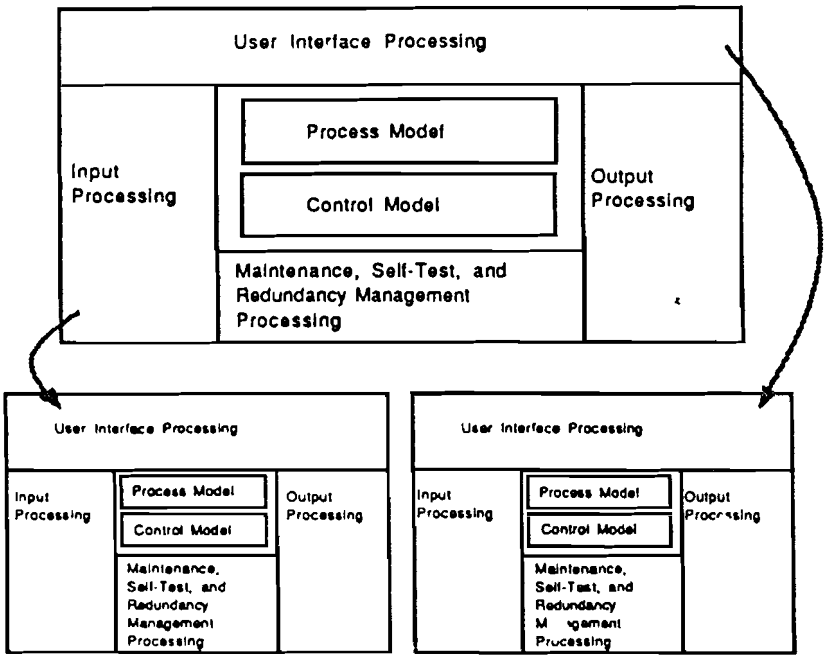

User Interface Processing
Input Processing
Process Model
Control Model
Maintenance, Self-Test, and Redundancy Management Processing
Output Processing

figure 12: architecture model layering

The user interface block is a special case of the input and output processing blocks. The environment access requires that we have specific interfaces between the system and its environment. It needs to be separated from the input and output blocks, because there are many special considerations, such as human factors, that affect the definition of the user interface.

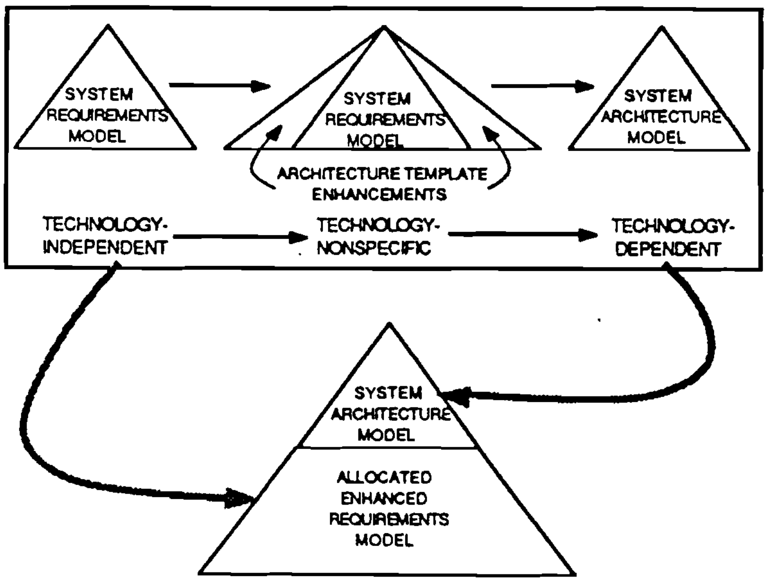

SYSTEM REQUIREMENTS MODEL
SYSTEM REQUIREMENTS MODEL
SYSTEM ARCHITECTURE MODEL
ARCHITECTURE TEMPLATE ENHANCEMENTS
TECHNOLOGY-INDEPENDENT
TECHNOLOGY-NONSPECIFIC
TECHNOLOGY-DEPENDENT
SYSTEM ARCHITECTURE MODEL
ALLOCATED ENHANCED REQUIREMENTS MODEL

figure 13: architecture developing process

Suppose we have four processes described in one **CFD** and one **CSPEC** (one finite state machine). The allocation process resulted in allocating process 1 and 2 to architecture module P and process 3 and 4 to architecture module Q, then the decision would have to be made about how to split the corresponding **CFD** and **CSPEC**. This depends entirely on the needs of the particular system and its architecture.

While allocating certain processes from the requirements model to the architecture model, it will be necessary to repartition the existing requirements. The architecture model is a hierarchical layering of modules that are defined by successive applications of the architecture template to each of the blocks in the model (see fig 12). Several template blocks won't be needed in chip design. The template is a tool.

We try to map the technology independent requirements model into a technology non specific physical model. The method is to insert a buffer between the requirements model core and the environment. (see fig 13). This is done to ensure that the essential requirements core is preserved wherever possible. Our reasons are:

- we want to be able to change the technology dependent buffer without changing the entire model, when interface technology changes;
- we want to be able to locate the system requirements and determine the impact when they change;
- we want to facilitate the task of testing the system's core requirements, both independent of and integrated with technology decisions.

2.4.3. ARCHITECTURE DIAGRAMS AND SPECIFICATIONS

A major component of the architecture model is the architecture diagram. An **architecture flow diagram** (AFD) is a network representation of a system's physical configuration. There is no predetermined relationship between the processes in the requirements model and their eventual place in the system architecture. The **AFD** is decomposed to show the next level of system architecture definition. The decomposition of the architecture model is not the same as the functional decomposition of the system. It is a decomposition in the sense that the architecture template is applied successively to each of the modules in the parent **AFD**. The template is used as an allocation guide at each level, but recall there is no requirement that every level of the system has modules in every block. *The newly defined data and control transformations should be specified and added to the requirements model.*

The highest level **AFD** is the **ACD**. **The architecture context diagram** (**ACD**) gives an overview of how the systems physically fits in its environment. The elements of the **ACD** are: one architecture module, representing the system; terminators that represents entities in the environment with which the system communicates, and information flow vectors that represent the communication taking place between the external entities in the environment.

The timing specifications from the requirements model are inherited by each architectural module affected by those requirements. The desired architecture must also meet the system's timing requirements. This is a consideration in the trade off process by which the system's implementation architecture is developed. The allocating procedure for the timing requirements is repeated through the architecure layers, until each module has a specification of the overall timing constraints allocated to it.

The AMSs can be written in a variety of ways; at the very least, it identifies requirements model components to show their allocation. An AMS consists of three parts: first, a brief narrative specification of what the module is required to do; second, a listing of any architectural requirements for the system and third a statement of the allocation of the requirements model component to the architecture module.

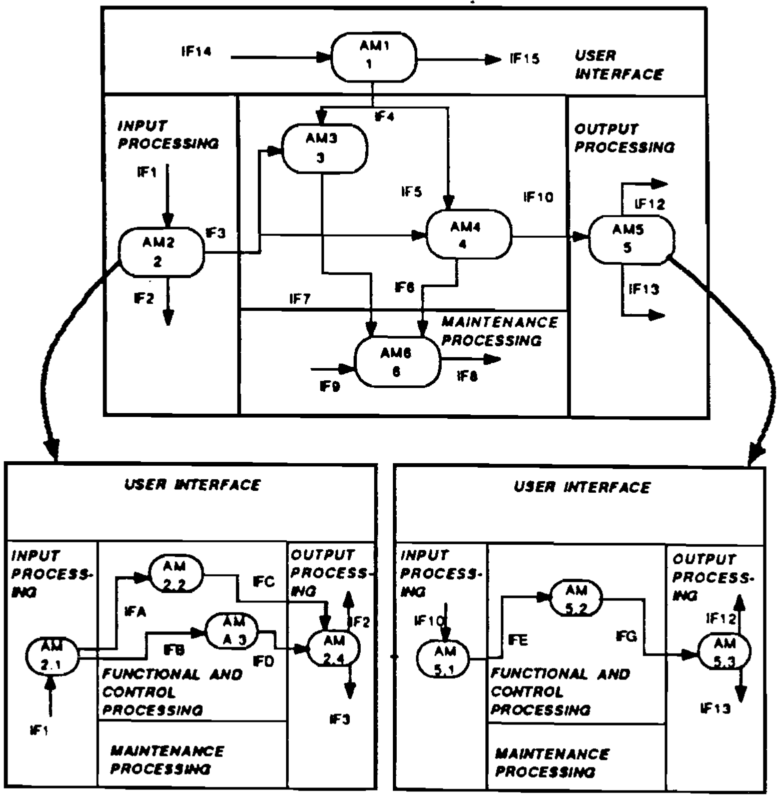


figure 14: layering

2.4.4. ALLOCATION TO HARDWARE AND SOFTWARE

Now the requirements model components are allocated to architecture modules, we will see how these architecture modules will accomplish their allocated tasks. Each module will be implemented in either hardware or software or a combination.

The completed architecture model with the hardware and software partitioning brings the system specification one step closer to implementation. Now we have system, hardware and software requirements and system, hardware and software architecture.

2.5. the mapping process and its problems

The biggest problem is mapping the data and control transformations on the architecture and into software and/or hardware. The heuristics of top-down allocation states that allocation to higher level implementation units should precede the allocation to lower level ones. Our implementation model consist of several parallel operating architecture modules. The bottom level modules operate sequentially.

We still have to implement in the architecture module the allocated processes. We classify the implementation process into three stages: the architecture module stage, the task stage and the program module stage. The overall organization of the implementation model is based on these stages.

The bottom level architecture module can carry out instructions (actions) and store data. The architecture module stage is represented as a network of architecture modules to which the requirements model has been allocated using the architecture model. Because the unit provides its own execution and storage resources, true concurrency of operation is possible within this stage.

A task is a set of instructions (actions), that is manipulated (started, stopped, interrupted and resumed) as an unit by the hardware and software implementation of synchronization primitives. The task stage is represented as a set of tasks networks, one network per architecture module. If required the task can be implemented sharing resources to simulate concurrency.

A program module is a set of instruction (actions) that is activated as an unit by the control logic within the task in a mutually exclusive fashion (only one program module may be active at a time). The program module stage is represented as a set of module hierarchies, one hierarchy per task. The program modules are function, procedures, subprograms. But it won't be usual, that an architecture module executes a procedure, since the problems don't require it. That kind of problems can be easier implemented on a general purpose CPU. The method for designing program modules is called structured design [Yourdon et al, 1978].

Classification at the architecture module stage will typically involve physical characteristics of the hardware (available arithmetic and logical units, special arithmetic logic, accessibility of data, execution speed, proximity to the physical environment of the system and so on). Task level classification may be made on the basis of activation and timing aspects of a task. Classification at the program module level is more likely to focus on logical or mathematical characteristics.

2.5.1. ARCHITECTURE MODULE ALLOCATION

It is possible that a primitive process appears several times on a **DFD**. This transformation can be implemented just once. Every time an activation event occurs, the process is started. The request for execution can be generated in several architecture modules. The transformation is called a **shared process** and the phenomenon shared processing. The heuristic of data abstractions from the requirements model states that the essential model should be searched for common elements, that will identify potential implementation resources. The goal is effective resource utilization.

In a multi architecture module allocation, it is possible for the allocation criteria determined by the processing characteristics to conflict with the essential model structure. *The allocation can introduce synchronization requirements* (example fig 15,16; a train tunnel with one track; decentralized control using two modules with a semaphore store; above the bar the event and below the actions). The casts and benefits of the effective use of shared processing resources versus essential model distortion must be taken into account. Allocation should be regarded as an interactive process, that allows the visualization and evaluation of alternatives. The identification of the architecture modules is a design aspect, which can't be captured with rules and guidelines.

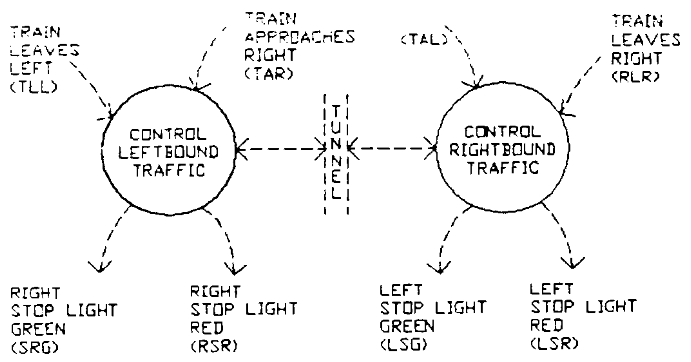

TRAIN LEAVES LEFT (TLL)
TRAIN APPROACHES RIGHT (TAR)
(TAL)
TRAIN LEAVES RIGHT (RLR)
CONTROL LEFTBOUND TRAFFIC
TUNNEL
CONTROL RIGHTBOUND TRAFFIC
RIGHT STOP LIGHT GREEN (SRG)
RIGHT STOP LIGHT RED (RSR)
LEFT STOP LIGHT GREEN (LSG)
LEFT STOP LIGHT RED (LSR)

figure 15: synchronization example [Ward,1985]

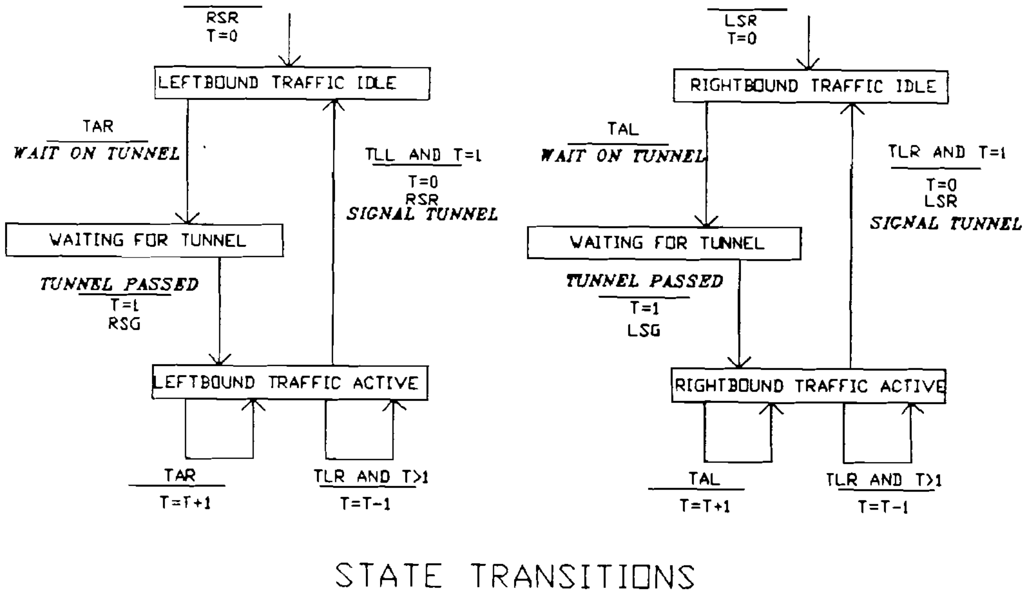


figure 16: state transition of the control model

The allocation of an entire transformation to a architecture module means that:

- the data of the input flows must be captured and provided to the logic that carries out the transformation;
- the output data must be sent to the appropriate destination;
- the work described by the transformation must be carried out by the logic;
- data corresponding to connections between the transformations and store must be retrieved or placed in storage by the module.

The allocation becomes more complex, when a low-level transformation must be split between modules. To split a transformation it is necessary to represent the work done by one part to the other part. The representation may be in terms of data already manipulated by the other part or in terms of flags representing the outcome of logic performed by the other part of the specification. [Ward,1985] volume 3 describes a splitting procedure of a data transformation. The splitting requires a control store and sometimes even semaphore operations. We need synchronization primitives to describe the result.

The stored data model allocation can give problems (shared use). There are a number of possible mechanisms for making a collection of stored data available to two or more architecture modules. In the case of sharing, a physical data storage mechanism (such as a multiport memory) is accessible to the modules. Although some synchronization is necessary (to prevent modules from accessing the same item together) the modules are equal in their ownership of the data. In the case of exclusive ownership, only one module has access to the physical data store. The other module can obtain or modify data only with the cooperation of two processes, that transfer and receive data. This means that if a transformation does its work in one module, but needs data owned by another module, the process that actually accesses the store must be allocated to the other module. In the case of duplication, each module has access to a data store containing a copy of the data. A module can only change stored data with the cooperation of the other module, since consistency of the duplicated data must be maintained.

2.5.2. TASK MODELING

The architecture model provides a specification of the transformations and stored data assigned to each module used to implement a system. The portion of the requirements model assigned to an architecture module still has no internal organization. Now we will reorganize and elaborate the model to adapt it to the internal organization of each architecture module. We discuss the application of the implementation modeling heuristics to construct a model, that describes the organization of data and control transformation tasks within a sequential module (bottom level architecture module). We don't differentiate between a data or control task, because it doesn't exist at this level.

An allocation to architecture modules doesn't change its contents of a requirements model, but simply redistributes the content. Neither does the architecture module allocation centralize control nor restrict any potential concurrency described by the model. The number of control transformations is at least as large as in the requirements model. Any potentially concurrent transformation remains potentially concurrent in the allocated model. Allocating portions of a model to a single task changes this situation, since control must then be centralized and concurrency is prohibited. Task allocation will therefore involve modifications of the model to combine control transformations and their associated state diagrams, and to impose sequences on potentially concurrent data transformations, ([Ward,1985] volume 3). We discuss several situations requiring task allocation.

The control transformations in several modules have to be combined to execute the tasks (transformations) sequentially. The combination of state diagrams centralizes control, but does not remove concurrency of these tasks. The concurrently enabling of transformations has to be sequenced. This is a design decision. The limitations on the number of tasks, that may be scheduled to run and the costs of switching between the tasks may lead to group several transformations together in one task.

A problem arises when a task is a continuous transformation and has to share the module with other tasks. The technique used is to allocate the continuously operating transformations to a task that runs periodically. The task samples the continuously available data, processes the data to produce outputs, and is then disabled, to be enabled again after a fixed interval. The sampling rate and therefore the interval can be computed for each transformation by examining the environmental constraints.

The situation becomes more complex, when several continuous transformations are allocated to a single periodic task, and the enabling and disabling actions are asynchronous. Each transformation is to be enabled or disabled independently. Therefore the transformations control must be implemented by enabling and disabling the entire task. We introduce flags to represent control information, that states which portion has to run. A task implementing the control transformation is responsible for setting and resetting the flags, which are an implementation of the enabling and disabling logic.

Tasks incorporating transformations of discrete data must run, when the data becomes available. We have an adequate implementation of a discrete transformation, if the time needed to process an input value is within response time constraints. The inputs should arrive at intervals longer than the time needed to process input values. If this is not true, then the average interval between the arrival of inputs should be less than the average time needed to process an input.

In the first case, a task containing discrete transformations executes until it requires input, at which point it must go idle or pass control back to the module control and enter the **waiting for input** state. Alternatively the task may idle until a flag indicating that the data is present, is set.

If the data arrives more rapidly than the task can transform data, a queuing mechanism can be introduced. The transformations perform the essential task and the task that queues the arriving input (a queuing task).

Scheduling can be necessary to use the hardware efficiently. When a task waits for the occurrence of an event, it can be suspended and another task could be activated. Another possibility is that a module has to execute several tasks depending on external events. The following figures contain a few scheduling examples of tasks allocations to one module.

```
task A  ---000---000---000---000---000---
task B  000---000000000---000000000---000

                                          ——→ time
```

figure 17: two tasks (a dashed line indicates activity)

```
task A ---00000000000000---00000000000000---0000000000
task B 000---00000000000000---00000000000000---0000000
task C 000000---00000000000000---00000000000000---0000
task D 000000000---0000000000000000000000000000000000
task E 0000000000000000000000000000000---0000000000000

                                          ——→ time
```

figure 18: several tasks

A more complicated scheme is necessary for the next example. Task A needs thirty units of execution time. Task B needs 5. Task B should be executed every ten units. Task A should be executed every 90 units. If we could split up task A, we could use the scheme in figure 19.

```
task A 00--00--00--00--00--00--0000000000--00--000
task B --00--00--00--00--00--00--00--00--00--00--0

                                          ——→ time
```

figure 19: scheduling scheme

Because all the scheduling schemes are known on forehand, we can implement them in hardware. A problem arises when we have to wait for an event, which occurs at an unexpected moment and we have to meet the timing requirements. If those requirement are so strict, that we have to interrupt a running task, we need a context switch. Since a context switch requires saving the processing status, we should try to avoid this situation by allocating the processes to another module. We can also try to reprogram the other tasks, to allow them to be interrupted at a convenient moment without having the trouble of a context switch. For scheduling techniques see ([Ward,1985] volume 3) and [Janson,1985]. *Since these processes are hard to implement in state machines, we should if possible implement parallel hardware.*

Task allocation is an activity similar to architecture module allocation, in that it involves reorganization around a set of chosen modules for implementation. However, *the requirements that tasks be sequential, have centralized control, operate discretely, model data arrival rates and maintains stored data integrity, may require substantial modifications to the model used as input to the allocation process.*

2.5.3. INTERFACE MODELING

The allocation of transformations to architecture modules and tasks tends to fragment the transformations and to introduce additional inter module and inter task data flows and stores. The interfaces between modules and tasks must therefore be examined for potential problems involving data exchange, stored data, access and control synchronization. The requirements model only showed the net flow of data. We must drop this now. If inter task flows must be sequenced, the cooperating control interactions between the tasks should be modeled.

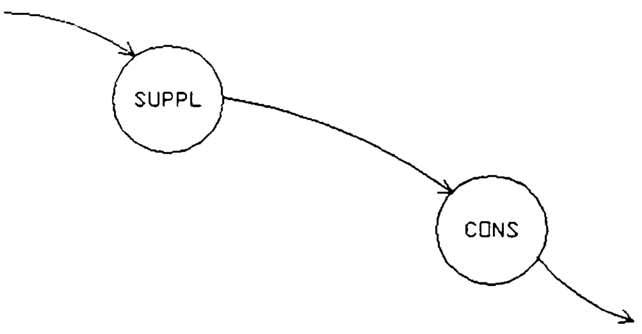


figure 20: DFD requirements model: two processes, one supplier and one consumer

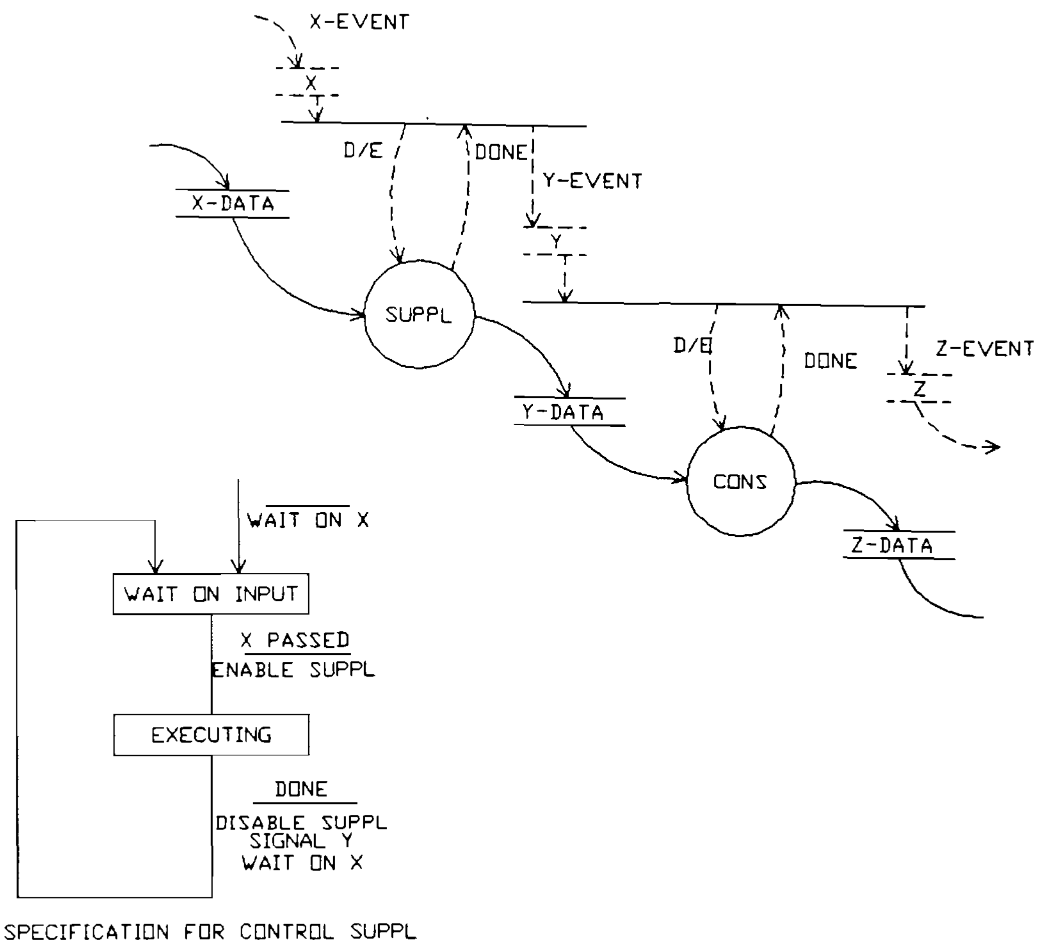


figure 21: solution for the communication tasks (semaphores are associated with the control stores) [Ward,1985]

We only discuss synchronization, because communication is a data flow accompanied with synchronization control. We will talk about task synchronization. A task is executed in an architecture module, so tasks communicate and not architecture modules. Tasks can be active at the same moment or at various points in time. We focusses on inter-task interfaces independently of whether the tasks are in the same or in different architecture modules.

SYNCHRONIZATION OF DATA FLOW

A general problem is the data flow synchronization between two tasks, of which one operates on the data of the other. How do tasks interact having data flow in common? These tasks have to be activated and deactivated at the right moment. We model data flows consisting of two components: the data itself and an associated trigger that signals the presence of data.

The two transformations can be synchronized in several ways. The second receiving task could be executed after the first. They can both be executed at the same time using pipelining. Even a store can be introduced to store the result of the first transformation (fig 21,22). The second task could be activated several times in succession until the store is empty.

SYNCHRONIZATION OF ACCESS TO STORED DATA

A fundamental requirement for implementing a queue for communication is that the reading and writing of the smallest transferrable data unit to or from the queue once initiated is uninterrupted. The other fundamental requirement is that no tasks have access together. Tasks in one architecture module though can never access a store at the same moment. Only the action can be interrupted. The read and write actions should be atomic.

Another problem is that a task can attempt to read data, which has not yet been produced. We introduce a flag to signal, if data is ready. A few solutions for the access problem will be discussed.

A solution is to make a task uninterruptable. The writing of data becomes an atomic operation by making the sections of the code (called critical sections), that write the data items uninterruptable. If this is impractical, a flag may be used to lock the data items. All tasks that share the data, must test the flag before using the data. The reset and set operations must be atomic actions. This solution can be used with tasks in the same or in different modules.

The implementation of an atomic actions for cooperating modules is difficult, because the partners can try to access the store at the same instant. Some unit/program has to arbitrate between the together arriving requests and avoid starvation. Starvation is the situation, where other requesters always lock out one requester for the access to the shared data. In the other situation only one task is active in the same module and this task should not be interrupted, so the atomic actions is easily implemented.

A second solution is the introduction of an intermediate task, a data server. Requests to read and write data are given to a task, that owns the data Since a task can by definition, carry out only one operations at time, each operation on the data can be guaranteed to be atomic.

SYNCHRONIZATION OF CONTROL

The sequencing and control of data that crosses the interface of the modules, may be modeled using two interlocked control transformations.

This causes no problems as long as both task are active at the same moment.

First we describe the interlocked control transformations to show the problem [Ward,1985]. Task A produces an event flow R, when P occurs. Depending on the state of task B an S event flow may or may not be produced. Task A responds differently to subsequent flow Q depending on whether an intervening S has been received. In the requirements model, the arrival of a flow instantaneously triggers all the transformations, that operate on that flow and all subsequent transformations with discrete flow connections to the first. In an implementation, task B may be suspended to allow A to run and task A may therefor respond to an Q event flow before it can receive an S. If task B is in the other module, the task can be inactive, since another task is running. This problem occurs because of the chosen implementation.

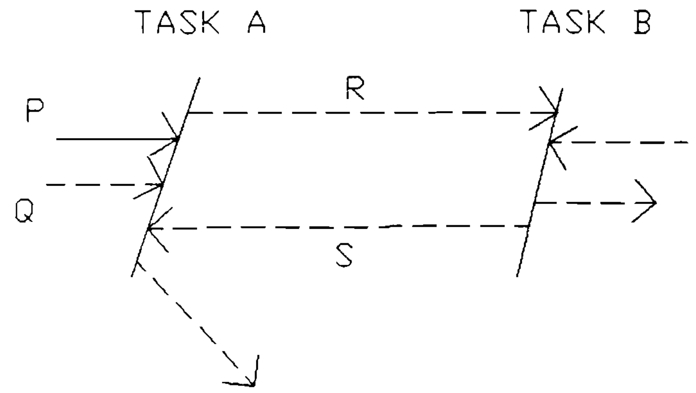


figure 23: two communicating control transformations

Three solutions are introduced. The first is only possible when the tasks reside in the same module. Task B may run with a higher priority if it must always finish its activity, before task A runs. This may not be possible, if the S event flow then causes another event flow to be produced, that then must be completely processed (possibly by task B).

Second, the control transformation specifications may be modified to send and receive positive confirmations of the communicating event flow. We would add a control store with semaphores operations for Q and an event flow from task B, whose interpretation is "**there is no S**" and modify the specifications.

Third, the transformations may be combined into a single task. This may be cumbersome. We will usually use solution two, which is easier to implement then a complex scheduler with priorities. Besides, we usually have inter module communication because of the requirements model.

2.6. summary and conclusions

The technique used to find an implementation of a digital system is to generate a requirements model and an architecture model using guidelines of implementation. The requirements model describes only required synchronization and communication for the problem definition. In developing the architecture model, we redefine these descriptions and introduce where necessary synchronization and communication primitives for the introduced partitioning. At each level we only concentrate at the interaction in that level. We hide the lower level synchronization. While refining the model we are introducing the implementation.

When implementing the requirement model on architecture modules, an effective implementation strategy requires careful separation of subject-matter-specific details from details that can be handled by general modules. To attend this goal, it is important to be able to visualize such a module as a system in its own right. Shared processes which several architecture modules want to execute, are separated from the subject-matter-specific architecture module and allocated to a separate module.

The requirements and architecture model consists of data and control flows, date and control stores, data and control transformations. Transformations share data (resources and signalling, control information). requiring mutual exclusion. Events start data and control transformations (activation and deactivation), but we had to schedule tasks to avoid starvation and to meet the timing requirements.

The data and control transformations are tasks in the implementation, where we don't see any difference between a control and data transformation, executing on the same hardware, if they are allocated in the same architecture module. Data processing tasks require a control and processing (datapath) unit.

The timing requirements can require coprocessing or pipelining. Coprocessing is a form of shared processing, where a task is allocated to another private architecture module and executed after a request. Pipelining needs parallel operating architecture modules.

Just mapping every control transformation into finite state machines and every data transformation into a datapath and a control unit, would be the simplest implementation and would guarantee a maximum of parallelism. The basic idea is that for control transformations separate finite state machine can be built. This can cause an inefficient use of our resources. The data transformations aren't all active at the same moment, so we can allocate several transformations to a module without reducing the concurrency. We try to optimize the resources usage, but this requires synchronization and communication.

We only need some simple synchronization primitives for task and architecture module synchronization and communication, a few primitives to control shared resource use, shared data and shared processing and to execute process management. The corresponding control transformations are implemented on parallel operating modules called scheduler, arbiter, resource manager and server.

3. SOFTWARE SOLUTIONS TRANSLATABLE INTO HARDWARE?

3.1. introduction

Parallel computer processes are a consequence of the nature of the problem (e.g. real-time and operating systems). The interactions of a set of parallel programs cause difficulties. Algorithms for parallel processes solve these problems and their main objective is mutual exclusion and synchronization.

Several algorithm have been proposed for synchronization and mutual exclusion [Andre,1985], [Raynal,1985]. Programming languages were introduced to solve the problem, especially for operating system and real-time applications. These languages introduce structuring the design and checking access to shared data at compilation time by defining data types for the synchronization. [Brinch Hansen,1975] [Brinch Hansen,1977], [Hoare,1978], [Kylstra,1984]

But basically all these programs have to be executed using low level primitives to control the hardware. We will study these primitives in this chapter using [Janson,1985].

3.2. operating systems and process management

An operating system implements the required low level primitives for synchronization and communication. The two main operating system objectives of interest to us are: multiplexing the system's resources among all processes (users) and allowing these processes to share information. In the context of processor management, the objectives mean multiplexing the physical processors among users and allowing user computations to interact with one another. In the following discussion users can be replaced by processes and physical processors by architecture modules.

Operating system consist of programs, which take care of the orderly sharing, multiplexing and protection of resources [Janson,1985], [Joseph,1985]. The control of sharing, multiplexing and protection is based on maintenance of detailed status information about resource use.

Physical processors are multiplexed over time. Time slots are allocated to each process. From a macroscopic point of view, each user has the impression that he executes on a processor of his own in parallel to other users. This form of processor multiplexing is called multiprogramming. As a result of multiprogramming, each user sees a virtual processor. Multiprocessing covers all issues related to communication and concurrency control among parallel virtual processors (i.e. parallel or pseudo parallel physical processors).

We study some operating system's processor management features to see, if we can translate software solutions into hardware. The problems are identical except that the pseudo concurrency introduces problems, we usually won't encounter in VLSI design.

3.3. mutual exclusion

A first multiprocessing problem is the need for mutual exclusion. The sections of code in which process 1 writes into B (variable) and process 2 reads from B are called critical sections, meaning that they are mutually exclusive. Only one process can be executing its critical section at any time. One mechanism to enforce mutual exclusion consists of using Dekker's primitives. They are a purely software primitive and the drawback is the busy-waiting of a processor locked out of its critical section. Processor cycles are wasted testing flags over and over again instead of allowing inactivity until the critical section can be entered.

In monoprocessor systems mutual exclusion can be achieved by inhibiting or masking interrupts. During the execution of a critical section all interrupts are disabled so guaranteeing the exclusive access. In a multiprocessor system masking the interrupts on one physical processors cannot prevent a virtual processor running on another physical processor from entering its critical section.

In multiprocessor systems, the most basic mutual exclusion mechanism is based on hardware instructions called locks or test and set (TAS). A TAS primitive takes three parameters: a test value, a set value and the address of a word in the memory [Janson,1985]. Interrupts are disabled from the start of the test operation to the end of the set operation. Locking suffers from the busy-waiting problem. The user programs need an ability to stop, when encountering a locked critical section (scheduling, synchronization). In system programs a lock instruction is used, when the busy waiting time can be kept short.

Mutual exclusion is essential in concurrent systems. So we will need to implement primitives for mutual exclusion. The implementation will be based on the mentioned methods. Only the TAS operation seems too complicated for our purpose. We won't need a memory value to test, but just one bit (free or occupied). We will need an arbiter to keep track of the requests and to give every request a fair chance.

3.4. serialization

The second multiprocessing problem is serialization. We will illustrate the problem using an example [Janson,1985]. Processor P1 waits to transfer money from an account A to an account b, while process P2 is trying to compute the sum of A and b. Clearly P1 and P2 will want to lock A and b, to guarantee mutual exclusion while using them. Imagine that P1 locks a, releases it and locks b. P2 locks a, releases it and locks b, which can give the wrong result. What we want is called serialization, i.e. that P1 and P2 while operating in parallel, preserve the illusion that they operate in series. The solution is called two phase locking. Two-phases locking states that a processor wanting to preserve the illusion of serial access must hold all locks it acquires, until it releases any of them.

Although serialization is a serious problem, it can't be solved in hardware. It should be solved in the process description using synchronization and mutual exclusion primitives.

3.5. synchronization

Two parallel operating processes need synchronization (e.g. a process has to read the result of another process and it needs to know when a new result is ready). A mechanism should enable processes to communicate and synchronize themselves, so they will know when to wait and when to restart. We discuss synchronization techniques.

3.5.1. SEMAPHORES

The **P** and V primitives are designed to manage variables called semaphores. A semaphores is a cell of memory, that resides inside the processor management mechanism and is accessed only by the primitives for handling semaphores. The **P(S)** primitive works like a lock instruction. It tests semaphore S and sets it to 0 if it now equals 1. Otherwise it puts the name of the process in a waiting list associated with S and stops it. An execution of the V(S) primitive would then restart the stopped processor and reset the semaphore to its initial value.

The implementation of semaphores primitives is based on the use of locks and interrupt masking inside the processor management mechanism. Problems will arise, if two processors invoked the primitive at the same time. **P** and V primitives are also critical sections. They are protected by a common lock. The interrupts are masked between locking and unlocking time. Semaphores eliminate the problem of busy-waiting, a processor trying to enter a critical section when it should not, is stopped.

The semaphore concept can be generalized. Suppose we have a buffer holding several messages and several processes produce and read messages. Counting semaphores can be used to coordinate all processes. The semaphore equals the number of messages in the buffer. A V operation increases the semaphore by one. A **P** operation increases the semaphores by one, if that semaphores is strictly positive.

The main disadvantage of semaphores in operating systems implementation as synchronization primitives is the number of variables they require which are stored in primary memory. The number depends on the programs in the system.

In application specific VLSI design the number of semaphores is known, so they can be implemented in hardware. We should also solve the critical sections problem of the V and **P** operators and the danger of executing two **P** and V operations together. We will have to implement a list of processes, but in dedicated hardware design the list is limited. If it gets too complex, we can always use a queue.

3.5.2. BLOCK AND WAKEUP

The BLOCK(B) and WAKEUP(W) primitives, which operate on the virtual processors solve the problem that semaphores present in operating systems. They require only one bit per virtual processor. Since the maximum number of virtual processors is always limited in practice, the number of bits required is also limited and known in advance. The B primitive simply stops the process, that invokes it and puts it in a global list of waiting processors. B and W are mutually exclusive. They are executed under a lock to protect the integrity of the list of blocked processes.

In the case of semaphores, waiting was associated with a particular event. Here processors do not wait for events, they just wait and expect their partners to wake them up. These partners have to know one another's identity as opposed to knowing semaphore names.

A problem is to ensure that the event for which the process is blocked, has occurred and not an event for that process for which it will be blocked later.

Because in VLSI design we don't have the disadvantage that semaphores require too much memory and block and wakeup require extra checks as mentioned, we choose the semaphore implementation for the synchronization.

3.5.3. HIGH LEVEL MECHANISM: MONITOR

The discussed primitives are of a very basic level in the sense, that processors programmed by the users are responsible for the correct usage. Failure to properly use the primitives may result in loss or destruction of shared data or inconsistency in shared data. To solve these problems several mechanism of a higher level have been proposed. These mechanisms have been invented in the context of parallel programming languages. The realization of these high-level synchronization mechanism uses the low-level primitives. [Raynal,1986], [Andre,1985], [Hoare,1976]

Monitors are a synchronization mechanism based on abstract data types. With abstract data typing, variables can not be manipulated directly. Instead, variables of a given type may be manipulated only through primitives provided for that effect. A monitor also specifies the necessary synchronization conditions for accessing data. The primitives composing the monitor are mutually exclusive. Synchronization signals are exchanged inside the monitor by the processors, that successively penetrate the fence, through some primitive signalling mechanism (such as a semaphore). So a monitor can cause the processors to be queued at various places, waiting for different events, once they invoke the monitor and enter it. [Hoare,1974] In the original definition of monitors synchronization is achieved through the use of signal and wait primitives. An important issue is the fair scheduling of waiting processes.

The implementation in an architecture module would require queues, arbiters, semaphores and a program to manipulate data. The main advantage of monitors is that they are an abstraction, which has been designed to be safely shared between processes, as it can be accessed only in a disciplined way and which removes the problem from the user, who only has to declare some data as a monitor! But a monitor isn't a suited solution for our problems. We will use the concept of disciplined access and a separate controller for a resource manager (an architecture module), which controls the access to the shared resource. Deadlock can be fought using prevention, avoidance or detection. [Janson, 1985]. But in applications specific VLSI we know on forehand if deadlock can occur and we can avoid it using a software synchronization technique.

3.6. scheduling/dispatching

Now we have discussed the part of the processor management mechanism dealing with multiprocessing issues, we will have a brief discussion of how to multiprogam one or a small number of physical processors to support the abstraction of a multiplicity of virtual processors. The task that schedules the virtual processors is called **dispatcher.** We are interested in a solution for scheduling a few tasks in an architecture module depending on external events to start the program or to run a few tasks having timing constraints.

Multiprogramming consists of multiplexing physical processor among virtual processors.

Two possibilities exists for stopping a virtual processor:

- a processor may be stopped when it asks to be stopped i.e. when it calls a primitive P, Block or when it is finished
- a virtual processor may be forced to stop if it has had a physical processor for a long time (clock interrupt) or if an event occurs that demands the attention

The state information must be saved when a physical process switches virtual processors. Processor states are saved and stored in a table VPT (virtual processor table)

The size of the VPT is limited, because it has to remain in primary memory. The algorithms to execute scheduling and dispatching are discussed in [Janson, 1985] and [Kylstra, 1984].

Some discussed techniques are too complex, because of the requirements of an operating system. An operating system should handle unknown processes and events (character and number), should simulate concurrency for the user, should not allow busy-waiting and should optimize resource use and response time. The restrictions in hardware design are less demanding and so easier to fulfill.

The multiprogramming can be implemented in the architecture modules. It requires a process to keep track of the scheduling/dispatching. We need to save the lists of tasks and task status information (VPT). Fortunately we know which processes to schedule and the number of processes is limited. *We choose a process using algorithms, which try to fulfill every timing constraint. If we would allow processes to be interrupted at every point in their program, we have to save an enormous amount of information for the processing state.*

Simpler features are required after reduction of interrupt situations. We allow stopping the task execution only, when the task allows this itself. A task may only be interrupted, when we can't allocate that interrupt handling to another architecture module, or the task state doesn't have to be saved and after the interrupt handling the task will be resumed.

The number of processes is fixed (no creation of processes). If in an architecture module several processes are runnable, one will be selected. A primitive can put a process in a wait state and restart the execution of another process only under very special circumstances. Running processes can be busy-waiting. We have to decide, what to do if a process is waiting for an event. Because the possibilities are limited, the solution is easy to implement.

A clock interrupt should not occur, which simplifies the dispatching and reduces the amount of state information to save.

Because *the required scheduling is known and the number of processes is limited*, we can implement the scheduling algorithms in hardware. One feature of all the scheduling algorithms we will frequently use, is task priority. The only problem with using priorities is starvation. Since another process has always a higher priority at the scheduling moment, a process is never started.

3.7. conclusion

The problems encountered by the implementation of parallel operating architecture modules are solved in operating systems using software. Some solutions used in operating systems are implementable in hardware. Several techniques are too complicated to use or to implement, since more complex situations occur in operating systems then in hardware. The main difference is that the number and characteristics of the processes and events in a VLSI design are known. In dedicated hardware we have more information about the timing of events.

As in operating systems we define low level primitives for mutual exclusion, synchronization, scheduling and dispatching. For synchronization the semaphore primitive is used. The concepts of the monitor for shared resources is translated in a parallel operating unit to control the access to shared resources avoiding starvation.

Although we have primitives, the synchronization problems still remain. They are solved using parallel programming techniques to synchronize critical sections and to implement mutual exclusion for critical sections. Deadlock should be avoided.

The control implementations of mutual exclusion and synchronization primitives are special architecture modules (arbiters, resource managers, schedulers). The module will avoid starvation using a rotating priority scheme.

The scheduling/dispatching in the architecture module is solved with a dispatching algorithm. The context switch is avoided by restricting the blocking and interrupting of tasks. A task should be interrupted without having to save the datapath contents.

The dispatching variations are reduced. Avoiding the saving of the processor state reduces the size of the VPT. By restricting the interrupt of tasks to situations where only little status information has to be saved, reduces the size of a VPT. Only a few tasks are allocated to one module and the number is known. The status of these tasks has to be recorded. The VPT has only a few entries. So we don't need large tables. This simplifies the dispatcher.

If a requirements model needs a more sofisticated operating system feature, we can built that feature using hardware primitives and a parallel control transformation module.

4. SYNCHRONIZATION AND COMMUNICATION PRIMITIVES

4.1. introduction

When formulating **PSPEC** and **CSPEC** we encountered the problem of introducing synchronization. At higher levels in the model it is not even clear how the synchronization will be implemented. The **CSPECs** and **PSPECs** can contain synchronization and communication actions. When synchronization primitives activate or deactivate transformations, they describe control actions. But the event processing to generate a data condition (CF) is performed in data transformation. **CSPECs** only activate and deactivate processes and don't process any control or event flow. At every level one **CSPEC** coordinates the transformations. We are focussing on actions between the transformations, so the primitives are introduced in the **PSPEC**. During further transformation decomposition the synchronization primitives can be allocated to a **CSPEC**.

Primitives to capture the necessary synchronization and communication will be introduced in this chapter. After the synchronization requirements model we discuss the allocation problem of the synchronization execution at the bottom level. We restrict the primitives definition to the ones needed for describing synchronization in the requirements model trying to capture concurrency. Primitives for task synchronization in one sequential module will be discussed later.

For further discussion on synchronization, communication, communicating finite state machines and mutual exclusion problems see [Raynal,1986], [Andre,1985], [Dykstra] and other publications on these subjects. For protocols and protocol implementations in VLSI see [Haynes,1981] and [Mead, Conway,1980].

4.2. process synchronization

Process execution can be waiting for the occurrence of an event or for a condition getting true. This condition check is a control transformation. Figure 25 contains the requirements model for the condition scheduling of one condition identifier. CFs can be events or control signals generated after processing an event. The condition can be quite complex requiring information of several modules, timers, calculations, logical expressions. If data should be processed, a data/control transformation is introduced.

The **CSPEC** *is allocated to a separate module (*SAM, **scheduling architecture module***) to take care of the asynchronous events.* A required data/control transformation will also be allocated to the **SAM**. The process itself will be allocated to a **PAM (processing architecture module)**, where another process can run, if this process is not runnable. Another allocation is to transform the **CSPEC** into a task running in the **PAM**. But it is questionable, if the **PAM** is suited for the condition checking and it also removes concurrency. *For every condition a* SAM *will be implemented.*

A process signals another process when a part of the condition is satisfied. The process shouldn't signal the other process that it can start, because the process can't know, what the synchronization condition is. We define a primitive **SIGPROC(IDENT)** (signal process) to signal the event (**CF**):

SIGPROC	= signal process
IDENT	= schedule condition identifier;

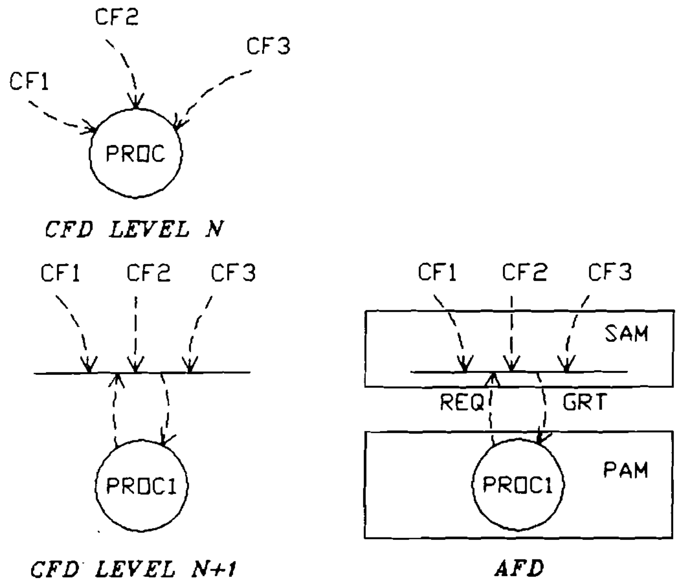


figure 25: model process synchronization

An identifier can refer to a schedule condition for several processes. The **SIGPROC** should be given directly after the action satisfying the constraint preventing that the execution of the other process has to wait unnecessarily.

The process waiting for the condition becoming true executes the primitive **WAITPROC(IDENT)** until it receives a signal to continue, depending on the condition to which the identifier refers.

```
WAITPROC(IDENT): REPEAT SKIP UNTIL COND;
                 CONDITION:= FALSE;
```

An implementation using a handshake protocol:

```
REQ:=TRUE;
REPEAT SKIP UNTIL GRT;
REQ:=FALSE;
```

If for example a semaphore is used, the primitives are simplified to:

```
SIGPROC(IDENT)  :ID:=ID+1;
WAITPROC(IDENT) :REPEAT SKIP UNTIL ID=0;
                 ID:=ID-1;
```

Both actions (increment and decrement) are executed in the **SAM** introducing a data transformation. The implementation of **SIGPROC** and **WAITPROC** are signals, which can be tested by the modules. The **monitor**, a control process of the processing module tests all grant signals for allocated processes and selects a process. *A monitor can receive several signals at the same moment, so a priority scheme selects the process to start. The scheme should prevent starvation.* After process execution the monitor selects a new process.

Appendix B contains implementations of the SAM using several techniques for process synchronization.

4.3. request/acknowledge synchronization

Request/acknowledge synchronization existing only between two partners is used to request something from another process (coprocessing) or to signal the partner to take action for some event (e.g. pipelining). (see fig 26).

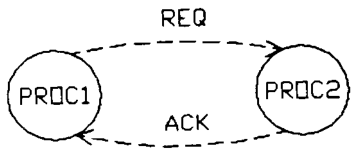


figure 26: requirements model for R/A synchronization

The requesting process expects an acknowledge from the requested process, after the action has been performed, so that it can generate a new request. We define the following primitives:

REQ(IDENT)	: sends a request to the partner (request)
WREQ(IDENT)	: the partner waits for a request (wait for request)
ACK(IDENT)	: sends a acknowledge to the requesting partner (acknowledge)
WACK(IDENT)	: the requesting partner waits for an acknowledge (wait for acknowledge)

The request and acknowledge combination of two processes has an identifier (IDENT).

```
PSPEC 1:          PSPEC 2:
----              ----
----              ----
REQ(ID);          WREQ(ID);
----              ----
----              ----
----              ----
WACK(ID);         ACK(ID);
----              ----
----              ----
```

figure 27: process specifications

The process specifications (figure 27) contain also the actions on the dataflows between the processes. If both processes are allocate to different architecture modules, a real parallel execution is possible, otherwise the **PSPECs** have to be rewritten, since the processes are executed after each other. We will use other primitives.

An implementation description using a boolean SIG is :

REQ(IDENT)	SIG:=TRUE;
WREQ(IDENT)	REPEAT SKIP UNTIL SIG;
ACK(IDENT)	SIG:=FALSE;
WACK(IDENT)	REPEAT SKIP UNTIL NOT(SIG);

4.4. mutual exclusion and correct synchronization

We should protect programs against wrong mutual use (shared variables). Mutual use of resources occurs when several processes want to access the same information (variable) or when a process wants to give information to another process or several other processes. Parameter communication is a mutual exclusion problem. Several processes access the identical variable. Parameters can be located in one resource. Communicating processes consisting only of critical sections (ie the program can't release the resource during the total execution time) should copy the parameters. The copying is a mutual exclusion action.

Mutual exclusion doesn't guarantee, that the contents of the resource is correct. The programmer should synchronize mutual exclusion resources using process or R/A synchronization, when the processes have to access the resources in a sequence.

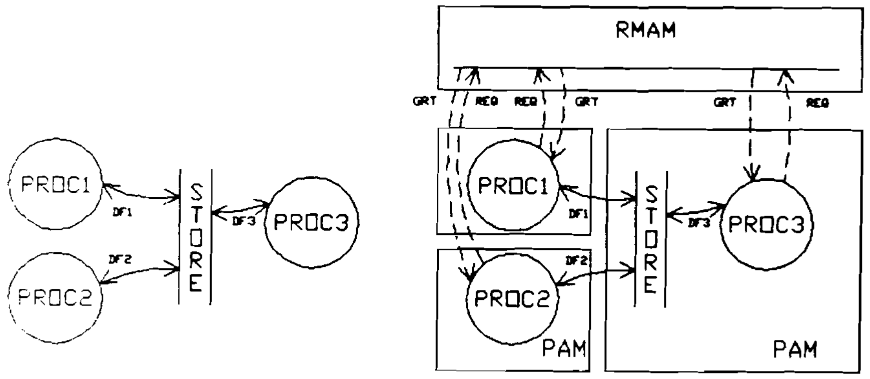


figure 28: model of mutual exclusion (RMAM= resource manager architecture module)

Figure 28 describes the requirements model introducing the problem of mutual exclusion. *Only after the allocation of the processes to separate architecture modules the mutual exclusion problem occurs.* In the requirements model we already try to describe the problem in terms of concurrency and describe mutual exclusion. Processes allocated at the bottom level to one architecture module can't be active at the same moment and so can't access the resource together. The process synchronization in that module should guarantee the correct sequencing of the access.

To indicate sharing of resources we introduce as early as possible the protection for the resource. We define mutual exclusion primitives:

WAITRES(ID):	REQ:=TRUE;
	REPEAT SKIP UNTIL GRT;
	(wait for grant access resource)

RELRES(ID):	REQ:=FALSE; (release resource)

ID is an identifier for the resource or the group of resources.

A weaker form of mutual exclusion allows all requesters to read a variable, but for changing the value only one process gets access. An implementation is setting a bit, which is tested before writing and reading. If more requests are generated at the same instant, conflicts arise. We introduce the following primitives:

WRDRES(ID):	WAIT UNTIL ALLOWED TO READ (wait for read access resource)
WWRRES(ID):	WAIT UNTIL ALLOWED TO WRITE (wait for wait access resource)
RLRDRES(ID):	RELEASE RESOURCE AFTER READ (release resource after read action)
RLWRRES(ID):	RELEASE RESOURCE AFTER WRITE (release resource after write action)

A read is allowed, when no process is writing or reading. All waiting READ requests are granted together. Only one process may write at the same moment. Starvation may not occur and reading and writing should be handled with the same priority.

The duration of the resource locking depends on the program. To avoid too much overhead by requesting several resources after each other, it is allowed to request together a group of resources. This technique also solves the serialization problems.

```
PSPEC;
----
WAITRES(ID);
-----
MANIPULATE RESOURCE
----
RELRES(ID);
-----
```

figure 29: process specification using mutual exclusion

The primitive execution introduces critical sections, since several processes can try to access the resources at the same moment. If the process sequence accessing the resource is known, a simple solution is process synchronization deciding whose turn it is.

We introduce a module, an arbiter granting the access. This arbiter is a parallel process, since requests can arrive at every moment. The allocation model introduces the **resource manager module (RMAM)** (see fig 28). *The* **RMAM** *implements the identifier relations and the types of request (single access or read/write access).* The arbiter tests the signals asking access and grants it to an architecture module. It takes a while before a request is acknowledged, checking all requests sequentially. A solution is a test on all signals, requiring priorities or conflict resolving request schemes.

The resource manager is a control transformation, whose goal is to avoid starvation making the implementation of the transformation difficult, needing a rotating priority scheme.

The **CSPECs** of the resource manager aren't easy to formulate. A finite state description can be very complex, if several (more then two) requesters exist. We formulate the **CSPEC** as follows: select the request, which has the highest priority and give that request after granting the lowest priority. For every set of resource identified by one identifier a manager is used. Only if identifiers have a relation, they are allocated to one **RMAM** (e.g. 3 resources, sometimes they are all requested together, or just one, or two independently, whose access may be granted together. These relations are implemented in the resource manager; see appendix B).

Locking the resource doesn't assure correct operation of the program, since the access sequence of the resources isn't synchronized. The following examples illustrate that mutual exclusion doesn't guarantee correct program execution. We will indicate a few solutions. In the literature several techniques are introduced for these problems. They also discuss the mutual exclusion access synchronization for more than two processes. [Raynal,1986], [Andre,1985]

```
PROC P;                          PROC Q;
REPEAT ....                      REPEAT ....
     BEGIN                         BEGIN
        (* CS *)                       (* CS *)
        WAITRES(IDENT);                WAITRES(IDENT);
        -----                          -----
        RESA:=                             :=F(RESA,..);
        -----                          -----
        RELRES(IDENT);                 RELRES(IDENT);
        (* NCS *)                      (* NCS *)
        ----                           ----
     END;                          END;
```

figure 30: example 1; CS is critical section; NCS is non critical section.

Process Q in example 1 should use the information prepared by PROC P in resource RESA. It can process the identical information of RESA several times, while it expects new data. This is caused, when PROC P isn't activated. The solution is using req/ack synchronization and the mutual exclusion primitives can be dropped, if no other process has access to the resources.

```
PROC P;                          PROC Q;
(* INIT ACK *)                   (* INIT REQ *)
REPEAT ....                      REPEAT ....
     BEGIN                         BEGIN
        (* CS *)                       (* CS *)
        WREQ(ID);                      WACK(ID);
        -----                          -----
        RESA:=                             :=F(RESA,..);
        -----                          -----
        ACK(ID);                       REQ(ID);
        (* NCS *)                      (* NCS *)
        ----                           ----
     END;                          END;
```

figure 31: solution for example 1 using R/A synchronization

By initializing the request and acknowledge stores correctly the critical sections are executed alternately.

When two communicating processes consist of only critical sections, the information in the identical resource has to be copied, so that the critical sections are shorted and non critical sections are introduced. The programs change into (RESA is only changed in the CS of PROC P during that period PROC Q can't copy RESA):

```
PROC P;                              PROC Q;
(* INIT ACK *)                       (* INIT REQ *)
REPEAT ....                          REPEAT ....
     BEGIN                             BEGIN
        (* CS *)                               (* CS *)
        WREQ(ID);                              WACK(ID);
        COPY;                                  COPY;
     (RESA:=RESA1)                         (RESA2:=RESA)
        ACK(ID);                               REQ(ID);
        (* NCS *)                              (* NCS *)
        ----                                   ----
        RESA1:=                                  :=F(RESA2,..);
        ----                                   ----
     END;                              END;
```

figure 32: solution for example 1 using copies

Sometimes processes access identical variables but not alternately. PROC P needs the variable only after PROC Q has processed it a fixed number of times. A synchronization solution for this problem is:

```
PROC P;                              PROC Q;
----                                 ----
LOOPA                                LOOP
  BEGIN                                BEGIN
     ---                                       ----
     WACK(ID);                                 WREQ(ID);
     ---                                       -----
     LOOPB                                     WAITRES(RESA);
       BEGIN                                     :=F(RESA,..);
          WAITRES(RESA);                       RESA:=G(RESA,...);
          RESA:=F(RESA,..);                    RELRES(RESA);
          RELRES(RESA);                        -----
       END;                                    ACK(ID);
     REQ(ID);                              END;
   END;                              ----
----
```

The synchronization prevents that the resource is accessed by both partners at the same instant, so the mutual exclusion primitives can be dropped. But if another process reads RESA at unexpected times (e.g. to check the evaluating of variable RESA) the primitives are essential. Read and write mutual exclusion could be implemented.

4.5. communication

Processes communicate with each other. Communication means giving information to other parties or exchanging information between at least two partners. In a hardware design several types of communication occur:

- communication to the external world;
- communication between processes scheduled after each other in time. These processes are allocated in different modules or in one module;
- communication between parallel processes in different modules; otherwise the processes aren't concurrent.

In every situation the integrity of the data should be guaranteed.

The only difference between the first type and the others is that the expected event doesn't have to occur, since the external world doesn't need to react. For the second type the timing is no problem. The information should be ready before the receiving partner can accept it. The allocation must assure this, otherwise synchronization must be introduced. The sender produces the information and stores it somewhere, but will never know if the receiving party gets the information. When the sender is active again and has to know if the information is fetched, synchronization e.g. R/A synchronization must be used. The receiver should be able to access the store. Communication between processes in the several modules requires that the information is accessible indicating that the resource is part of several datapaths. For processes in the same module the store needn't be accessible for other modules.

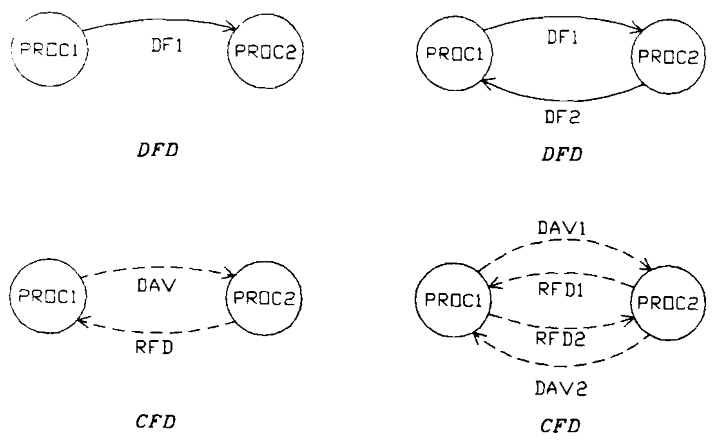


figure 33: requirements model for communication

The receiver knows the store's address (direct access) or knows the address of a store containing an address (indirect access). Several addressing methods can be used. Because the information exchange occurs in the time, it is possible that the information is changed by another process before the receiver fetches it. The allocation redefining PSPECs and CSPECs should avoid this.

For the third type the communication takes place while the processes are both running requiring synchronization. A handshake transports the data over a resource (databus). The receiver translates the protocol signals into control signals for its datapath, or another architecture module accesses directly the resource belonging to the datapath of the receiver, which requires that the control lines are multiplexed, but the exchange is still synchronized. An objective for this third type is to minimize the number of resources. Only if the required program flow/execution is slowed down, the information is exchanged using copies.

Exchanging information between two parallel processes requires that both partners are ready for the transaction. Otherwise the receiver can miss the information. Besides we should try to minimize busy-waiting of one partner in the exchange, happening when only one partner is ready. The partner should be signalled, that the correct data is presents. A handshake protocol is introduced to synchronize the communication (see fig 34).

PROC1;	**PROC 2;**
SETUP LINK	ACK SET UP LINK
SEND INFO	RECEIVE INFO
REMOVE LINK	ACK REMOVE LINK

figure 34: specification of communication

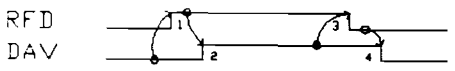


figure 35: data communication protocol using handshake and ready for data (RFD) and data available (DAV).

In the architecture model a data bus connects modules. Several modules can try to access that bus (shared resource) at the same instant. The allocation introduces a resource manager for the bus (**bus arbiter module** BAM). In the implementation a method is needed to select the correct partner. The sender can select the receiver by generating a synchronization signal (e.g. DAV for AM1). A DAV line is connected to every architecture module where to data will be sent. A more common method is the put an address on the bus, which selects the receiving module.

The data is put on the bus after the grant of the arbiter. The protocol can be found in figure 35. So we use the handshake of (see figure 36):

1. ready for data (**RFD**)
2. data available (**DAV**)
3. data copied
4. no data available anymore

This protocol is necessary because of the parallel operating processes.

If both modules use the same clock frequency, the protocol can be simplified. After making the data available, the data will be copied in a fixed period. So the signal **data copied** is redundant. Even so **no data available any more**. The implementation of the synchronization and communication depends on the process allocation.

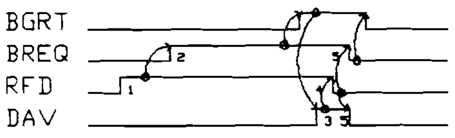


figure 36: data communication protocol using handshake and a resource manager.

The **PSPEC** *or* **CSPEC** *use program structures, synchronization and mutual exclusion primitives to guarantee the data integrity.*

Sometimes the **PSPEC** of the communicating processes don't require a protocol for the exchange. The necessary synchronization (e.g. critical sections) assures correct information exchange avoiding problems in the allocation as long as an information channel is allocated exclusively to the communicating modules. With an exclusive channel only some hardware register control signals are generated for the datapath belonging to the other module or communication synchronization signals are generated for the control unit of that module, where they are translated into control signal for the resources in the datapath. Otherwise the channel is shared requiring mutual exclusion.

Later we will discuss in detail the communication for shared processing, which has a parameter and result phase. Mutual exclusion and a correct synchronization is needed, so that the process receives the correct parameters/results at the correct moment.

THE DATA COMMUNICATION PRIMITIVES

We define primitives for the data transport on a private channel using a handshake protocol. When the channel is shared, we use a bus arbiter. The number refers to actions in figure 36.

```
PUTARB :=
    REPEAT SKIP UNTIL RDF; (*1*)
    WAITRES(BUS); (*2*)
    COBEGIN
        DAV:=TRUE; (*3*)
        ENBUS:=TRUE; (* Enable data on bus *)
    COEND;
    REPEAT SKIP UNTIL NOT(RDF); (*4*)
    COBEGIN
        DAV:=FALSE; (*5*)
        ENBUS:=FALSE;
        RELRES(BUS);
    COEND
```

```
PUTARB (SOURCE) :=
    REPEAT SKIP UNTIL RDF; (*1*)
    WAITRES(BUS); (*2*)
    COBEGIN
        DAV:=TRUE; (*3*)
        EN[SOURCE]:=TRUE; (* Enable data on bus *)
    COEND;
    REPEAT SKIP UNTIL NOT(RDF); (*4*)
    COBEGIN
        DAV:=FALSE; (*5*)
        EN[SOURCE]:=FALSE;
        RELRES(BUS);
    COEND
```

This last primitive will be used, when several registers can put data on the bus. Using a private channel they change into: (see figure 36 data communication protocol).

```
PUT :=
    REPEAT SKIP UNTIL RDF; (*1*)
    DAV:=TRUE; (*2*)
    REPEAT SKIP UNTIL NOT(RDF); (*3*)
    DAV:=FALSE; (*4*)

PUT (SOURCE) :=
    REPEAT SKIP UNTIL RDF; (*1*)
    COBEGIN
        DAV:=TRUE; (*2*)
        EN[SOURCE]:=TRUE; (* Enable data on bus *)
    COEND;
    REPEAT SKIP UNTIL NOT(RDF); (*3*)
    COBEGIN
        DAV:=FALSE; (*4*)
        EN[SOURCE]:=FALSE;
    COEND
```

The data owner puts the data on the bus, since the bus should be requested when the actual transport will take place to avoid locking the resource unnecessarily long. The receiver doesn't request the bus, so the primitives don't differ for a private or shared channel.

```
GET :=
    REPEAT SKIP UNTIL NOT(DAV);
    RFD:=TRUE;
    REPEAT SKIP UNTIL DAV;
    COBEGIN
        LDREG; (* load reg *)
        RFD:=FALSE;
    COEND

GET (DEST) :=
    REPEAT SKIP UNTIL NOT(DAV);
    RFD:=TRUE;
    REPEAT SKIP UNTIL DAV;
    COBEGIN
        LD[DEST]; (* load destination register *)
        RFD:=FALSE;
    COEND
```

5. SHARED PROCESSING

In the requirements model identical transformations can be found at various places operating on different data flows of the same composition (fig 37). This process can be allocated to one architecture module redefining the requirements model and introducing the shared process (fig 38). To simplify the discussion and the figures we allocate one process to one module, which is divided into several calling processes, or shared processes and control transformations at the next level.

A shared process is called by processes allocated to processing architecture modules (PAMs). Different shared processes are allocated to one architecture module. So the process SP in figure 38 can contain several shared processes at the next level. Shared processes are allocated in one module, since they require special hardware to communicate with the callers (requesters). *In some applications a transformation is encountered so often, that the identical shared process is implemented in more modules.* In both configurations it is not yet defined, how the architecture module selects a call.

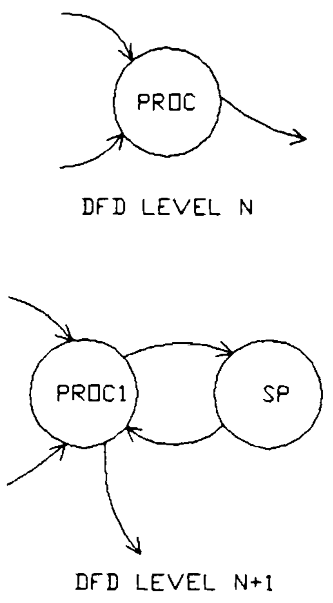


figure 37: identification of shared process

We distinguish three partners: the caller (calling process (CP)), the shared process (SP) and a serving process (server). A calling process is executing its program when it encounters a set of instruction to execute a transformation allocated in another architecture module, better fit to execute the called job (specialized hardware e.g. multiplier) or speeding up the program execution (parallelism, pipelining) or just saving resources in the implementation. The calling process gives its request to the server, which schedules the requested process on the **shared processing architecture modules (SPAM)**, after they request a partner.

During process execution the caller and shared process communicate, so synchronization signals are exchanged between the partners to assure correct communication requiring redefinition of the requirements model introducing new processes, since an input and output flow of the shared process must be directed to the caller.

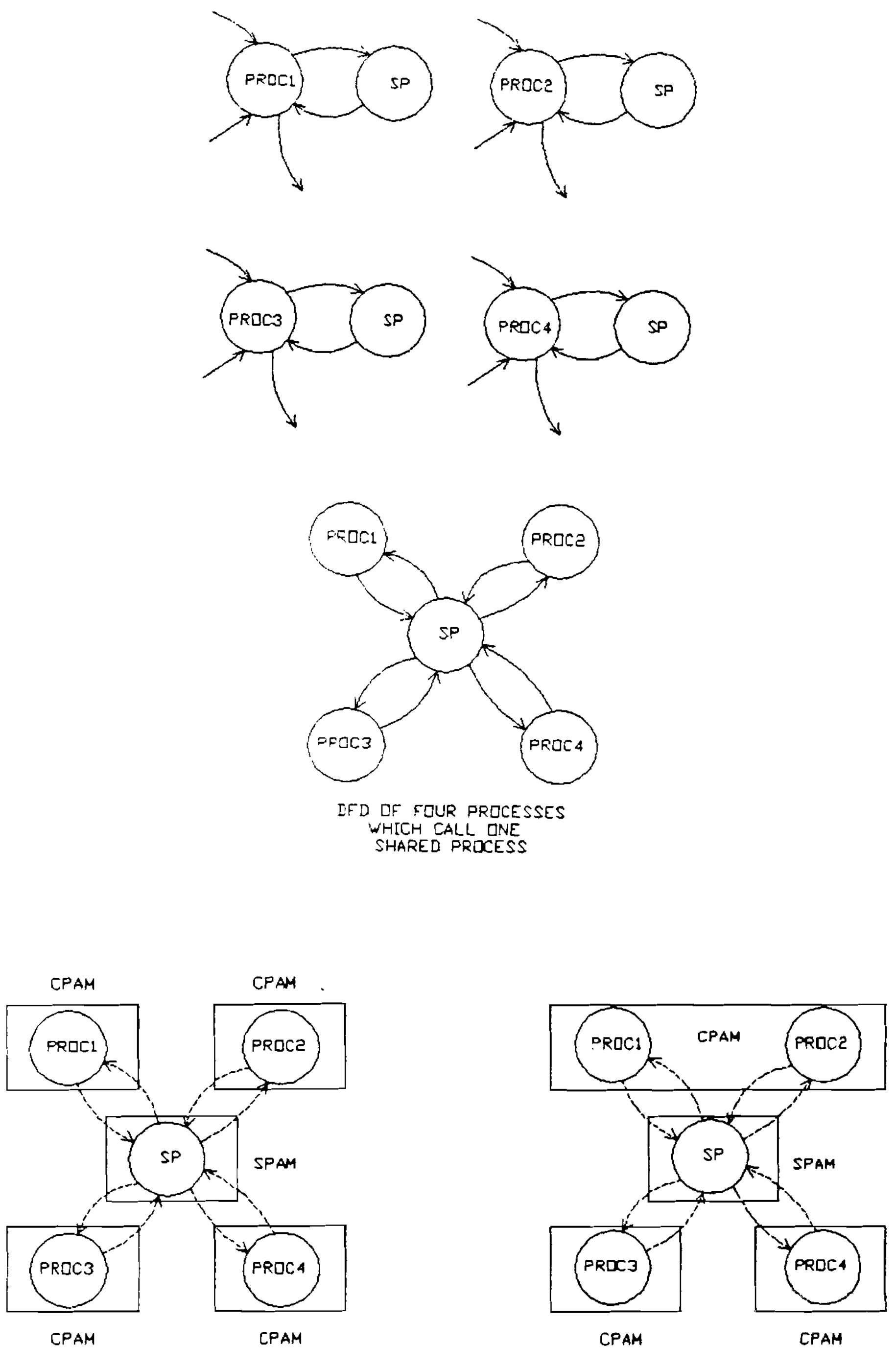


figure 38: shared process and its allocation

In a SPAM *containing several shared processes, a program (monitor) will start the correct process after requesting from the server the process identity.* The server knowing all the pending service requests and their priorities gives the identity of the requested process can be given to the SPAM by raising a signal identifying the process. A CPAM can have more request lines (one for every shared process). Another method is to generate a command word included in the parameter communication phase.

We introduce three kinds of servers: first, the hardwired server, which selects the process to start and sets up the synchronization link and gives the **SPAM** the identity of the requested process, second, the arbiter server, which is only an arbiter and where the caller gives the shared processing module a command indicating the requested process and third, the buffer server, where buffers for the input and output communication are used. The first two servers require a direct connection between the **CPAM** and the **SPAM** to synchronize actions.

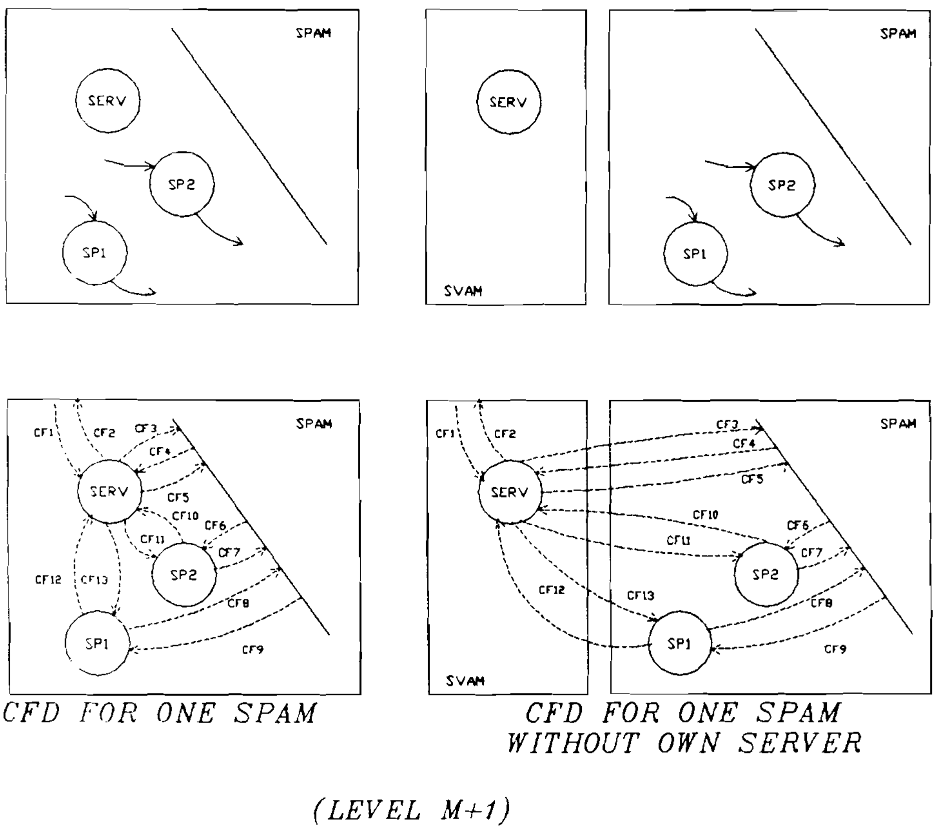
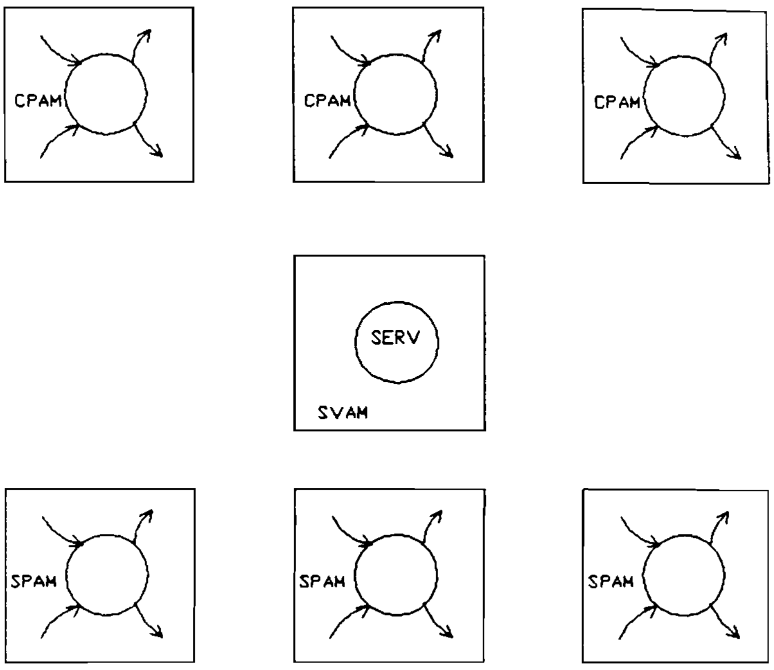


figure 39: allocation to one SPAM

The task of the server is to decide, which request will be granted, after the shared processing module requests a new process. If no call is pending, the server keeps testing all the inputs until a call is received. The server selects the call with the highest priority (no starvation is possible using a rotating priority scheme). Using a hardwired server the call will be decoded into the correct SPID (shared process identifier), for the monitor the identity of the process to execute. The calling process will receive a grant.

First we discuss the allocation of the shared processes into one **SPAM**. Several **calling process architecture modules** (**CPAM**) operate in parallel. Out of the generated requests one is chosen, then the called process must be determined and started. The **SPAM** has to synchronize and to communicate with the correct **CPAM** requiring a multiplexer. The server executes these functions. Figure 39 contains the **CFD** for one **SPAM**.



DFD

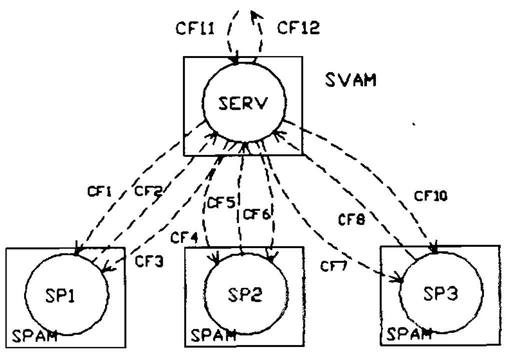


figure 40: allocation to several SPAMs

We won't define the control flows at this point, since they depend on the communication and synchronization protocols. **CF1** and 2 combine all control signals of all **CPAMs**. **CFs** 10,11,12,13 contain all control flows of the selected **CPAM**. **CF3** signals the selection of a caller and **CF5** gives the identity of the requested process. Using one **SPAM** the server can be allocated to that **SPAM** or to a separate module (SVAM server architecture module).

In an allocation with several **SPAMs** the server has to be allocated to an architecture module (fig 40). The second control flow from the server to the **SPAM** exists of the identity of the called process. Since a databus is used, all modules are connected with each other. The data communication synchronization is allocated to the control flow directed to the multiplexing server making the connection.

Every allocation depends on the method of generating the identity of the called process, the server type implemented, the number of **SPAMs** *and the data parameter/result communications method.* We will see several allocations in the next chapters. Because of the complexity we won't discuss the techniques separately, but they will be illustrated in implementations combining several methods.

6. DIRECT CONNECTION SHARED PROCESSING METHODS

6.1. the primitives for starting a shared process

6.1.1. SERVER AND CALLER SYNCHRONIZATION

A purpose of concurrency is to call a shared process and still execute a program. The caller doesn't know when its call will be served, so it is essential to execute a parallel program while generating a request to activate the shared process. The request will be active until it is served. After a while the server grants the request signalling it to the caller, which removes the request. The caller should start testing the grant immediately after the request generation, so that actions (e.g. parameter preparation) can be started after the grant receipt causing an efficient use of the shared processing module. A four phase signaling is used with two lines called service request (**SVREQ**) and service grant (**SVGRT**). Because of the parallel execution, the receipt of a signal should be acknowledged, otherwise the sender doesn't know, if its signal is received and if the correct action will be started.

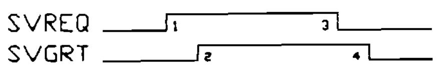


figure 41: svreq and svgrt

The synchronization between the server and caller:

CALLER	**SERVER**
1 SVREQ(IDCPAM)	-
2/3 WSVGRT(IDCPAM)	2 SVGRT(IDCPAM)
-	4 WREMSVREQ(IDCPAM)

IDCPAM	: IDENTIFIER OF THE CPAM
SVREQ(IDCPAM)	: SVREQ:=TRUE; (service request)
SVGRT(IDCPAM)	: SVGRT:=TRUE; (service grant)
WSVGRT(IDCPAM)	: REPEAT SKIP UNTIL SVGRT; (wait for service grant)
REMSVREQ(IDCPAM)	: SVREQ:=FALSE; (remove service request)
WREMSVREQ(IDCPAM)	: REPEAT SKIP UNTIL NOT(SVREQ); SVGRT:=FALSE; (wait remove service request)

The caller has the responsibility to remove the request indicating that the **SVGRT** has been received and the next phase can be started. After the request removal, the grant will be removed.
The requirements model of the calling process:

CALLER PSPECs:

```
....
svreq(idcpam);
cobegin
    begin
      prog; (* parallel program *)
    end
    begin
      wsvgrt(idcpam)
      prog__com; (* communication necessary for spam *)
    end
coend;
```

The part PROG_COM contains the REMSVREQ(IDCPAM). This statement can be executed with WSVGRT(IDCPAM).

6.1.2. SPAM MONITOR AND SERVER SYNCHRONIZATION

A monitor and several processes are allocated to the SPAM. After process execution the SPAM monitor requests a new partner and starts the new program after receiving a server acknowledge, generating the correct handshake signals and getting the identity of the requested process. We introduce again a four phase signaling with one monitor request line (MNREQ) and one monitor acknowledge line (MNACK). A handshake is used to signal the receipt.

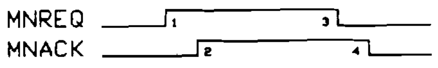


figure 42: synchronization monitor and server

The communication between both parties:

MONITOR	SERVER
1 MNREQ(IDSPAM)	WMNREQ(IDSPAM)
2 WMNACK(IDSPAM)	MNACK(IDSPAM)
3 REMMNREQ(IDSPAM)	WREMMNREQ(IDSPAM)
4	REMMNACK(IDSPAM)

IDSPAM	: IDENTIFIER FOR MONITOR AND SERVER SYNC
MNREQ(IDSPAM)	: MNREQ:=TRUE; (monitor request)
MNACK(IDSPAM)	: MNACK:=TRUE; (monitor acknowledge)
REMMNREQ(IDSPAM)	: MNREQ:=FALSE; (remove monitor request)
WMNREQ(IDSPAM)	: REPEAT SKIP UNTIL MNREQ; (wait for monitor request)
WMNACK(IDSPAM)	: REPEAT SKIP UNTIL MNACK; (wait for acknowledge)
WREMMNREQ(IDSPAM)	: REPEAT SKIP UNTIL NOT(MNREQ); (wait for removal mnreq)

To illustrate the synchronization, we use a server generating the SPID (shared process identity) out of the service request and giving it to the monitor. The SPID decoding starts the correct process. The monitor acknowledge (MNACK) signals the partner selection and the SPID generation. How this SPID is fetched, depends on the implementation.

The requirements model of the monitor (example):

SPAM MONITOR PSPECs:

```
while true do
    begin
        mnreq(idspam); (* request new process to execute *)
        wmnack(idspam); (* wait for spid to execute *)
        fetch__spid;
        remmnreq(idspam); (* remove request for new process*)
        wrmnack(idspam); (* wait  for end of protocol *)
        case spid do
                id1    : proc id1;
                id2    : proc id2;
                ....  .....
                end;
    end;
od;
```

The server acknowledge will be held active, until the monitor has fetched the SPID. Otherwise it could be removed too early causing the generation of a new SPID by the server. Therefor the server will test the SVREQ line, before the grant will be removed.

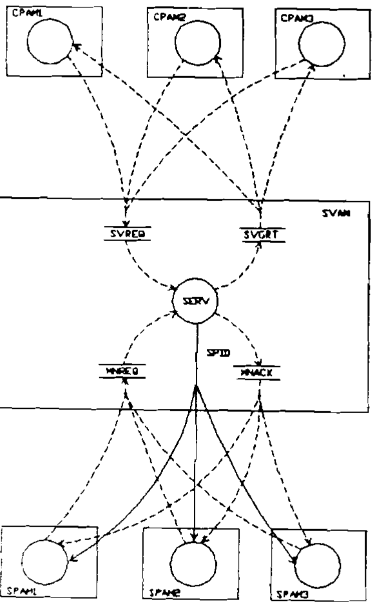


figure 43: the requirements model for shared process execution using a SPID server.

If only one **SPAM** is used, the allocation can change redefining the handshaking, since no parallel operation is possible allocating the server to the SPAM.

6.2. parameter and result communication protocol

A shared process needs data to operate on (parameters). The output flow (the result) is directed to the caller. To exchange this information synchronization is required, because both modules operate in parallel. We will discuss two synchronization schemes. The first one uses multiplexed control lines for the datapaths. Another module can control resources in the datapath. The shared process doesn't know the caller's identity, so it can't access the caller, which has the responsibility for the data transport (putting the parameters in the registers and getting the result out of the registers of the shared processing module). The caller will request the bus for the transactions and generate the control signals for the registers. The data transport doesn't need any handshaking accessing the resources correctly, since the **SPAM** is waiting for parameters and afterwards will be waiting until the result is fetched. In the other scheme we will introduce a handshake protocol for data transport using the process knowledge how much data will be transferred.

First we discuss the result and parameter communication, which multiplexes control lines. We introduce the **PTR** (parameter result) line and the **RDY** (ready) line.

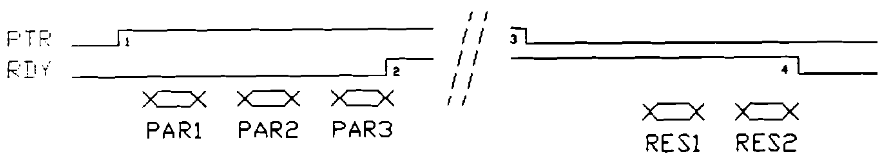


figure 44: synchronization between called and shared process

The communication between the caller and the shared process.

CALLER	SHARED PROCESS
1. WREQPA;	REQPA; (* ready to receive parameters *)
2. RDYPA;	WRDYPA; (* wait until all parameters are received *)
3. WSIGRSU;	SIGRSU; (* signal result is ready *)
4. RDYRSU;	WRDYRSU; (* wait until result fetched *)

Implementation:

REQPA	: PTR:=TRUE; (request parameters)
WREQPA	: REPEAT SKIP UNTIL PTR; (wait for request parameters)
RDYPA	: RDY:=TRUE; (ready parameters)
WRDYPA	: REPEAT SKIP UNTIL RDY; (wait for ready parameters)
SIGRSU	: PTR:=FALSE;(signal result ready)
WSIGRSU	: REPEAT SKIP UNTIL NOT(PTR); (wait for signal result)
RDYRSU	: RDY:=FALSE; (result fetch ready)
WRDYRSU	: REPEAT SKIP UNTIL NOT(RDY); (wait for ready result fetch)

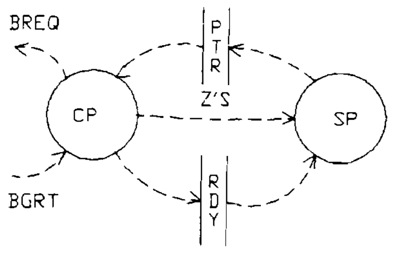


figure 45: data protocol using multiplexed control lines to resources.

In the other scheme the data communication protocol is implemented. This protocol requiring **DAV** and **RFD** signals indicating that parameters or results will be transmitted, which is allowed since the result and parameter phase have a known sequencing. The **DAV** and **RFD** stores are exclusive for the specified communication between the selected **CPAM** and **SPAM** and are only used for result and parameter communication.

CALLER:	SHARED:
PUT(PAR2);	GET(PAR2);
PUT(PAR3);	GET(PAR3);

figure 46: example of parameter communication

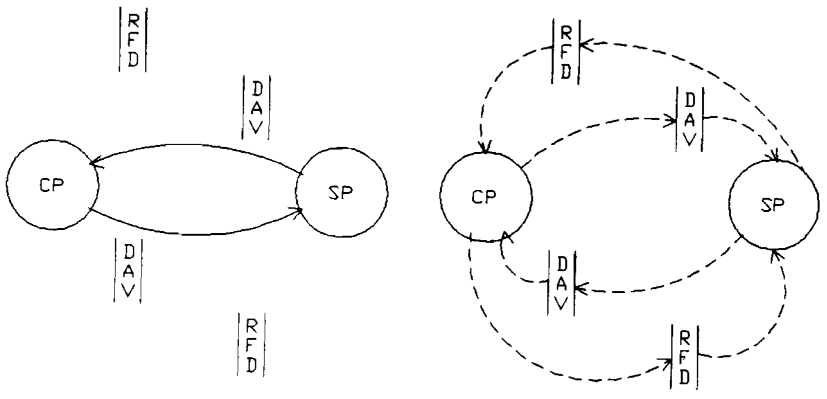


figure 47: requirements model for parameter and result communication using DAV and RFD.

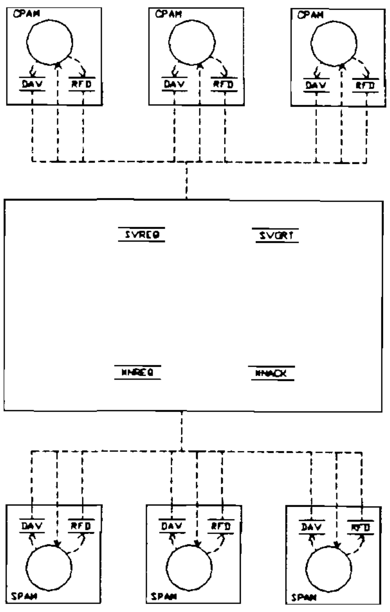


figure 48: allocation of control stores for ptr communication

Both schemes will be illustrated, when the servers are discusses.

6.3. a hardwired process call identity allocation model and pspecs

6.3.1. ONE SPAM

The hardwired process identity server avoids starvation of the request calls and builds up the link between the **CPAM** and the **SPAM**, which doesn't know the caller's identity. The partner of the caller is the server, which has more partners (the shared processing module and other callers). A request for a shared process is generated by raising a signal. The caller can select the shared processing module registers using address without needing the protection of a mutual exclusion primitive.

For the moment, we will use just one shared processing module. The server determines the caller and the identity of the requested process. The service request indicates the callers identity and the requested **SPAM** process, so one **CPAM** can generate different service requests.

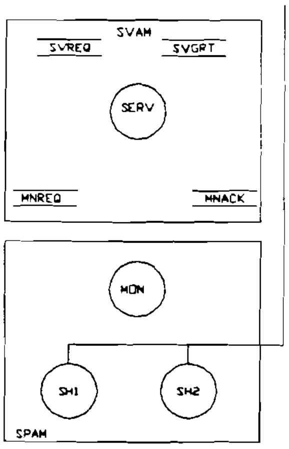
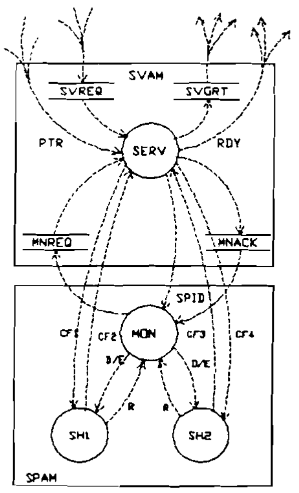


figure 49: allocation model



HARDWIRED CALL AFD SHOWING CF
R= READY
D/E= DISABLE ENABLE

```
CALLER PSPEC:
svreq(idcpam);(* request shared process *)
cobegin
   begin
      wsvgrt(idcpam); (* wait until process assigned and grant this receipt *)
      wreqpa;(* wait until parameters are requested*)
      (* transmit parameters (including bus request) *)
      waitres(bus);
      reg1:=par1; (*load parameter 1 in register 1 of shared processing module *)
      reg2:=par2;
      reg3:=par3;
      .....
      cobegin
         reqn:=parn;
         relres(bus);
         rdypa; (* signal parameters transmitted *);
      coend;
      prog; (* program execution *)
      wsigrsu; (* wait until result is ready *)
      (* copy result; (includes bus requests)*)
      waitres(bus);
      rrsu1:=rsu1; (* load result 1 contained in register rsu1 of the shared
                         processing module in the register  rrsu1 of the caller *)
      ......
      cobegin
         rrsum:=rsum;
         relres(bus);
         rdyrsu; (* signal result copied *)
      coend;
   end
   begin
      prog (parallel program)
   end
coend;
```

The PSPEC of the called shared process:

```
PROCESS SPID PSPEC:
begin
   cobegin
      init; (* starts some initialization independent of the parameters *);
      begin
         reqpa;
         wrdypa;
         init2; (*initialization depending on the parameters *)
      end
   coend;
   processing;
   cobegin
      exit1; (* statements independent of the result communication to leave the process*)
      begin
         sigrsu;
         wrdyrsu;
         exit2; (* exit statements, which can only be executed after the result is fetched*)
      end;
   coend;
end;
```

```
SPAM MONITOR PSPEC:
while true do
   begin
      mnreq(idspam); (* request new process to execute *)
      wmnack(idspam); (* low active *)
      load__spid:
      remnnreq(idspam);
      case spid do
            id1    : proc id1;
            id2    : proc id2;
            ..     .....
            end;
   end;
```

Combining the introduced **PSPECs** we produce an **AFD** containing the primitive signals (figure 50). The calling process module controls some resources in the **SPAM** (z-bus). The **SVREQ** is decoded by the server into a SPID. When the server is active, the shared processing module is idle. Hence the server could be implemented in the **SPAM**.

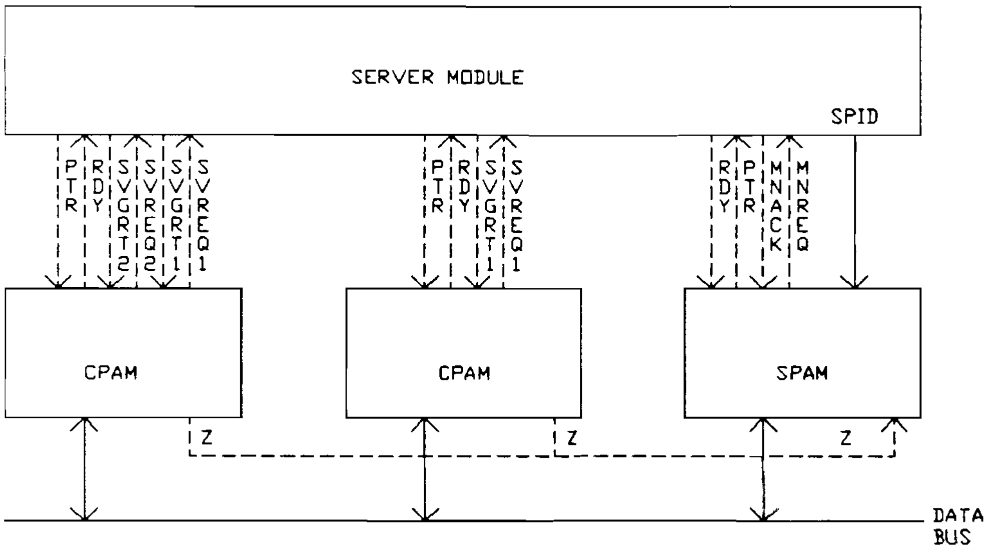


figure 50: AFD of the hardwired calls shared processing model

```
SERVER PSPEC:
begin
  while true do
    begin
      wmnreq(idspam);
      repeat skip until request;
      select__request__idcpam;
      svgrant(idcpam);
      mnack(idspam);
      cobegin
        wremmnreq(idspam);
        wremsvreq(idcpam);
      coend;
      remmnack(idspam);
      connect; (* select the primitive signals of the cpam *)
      wrdy; (* the controller monitors the changes of mnack/rdy line *)
      wremrdy; (* necessary to keep track of the state *)
      disconnect; (* disconnect the primitive signals of the cpam for the spam *)
    end;
  od;
end;
```

The server monitors event after each other SVREQ, removal SVREQ, MNREQ, removal MNREQ, WRDY, WREMRDY. In appendix C the protocol execution is discussed.

6.3.2. PARALLEL OPERATING SPAMS

When several shared processing modules operate in parallel, they can generate a MNREQ in the same moment requiring an arbiter to select the MNREQ to serve. After the service grant, the SPID and the CONID (the id of the CPAM to serve and to build up the connection with) are ready.

The server module should monitor several events at the same instant (SVREQ, removal SVREQ, MNREQ, removal MNREQ, WRDY, WREMRDY), now SPAMs operate in parallel. The monitoring of several WREMRDY and WRDY signals means that parallel processes are running in the server; a server process is active for every SPAM . Using one sequential module these processes have to be scheduled making them semi parallel introducing waiting time. One server module for every SPAM doesn't work, because a request can be granted by more modules at the same moment. The solution is to introduce logic to generate the control signals after the event occurrence. Only separate logic can guarantee parallel operation. The event (e.g. MNACK) is translated by the logic into a control signal (e.g. load SPID and CONID). This introduces a large amount of logic which is difficult to design and can give timing and hazards problems. We conclude that the technique used for one SPAM isn't useful here.

If the connect module is located in the server module, the server has to know the identity of the SPAM to build up the correct link between the SPAM and the CPAM. The SPAM has the knowledge to break down the link, hence it will control the connect module. No problems arise anymore about what multiplexer to select to serve the correct SPAM. After selecting a CPAM for granting a SPAM, the SPAM consisting of a connect and process module, getting the SPID and the CONID sets up the link. The server had already to send data to the SPAMs (SPID) and now it sends more information (CONID).

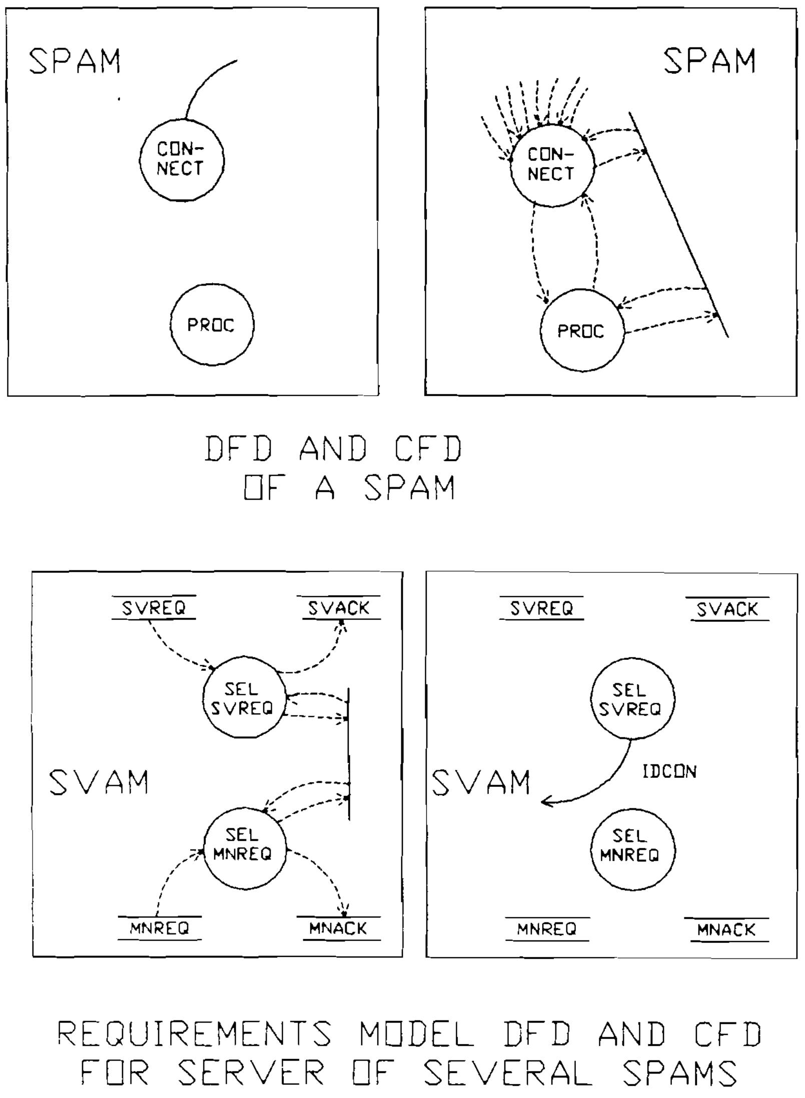


figure 51: requirements model SVAM and SPAM

A resource manager gives access to the server module. After being granted the server, the SPAM generates a MNREQ resulting in the server generating the SPID and the CONID. The SPAM clocks the data. The resource manager is the arbiter for the MNREQs, which are changed into a start signal for the server arbiter. Only one PAM picking up the signals, has access to the server module. Because the other SPAMs don't have a grant for the server module, they ignore the generated signals.

For the specification we define the following identifiers:

SA	: resource management and SPAM
IDSPAM	: id of shared processing module
IDCPAM	: calling cp

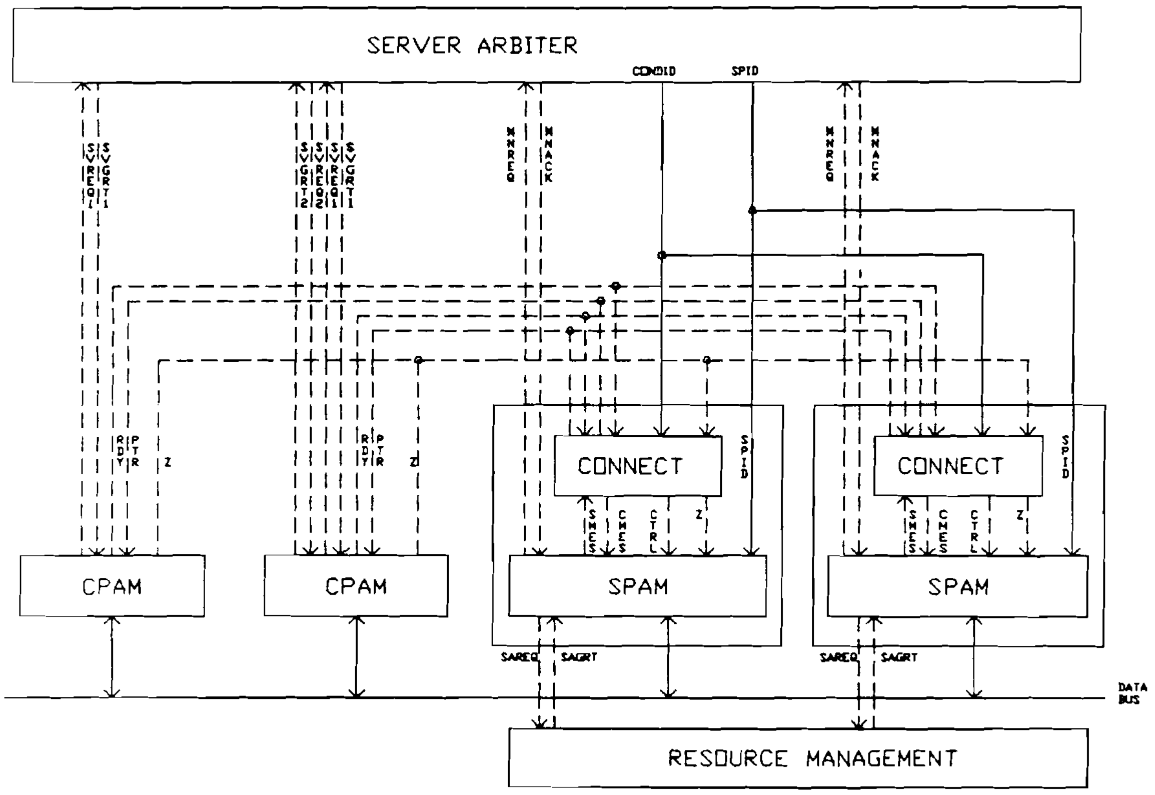


figure 52: hardwired calls AFD/AID using resource management

SPAM MONITOR PSPEC:

```
while true do
    begin
      cobegin
            res_conid; (* problems can arise, if this connection isn't reset, when another
                  spam serves this conid, and no new svreq can be assigned to this spam.*)
            waitres(sa);
      coend
      mnreq(idspam); (* request new process to execute *)
      wmnack(idspam); (* spid and conid are ready *)
      cobegin
            remnnreq(idspam);
            load_conid;
            load_spid:
            relres(sa);
      coend
      case spid do
            id1   : proc id1;
            id2   : proc id2;
            ..    .....
            end;
    end;
od;
```

```
PROCESS ID1 PSPEC:
begin
  cobegin
     init; (* starts some initialization independent of the parameters *);
     begin
        reqpa;
        wrdypa;
        init2; (*initialization depending on the parameters *)
     end
  coend;
  processing;
  cobegin
     exit1; (* start statements to leave the process independent of the result communication*)
     begin
        sigrsu;
        wrdyrsu;
        exit2
     end;
  coend;
end;
```

The caller specifications don't change. The CONID and SPID can be transported on the data bus, if the server requests the bus and the MNACK is used as **DAV** lines (twice). In appendix C the protocol execution, server implementation and optimization for the AID/AIF are discussed.

6.4. command allocation model and pspecs

6.4.1. INTRODUCTION

The shared processing module was only able to execute a few processes. Now this module gets a command. The module wants a number of parameters depending on the received command. The calling process knows the required number of parameters. The shared processing module will signal the caller, when it is ready with the execution. First it will send some status information telling the caller how many results bytes will be transmitted. This byte transported using the databus and **DAV** and **RFD** primitives, is decoded by the controller of the calling processing module.

A direct link exchanging synchronization signals exists between the caller and processing module. The server selects a calling processing module to serve and initiates the connection set-up. We discuss several situations (one SPAM and several SPAMs). We give first a simple solution, that is optimized (speed and silicon) in appendix C.

6.4.2. ONE SPAM

Several calling processing modules are connected to a server module serving as a buffer between the caller and the server processor (figure 53).

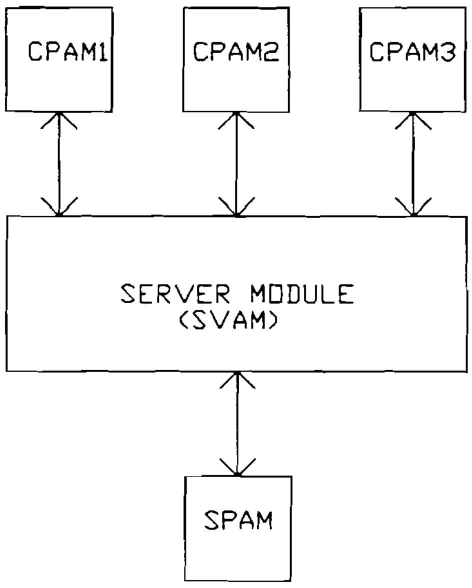


figure 53: general model

We describe the calling and the shared process actions using primitives sending the generated signals over dedicated lines. Again we introduce the identifiers:

IDSPAM	: identifier of shared process
IDCPAM	: identifier of calling process

THE CALLING PROCESS PSPEC:

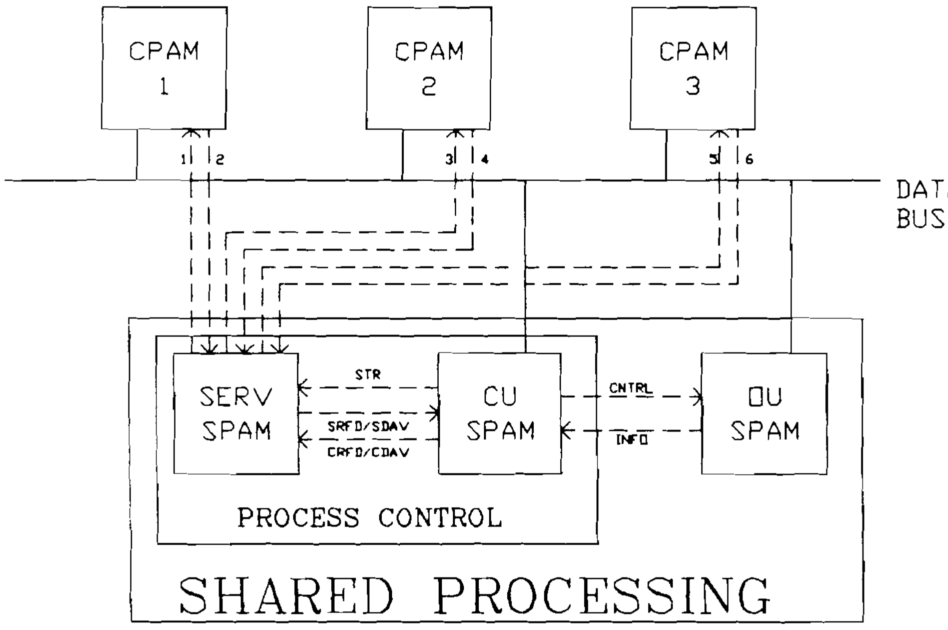
```
prog;
svreq(idcpam);
cobegin
   begin
      wsvgrt(idcpam);
      remsvreq(idcpam);
      put(par1); (* put command on bus *)
      put(par2); (* put parameter on bus *)
      ....
      prog
      get(res1); (* get status of bus *)
      decode(res1,ok); (*decode the status info *)
      if ok
           then
              begin
                 get(res2); (* get next result byte *)
                 get(res3);
                 ....
              end
           else proc errorhandling
   end
   prog; (* parallel program *)
coend;
```

In this description a parallel program is executed during the communication with the SPAM. The implementation should take care of the correct sequencing. The result can be ready before the CPAM can accept it, because of the parallel process execution. So the get primitive can only be executed by the CPAM after checking its own state looking for allowed interruption of its own program.

Using only one register for all parameters (put), after each put the register has to be loaded with a new parameter before the next put can be executed. This is also true for the get primitive. The SPAM wait until the CPAM is ready to receive the result.

```
THE SHARED PROCESSING PSPEC:
begin
    init
    get(par2); (* fetch parameter *)
    get(par3);
    execute
    status(res1,ok);(* generate status info; ok is false indicates that no result is produced.*)
    put(res1); (* put status on the bus *)
    if ok then
            begin
                put(res2)
                .....
            end;
end;

THE SPAM MONITOR PSPEC:
while true do
  begin
    mnreq(idspam);(* request new caller *)
    init; (* some initialization *)
    wmnack(idspam);
    cobegin
        remmnreq(idspam);
        get(parl);
    coend
    deccmd(procnumb,parl); (* decode cmd *)
    case procnumb do
        1     : proc 1;
        ..      ...
        ..      ...
        end;
  end;

SERVER PSPEC:
while true do
    begin
        wmnreq(idspam);
        disconnect; (* break down the connection*)
        repeat servarb(svreq[1..M],true,rdy,idcpam) until rdy;
        cobegin
            begin
                svgrt(idcpam);
                wremsvreq(idcpam);
                remsvgrt(idcpam);
            end;
            begin
                mnack(idspam);
                wremmnreq(idspam);
                remmnack(idspam);
            end;
        coend;
        connect; (* set the connection between the partners*)
    end;
od;
```

The program SERVARB selects out of the SVREQ[1..M] one request and grants it while generating the identity of the calling processing module (IDCPAM). The variable **RDY** indicates, that a SVREQ is found. If no request is pending, the program SERVARB keeps executing until a request is received. The value true is given to the arbiter's STR signal (appendix B). The DISCONNECT should be executed as early as possible to avoid that the SPAM receives signals from the wrong partner (see appendix C). In appendix C (implementation and allocation) separate signals for the data transport by the caller (CDAV, CRFD) and the SPAM (SDAV, SRFD) are introduced (C=CALLER, S=SHARED). SRFD requests a parameter and CRFD requests a result. Knowing the state a signal is interpreted correctly. So one line is used for SVREQ, CDAV, CRFD (CMES: caller message) and one line for SVGRT, SDAV, SRFD (SMES: shared processing message); as long as the implementation has only one SPAM. The allocation gets a separate server started by a signal STR. Figure 54 gives an allocations for direct connection command shared processing using one SPAM.



1. GRT1/SRFD1/SDAV1
2. REQ1/CRFD1/CDAV1
3. GRT2/SRFD2/SDAV2
4. REQ2/CRFD2/CDAV2
5. GRT3/SRFD3/SDAV3
6. REQ3/CRFD3/CDAV3

figure 54: AFD/AID CMD direct connection shared process using one SPAM

6.4.3. SEVERAL SHARED PROCESSING MODULES

The server in figure 55 containing several SPAMs with their own connection module, is a shared resource managed by the resource manager (RMAM). The caller and shared process description don't change.

Using several SPAMs we should make the correct connections. The server is no part of the shared processing module, since giving each module its own server requires them to ensure that no requests are granted twice by accounting which requests are served by who. Because allocating the connect module to the server requires bookkeeping of the mapping between the CPAM and the SPAM, the connect module is moved to the SPAM, which has only to pick its IDCPAM to set-up the connection.

The server is a shared resource given to a SPAM until the SVREQ is removed of the selected CPAM. After protocol execution with the caller, the server gives the IDCPAM to the SPAM for setting up the connection. A disadvantage of this solution is that the server can't be used again before the SVREQ removal (see appendix C).

The new process descriptions for the monitor and the server are (using SA as identifier for the server arbiter):

PSPEC OF THE SPAM MONITOR:

```
while true do
    begin
        disconnect; (* reset *)
        waitres(sa);
        mnreq(idspam); (* request idcpam *)
        wmnack(idspam); (* idcpam ready *)
        cobegin
            remmnreq(idspam);
            relres(sa);
            connect; (* load idcpam*)
        coend;
        init;
        get(parl);
        deccmd(procnumb,parl); (* decode cmd *)
        case procnumb do
            1      : proc 1;
            ..       ...
            ..       ...
            end;
    end;
```

The SPAM knows that the IDCPAM is the correct one, because no other SPAM can access the server (resource management). So no confusion can occur about the MNREQ and the MNACK.

SERVER PSPEC:

```
while true do
    begin
        wmnreq(idspam);
        repeat servarb(svreq[1..M],true,rdy,idcpam) until rdy;
        cobegin
            begin
                svgrt(idcpam);
                wremsvreq(idcpam);
                remsvgrt(idcpam);
            end;
            begin
                mnack(idspam);
                wremmnreq(idspam);
                remmnack(idspam);
            end;
        coend;
    end;
  od;
```

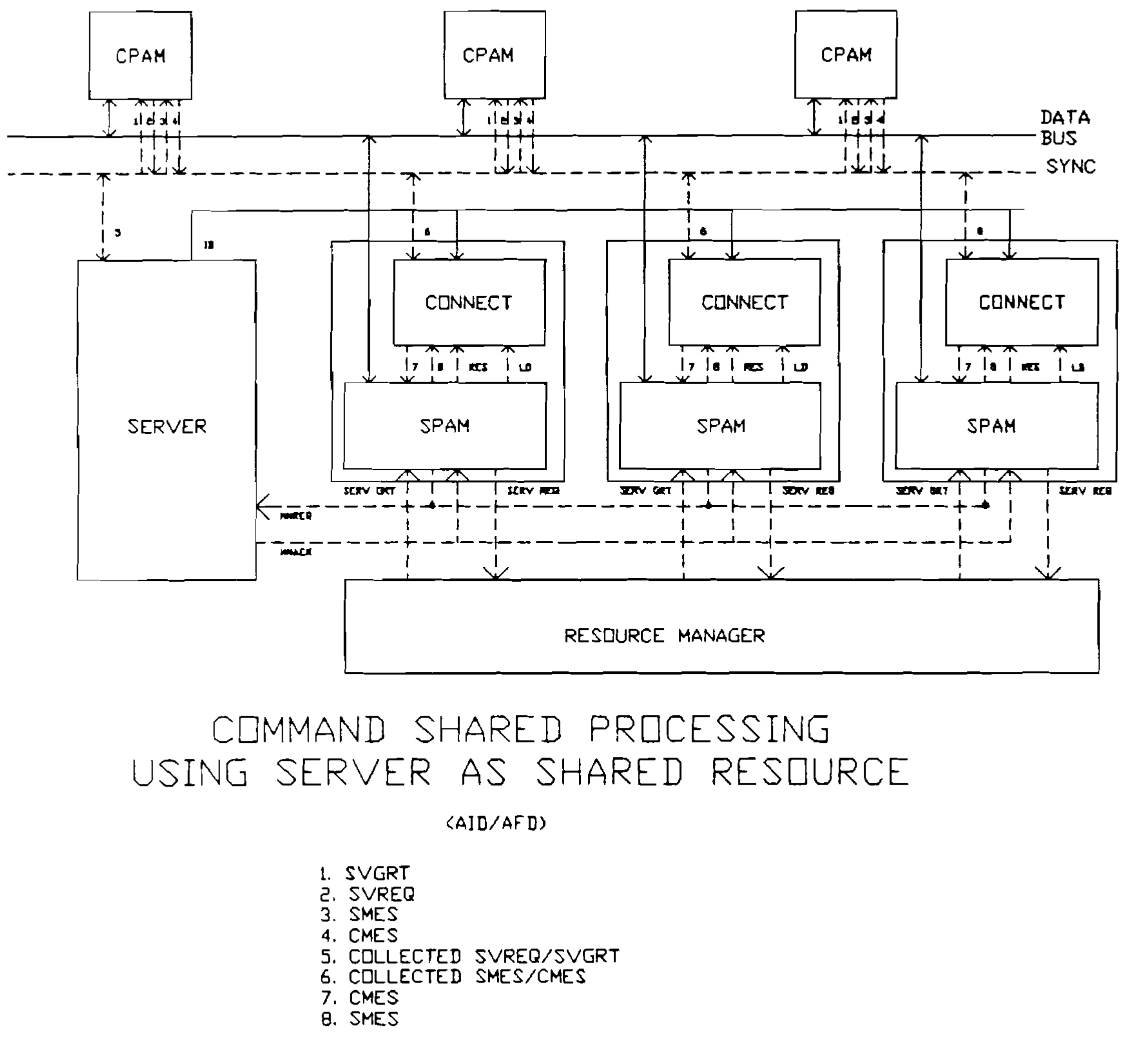


figure 55: AFD/AID command shared processing using resource management

In the appendix C we arrive at the following conclusions. The **MNACK** can be a pulse and the handshake can be removed, since the **SPAM** monitor is waiting for the event. The IDCPAM can be transported via the data bus. The server will request the bus. The **MNREQ** signal is then a **RFD** signal and the **MNACK** is a **DAV**. No separate lines are necessary, because only one **SPAM** can be in that state (resource management). The CDAV and CRFD would be seen by the server arbiter as a **SVREQ**, so the **SVREQ** and the CDAV/CRFD aren't multiplexed. Using the data bus for the IDCPAM communication the description changes into (**MNREQ** line=RFD, **MNACK** line=DAV):

MONITOR PSPEC:

```
while true do
      begin
            disconnect; (* reset *)
            waitres(sa);
            rfd:=true;
            repeat skip until dav;
            cobegin
                  rfd:=false;
                  relres(sa);
                  connect; (* load idcpam*)
            coend;
            init;
            get(parl);
            deccmd(procnumb,parl); (* decode cmd *)
            case procnumb do
                  1      : proc 1;
                  ..        ...
                  ..        ...
            end;
      end;
end;
```

SERVER PSPEC:

```
      while true do
         begin
            repeat skip until rfd;
            repeat servarb(svreq[1..M],true,rdy,idcpam) until rdy;
            cobegin
                  begin
                        svgrt(idcpam);
                        wremsvreq(idcpam);
                        remsvgrt(idcpam);
                  end;
                  begin
                        waitres(bus);
                        dav:=true;
                        en__idcpam__on__bus;
                        repeat skip until not(rfd);
                        cobegin
                              dav:=false;
                              relres(bus);
                        coend;
                  end;
            coend;
         end;
         od;
```

In the same appendix C the implementations of the server are discussed introducing architecture models for the CMD direct shared processing and trying to optimize the models. The model bottlenecks are the waiting time for the **SVREQ** and for data communication depending on the programs in the **CPAM** (critical section, shared resources) and on bus traffic, which of course depends on the same programs. Only a simulation can tell, if an allocation is the best for that problem. A real parallel server reducing the waiting time, has a very expensive implementation.

7. SHARED PROCESSING USING BUFFERING

7.1. the model

The shared processing module and the calling processing module have to wait, when a partner isn't ready to receive data. *To increase the independence of the processes execution we introduce buffers having an input and output process operating concurrent.* The calling process knowing the number of required parameters and the address of the buffer **BUFCMD**, sends a command, the parameters and an identification (IDCPAM) to that buffer. *A buffer allows a* **CPAM** *to generate more service requests, while an old request isn't processed yet.* The **SPAM** won't know the results destination, because no direct connection exists between the **SPAM** and **CPAM**. Therefore the calling process sends an identification of the requesting processing module to the **BUFCMD**.

The processing result can be ready before the **CPAM** can receive it. So the **SPAM** sends the results, the corresponding IDCPAM of the caller and the number of result bytes to the buffer **BUFRSU**.

Since the data is transmitted by the owner of the data, the **CPAM** puts the commands in the **BUFCMD** and the **BUFRSU** puts the result on the data bus using the data communication handshake protocol (**DAV**, **RFD**). Only the data owner can generate the bus request efficiently requesting the bus, when the receiver is ready to accept the data.

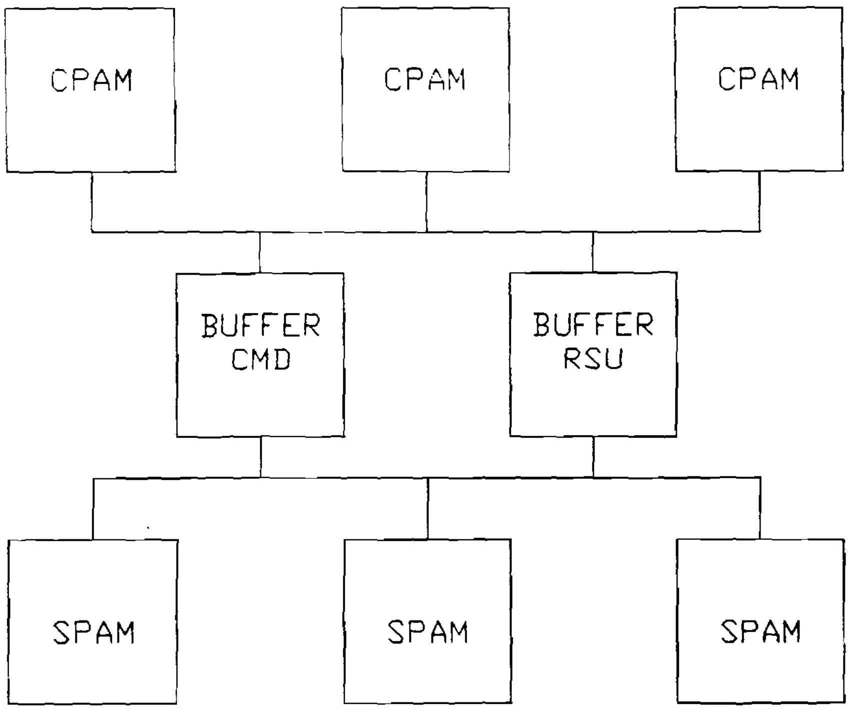


figure 56: AID for buffer communication

We define a **shared processing block (SPB)** containing two buffers (the **BUFCMD** and the **BUFRSU**) and one or several shared processing modules.

The **SPB** generates a signal, after the result is ready for the **CPAM**, requests the bus for the result communication and puts the result bytes on the bus after an acknowledge.

Starting a new process, a **SPAM** in the **SPB** requests access to the **BUFCMD**, waits when no data is available in the buffer. It knows after decoding the first command how many parameters to fetch in the **BUFCMD**. When a process is executed, the **SPAM** puts the result bytes in the **BUFRSU**.

7.2. resource management

When a **CPAM** starts sending its parameters to the **BUFCMD**, it should be the only module to have access. Avoiding mixing of commands and parameters we introduce a resource manager for the **BUFCMD**. The results are stored in the buffer **BUFRSU**, whose controller translates the calling process IDCPAM into a **SIGRSU** signal for the corresponding **CPAM**. Resource management isn't needed for the result communication, because only one source puts the result bytes on the data bus until all bytes for that **CPAM** are transmitted. After an acknowledge using **RFDCP** the result bytes are put on the bus (figure 57).

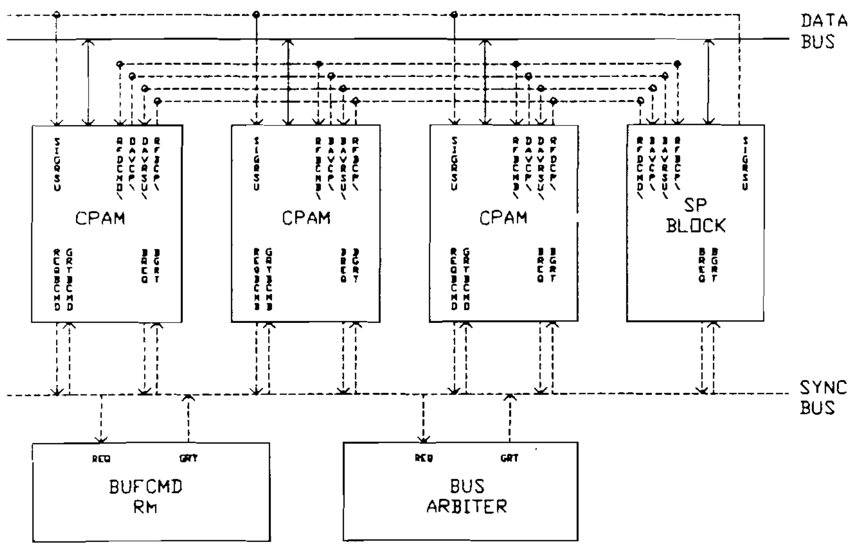


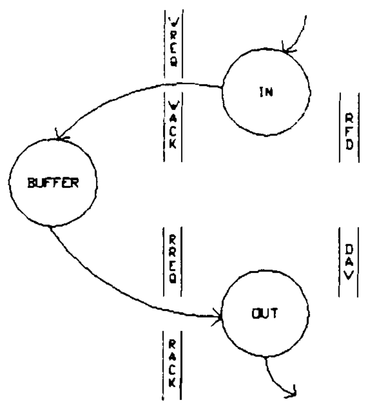
figure 57: AFD using resource management (CP level)

We define the channels ("\" indicates low active signals for the AIS):

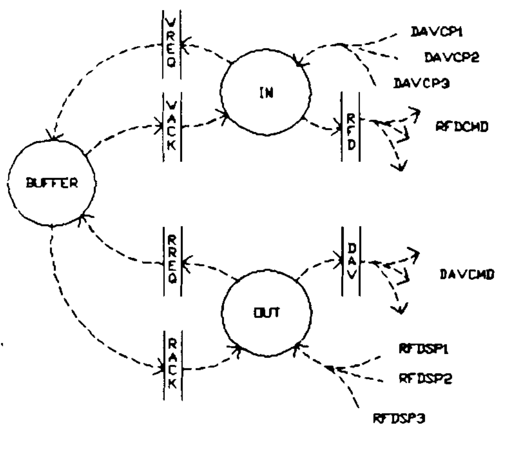
REQBCMD	: request for buffer
GRTBCMD	: grant for buffer
BREQCP	: bus request of cp
BGRTCP	: bus grant for CPAM
DAVCP\	: low active; CPAM has data available (shared line)
RFDCMD\	: low active; command buffer can receive data
SIGRSU	: signal result ready for this CPAM
BREQRSU	: bus request of BUFRSU
BGRTRSU	: bus grant for BUFRSU
DAVRSU\	; low active; BUFRSU has put data on bus
RFDCP\	: low active; shared line; CPAM ready for data

Several shared processing modules are allocated in the **SPB**. After fetching the parameters using resource management a **SPAM** releases the **BUFCMD**. After process execution, the **SPAM** requests access to the **BUFRSU** to store the result bytes releasing it after putting the last result in the buffer (figure 59).

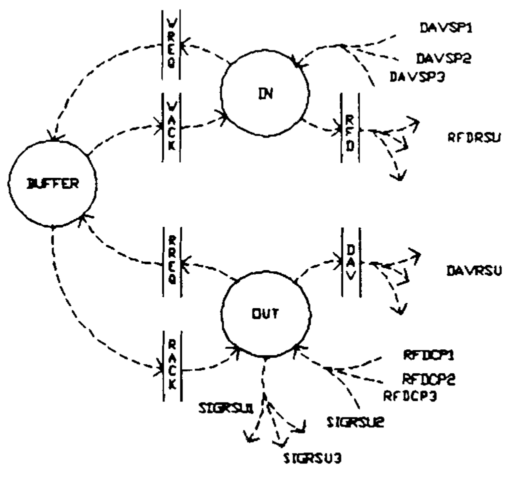
A requirements model of both buffers using the buffer introduced in appendix A can be found in figure 58. The BUFCMD_IN process generates a **RFDCMD**, when it can accept a new data byte (the buffer is not full). The **CPAM** puts the data on the bus using **DAVCP** (shared line) after buffer access is granted to the processing module. The **SPAM** will signal with the **RFDSP** line, that data can be received (after requesting the buffer from the resource manager). The BUFCMD_IN generates a bus request, when data is available. After being granted the bus, **DAVCMD** signals to the **SPAM** that the data is on the bus.



DFD GENERAL BUFFER MODEL



CFD FOR BUFCMD



CFD FOR BUFRSU

figure 58: requirements model buffer

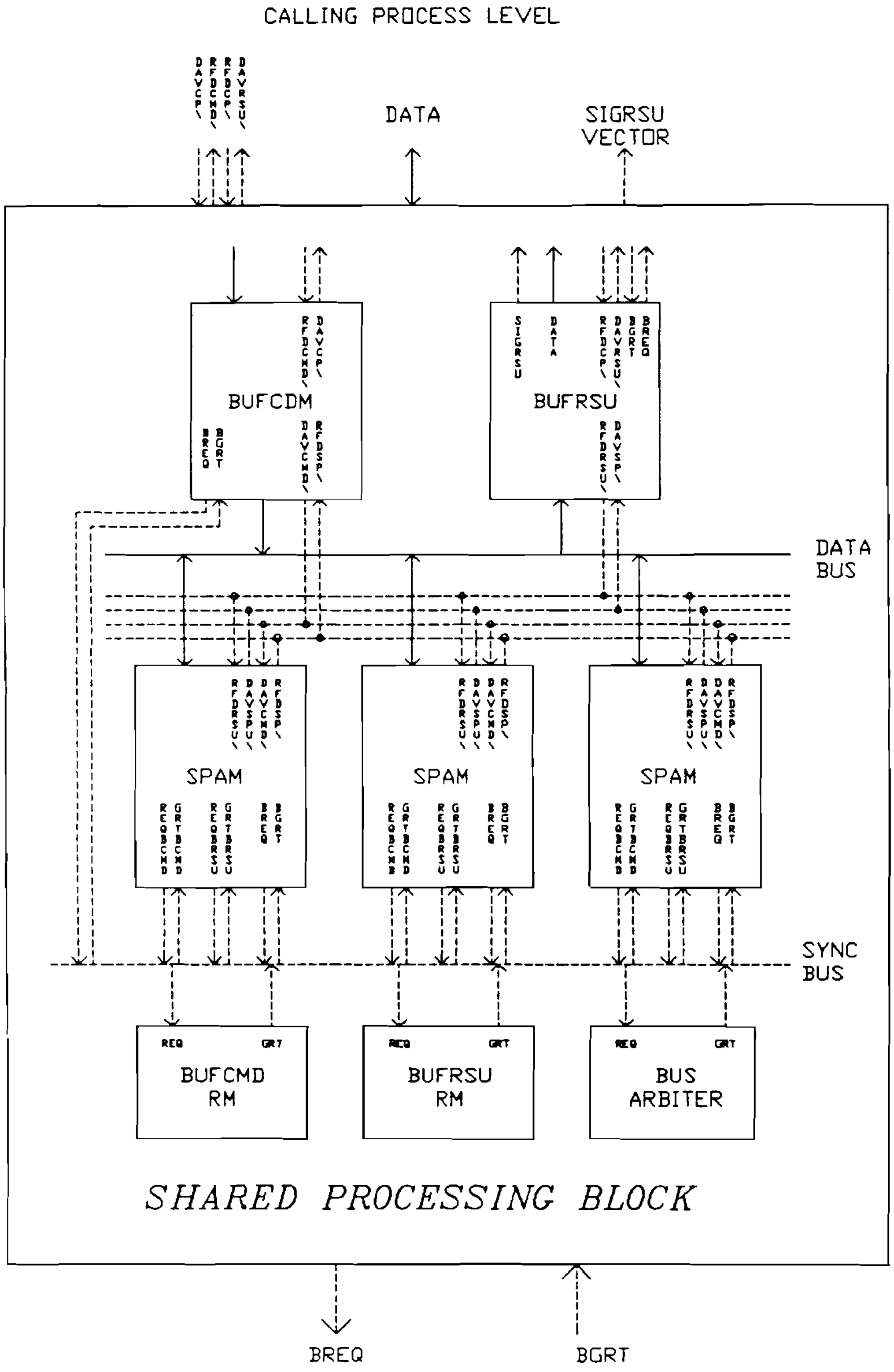


figure 59: AID of SPAM level

The BUFCMD_OUT process puts a data item on the bus, when a shared processing module is ready for data. The line (**RFDSP**) should be only active, if a **SPAM** wants data. The responsibility to release the BUFCMD_OUT is for the **SPAM**, since the **BUFCMD** doesn't know what it is putting on the bus.

Having the result a **SPAM** will request access to the **BUFRSU**. After receiving the grant the bus is requested, when the **RFDRSU** is active. The BUFRSU_IN process fetches the data.

Since for all mentioned buffer processes a resource manager is implemented, the requester terminates the communications. The processes only transport data from source to destination, so that they could be implemented in the buffer itself (see appendix C). Only the BUFRSU_OUT is more complex.

The **CPAM** knows how many result data bytes will be transmitted after decoding the result status byte, so it seems logical to give the **CPAM** the responsibility to fetch the results. But the **CPAM** is not allowed to request the bus, if it is not sure that it will get a data byte requiring a new protocol in which the buffer raises the **DAV** line. The **CPAM** tests this line and generates a bus request. Still we haven't solved the problem of transmitting the first data byte to the correct **CPAM**. The out process decoding the IDCPAM and signaling the **CPAM** that its result is ready, is the only logical solution.

The BUFRSU_OUT generates a signal (**SIGRSU**) for the correct **CPAM** and waits for the **RFDCP**. After requesting the bus it puts the data on the bus using **DAVRSU**. The **SPAM** has given to the buffer the IDCPAM and a number count telling how many bytes the result is. The out process uses this to track down the moment to generate a new **SIGRSU**.

A buffer is connected with two interfaces. We don't choose an implementation yet. (see appendix C).

In the allocation a bus arbiter is needed, since more data is transported beside the parameter and result bytes. If only two buffers access the data bus, the bus could be given to one buffer for the total communication period. But if the buffer sizes are too small, a deadlock can occur. A **CPAM** wanting to put bytes in the **BUFCMD**, which is full. It has to wait until a **SPAM** has fetched bytes. But no **SPAM** fetches data, because the **SPAM** can't store any result bytes as long as the **BUFRSU** is full. The **BUFRSU** can't be emptied, while the **CPAM** is locking the bus. The bus must be requested for every data byte.

All process specifications are given in appendix C. The main difference for the caller and shared process description in chapter 6 is that mutual exclusion primitives are introduced. The appendix discusses the communication protocol specification and its execution.

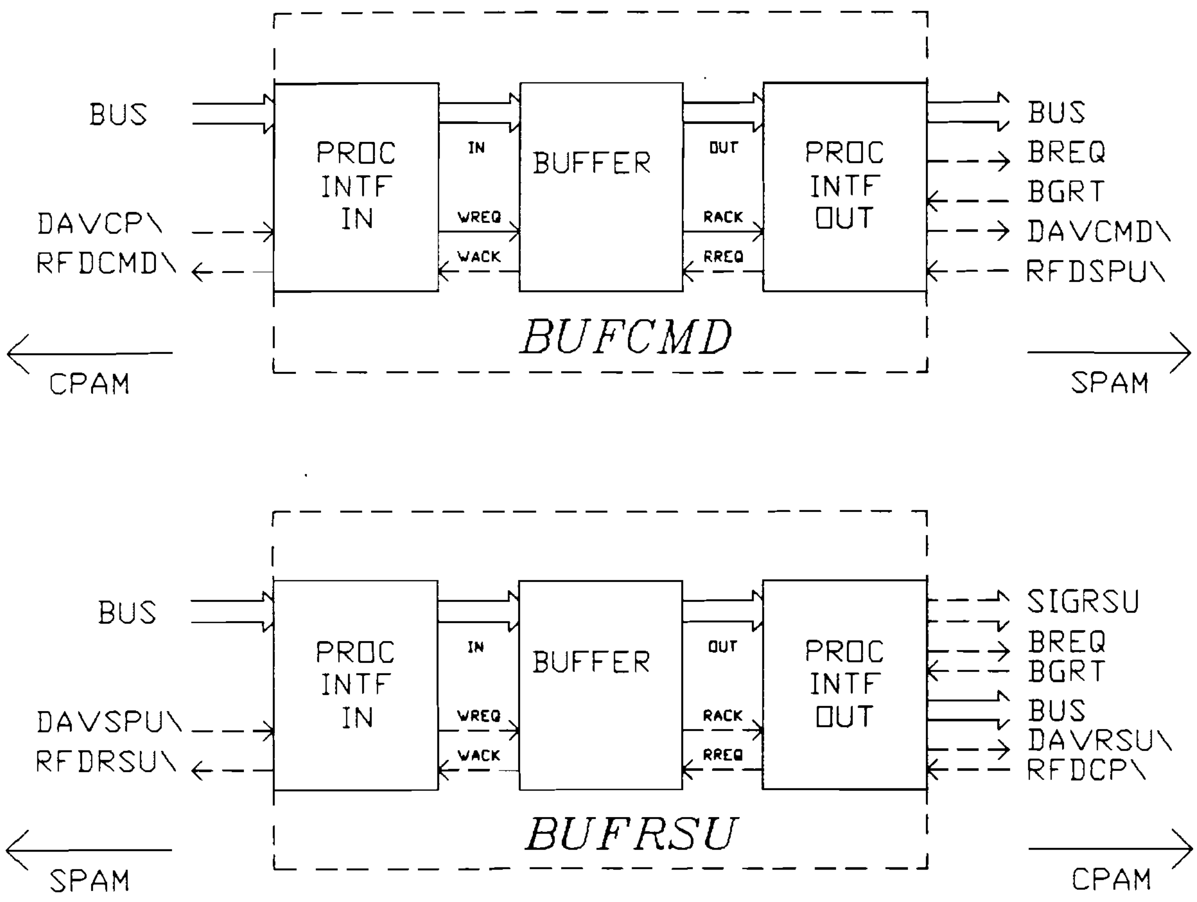


figure 60: buffer implementation

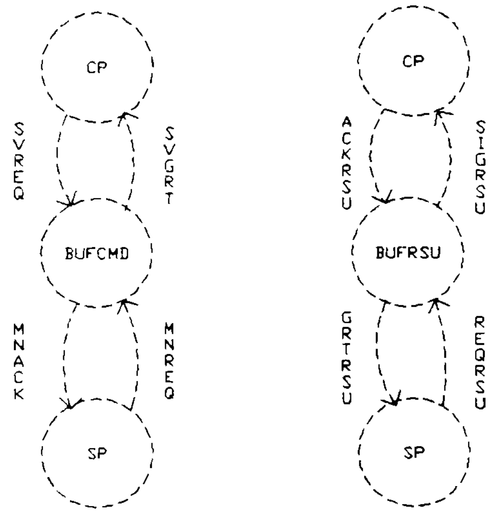
7.3. without resource management

To reduce waiting problems we introduced buffers. We used the old synchronization primitives needing resource managers for the buffers ensuring that the data is put in the correct order in the buffer or is send in the correct order to a **CPAM** or **SPAM**.

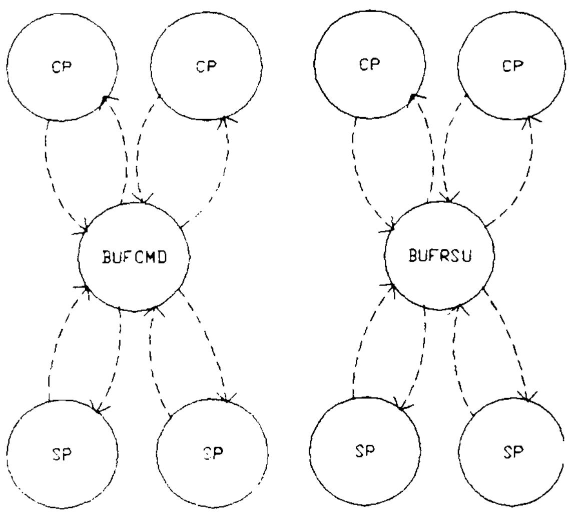
Having an identical requirements model, we use another arbitration technique resulting in a different allocation and in different PSPECs. For this allocation new primitives describing the process synchronization and interaction are needed. To optimize the number of resources, we try another allocation.

This allocation is a demonstration of the iterative loop between requirements model and architecture model. It also illustrates the complexity of the allocation process.

Servers are introduced instead of resource management. A buffer server selects a partner out of the pending requests and gives the selected partner's identity to the input or output process of the buffer, which uses the id to signal and to select the correct channels of the new partner.



PROTOCOL FOR BUFFERS



PRINCIPLE OF SERVING

figure 61: protocol for server buffer

Because only one module is communicating with the server during a certain period and the sequencing of events is know, several lines are multiplexed (see appendix C and fig 61). Hence the receiver has to know how many bytes will be sent. A number count is transmitted beside the IDSPAM. The server decodes the necessary information. Another solution uses the **SVREQ** and **MNREQ** protocols to signal the end of the communication.

The **BUFCMD** has two servers: CMD_IN_SERV and the CMD_OUT_SERV. The CMD_IN_SERV gets service requests and generates the SVGRT. After sending a **SVGRT** the input process fetches the command words of the bus and puts them in the buffer. The CMD_OUT_SERV gets monitor requests and serves them requesting the bus and putting the bytes on the bus. Since the IDCPAM and the number of bytes are known, the output process knows when the communication is finished and a new **MNREQ** can be served.

The BUFRSU has also two access servers: RSU_IN_SERV and RSU_OUT_SERV. The first one gets REQRSUs of the SPAMs. The SPAM requests the bus and puts the bytes on the bus after the server selected the SPAM and sent a **RFD**. The first byte contains the IDCPAM and the number of bytes is used by the RSU_IN_SERV to detect the end of the communication and to select a new partner.

All buffer processes until now use a number count to generate an enable for the server arbiter to select a new partner. The process counts the bytes exchanged with the partner and compares it with the number count.

The RSU_OUT_SERV generates the signal result ready (SIGRSU) for the correct CPAM out of the IDCPAM. After the acknowledge (ACKRSU) and requesting the bus, the server transmits the data bytes. When it has transmitted the indicated number of bytes, another SIGRSU is generated.

Before describing the processes we define new primitives and protocols:

A communication protocol between the SPAM and BUFRSU for getting access to the buffer BUFRSU:

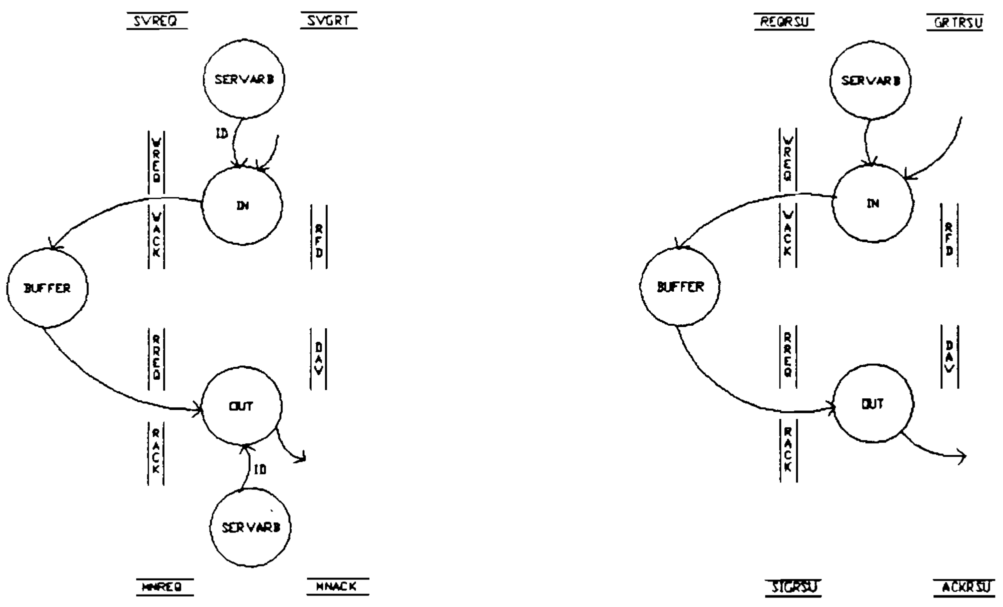
WGRTRSU(IDSPAM)	: REQRSU:=TRUE;
	REPEAT SKIP UNTIL RFDRSU;
	(wait grant access BUFRSU)

GRTRSU(IDSPAM)	: RFDSP:=TRUE; (grant access BUFRSU)
REMGRTRSU(IDSPAM)	: RFDSP:=FALSE; (remove grant access BUFRSU)
WREMGRTRSU(IDSPAM)	: REPEAT SKIP UNTIL NOT(RFDSP); (wait for remove grant BUFRSU)
REMREQRSU(IDSPAM)	: REQRSU:=FALSE; (remove request access BUFRSU)

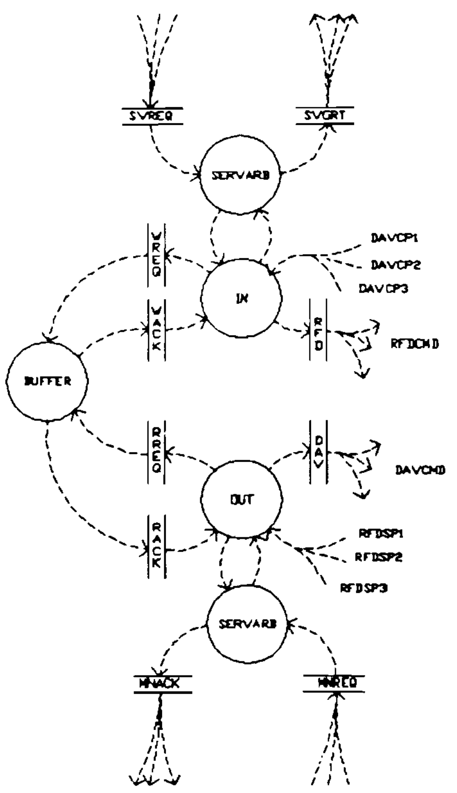
A communication protocol between the BUFRSU and the CPAM for the result signalling:

SIGRSU(IDCPAM)	: SIGRSU:=TRUE; (signal result ready)
WSIGRSU(IDCPAM)	: REPEAT SKIP UNTIL SIGRSU; (wait for signal result ready)
REMSIGRSU(IDCPAM)	: SIGRSU:=FALSE; (remove signal result ready)
WREMSIGRSU(IDCPAM)	: REPEAT SKIP UNTIL NOT(SIGRSU); (wait for remove signal result ready)
ACKRSU(IDCPAM)	: RFDCP:=TRUE; (acknowledge result ready)
REMACKRSU(IDCPAM)	: RFDCP:=FALSE;

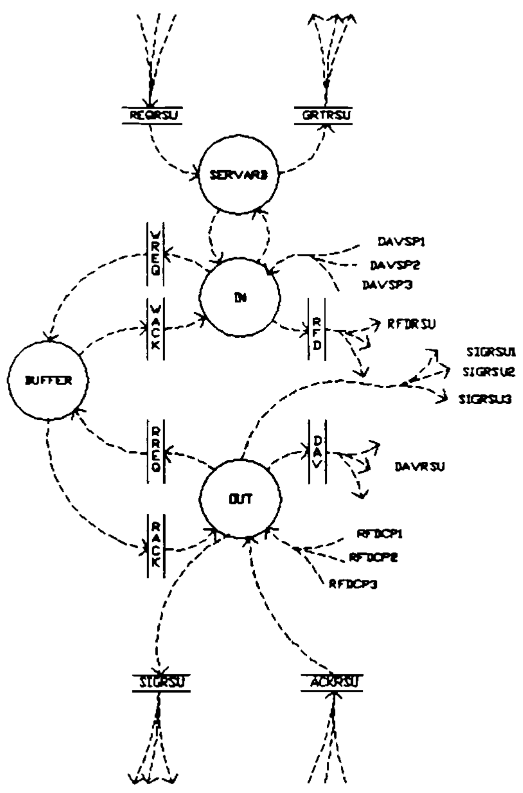
As before a handshake signal ensures the signal receipt. For the BUFCMD we use the service request and grant to get access for the CPAM and MNREQ and MNACK to get access for the SPAM.



DFD GENERAL BUFFER MODEL



CFD FOR BUFCMD



CFD FOR BUFRSU

figure 62: requirements model for buffer server

SHARED PROCESS PSPEC:

```
begin
     init
     get(par2); (* fetch parameter *)
     get(par3);
     execute
     cobegin
          status(res1,ok);(* generate status info *)
          coded(id); (* generate idcpam and number cntrsu to indicate the number of
                              result bytes *)
     coend
     wgrtrsu(idspam); (* request the bufrsu by raising reqrsu, wait for the grant rfdsp*)
     remreqrsu(idspam); (* remove request; implemented in hardware *)
     wremgrtrsu(idspam); (* this is not necessary, because signal will be raised again to
                    indicate that the buffer is ready after the lines are set up *)
     putarb(id);
     putarb(res1); (* put status on the bus *)
     if ok then
                    begin
                         putarb(res2);
                         .....
                    end;
end;
```

THE SPAM MONITOR PSPEC:

```
while true do
   begin
     mnreq(idspam);(* request new caller *)
     init; (* some initialization *)
     wmnack(idspam);
     cobegin
          remmnreq(idspam); (* this is necessary to remove mnack, which could be seen
                         as a davsp by the get primitive, so that wrong data is fetched. *)
          get(id); (* must be saved to be decoded into the result identifier *)
          get(par1);
     coend
     deccmd(procnumb,par1); (* decode command *)
     case procnumb do
          1        : proc 1;
          ..           ...
          end;
   end;
```

THE CALLER PSPEC:

```
prog;
svreq(idcpam);
cobegin
   begin
     wsvgrt(idcpam);
     remsvreq(idcpam);
     wremsvreq(idcpam); (* necessary to reset logic *)
     putarb(id); (* put the idcpam and the number count of parameters on the bus*)
     putarb(par1); (* put command on bus *)
     putarb(par2); (* put parameter on bus *)
     .......
     prog
     wsigrsu(idcpam); (* wait until a result is ready *)
     ackrsu(idcpam); (* this necessary when a parallel program is executed, which can't
          be interrupted. And also to remove the sigrsu signal, which can be seen as a
          dav, so that the get primitive is never  executed *)
     wremsigrsu(idcpam);
     remackrsu(idcpam);
     get(res1); (* get status of bus *)
     decode(res1,ok); (*decode the status info *)
     if ok
          then
             begin
                get(res2); (* get next result byte *)
                .....
             end
          else proc errorhandling;
   end;
   prog; (* parallel program *)
coend;
```

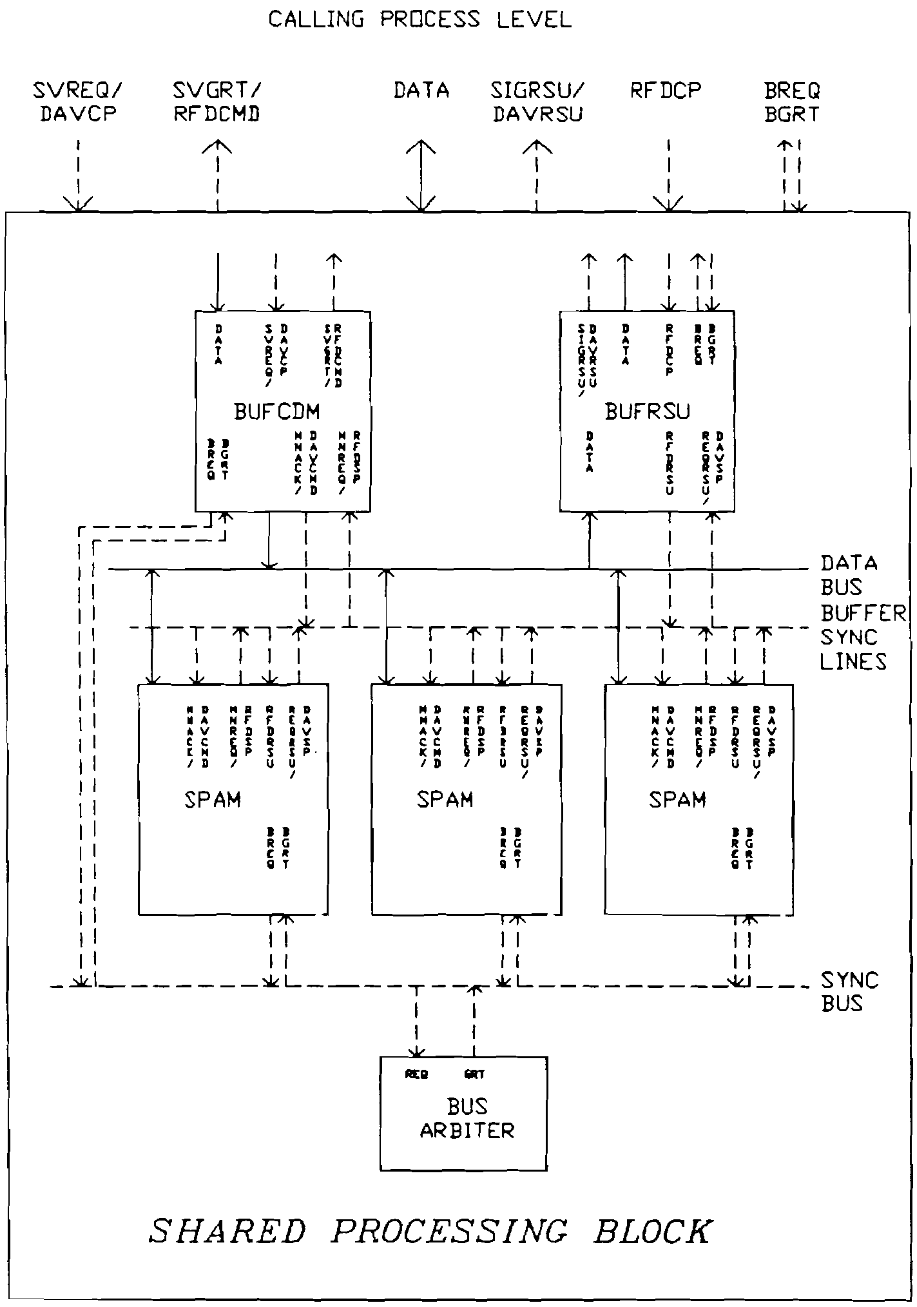


figure 63: SPAM level AFD/AID

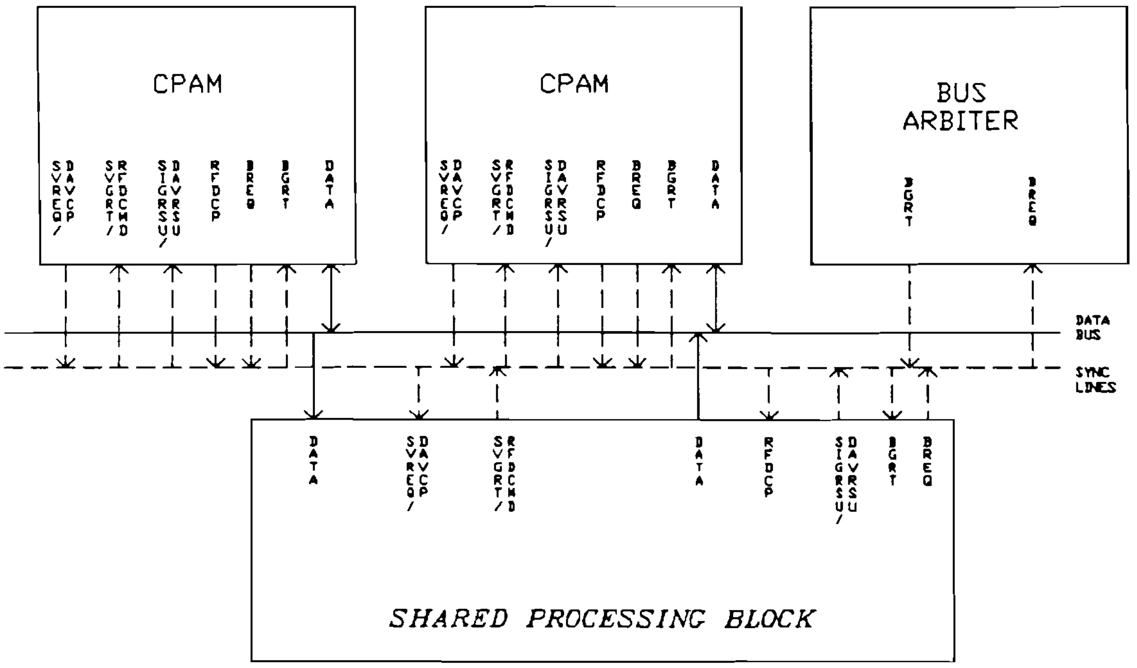


figure 64: CP level AID

The buffer processes are described in appendix C, where also the actual implementation and the protocol execution is discussed. The final conclusion in the appendix is that the resource management architecture model is easier for modeling, but has no profit in the implementation compared with the server buffers.

8. A SUMMARY OF THE METHODS FOR SHARED PROCESSING

We discussed shared processing in the last chapters and in appendix C. During this discussion we introduced several methods for data transport, serving the request, result/parameter communication, identity of the requested process, and connecting the caller and the shared processing module. These methods can be used in several combinations depending on the problem using primitives, which functions don't differ; only the implementation. We introduced service request, service grant, monitor request, monitor grant and a few result/parameters primitives.

For the data transport we can choose between the **data available and ready for data** protocol and control lines to the destination datapath. This last solution requires that the transmitter knows the state of the receiver. The data communication protocol stimulates the partners independence.

The serving arbitration can be implemented by resource management, a server arbiter or a parallel server. A server arbiter contains service request and monitor request arbiters and generates information for the **SPAM** about the shared processing request. Resource management gives a **SPAM** access to server, but doesn't generate any information. Resource management generates a clearer partitioning of modules. A parallel server processes requests concurrent and speeds up the serving. The execution is faster, but requires an enormous amount of logic.

The requested shared process can be identified by signals and by command words. The command words method can implement shared processes more economical. The first solution should be used, when only a few processes can be requested.

The result and parameter communication can be implemented by a **PTR/RDY** protocol, a signal result and data handshaking protocol or buffers with communication protocols. The first protocol requires that the caller can access the shared processing module to put and fetch the data. The others improve the communicating modules independence. The **DAV/RFD** protocol differs from the normal **DAV/RFD**, because these signals indicate that shared processing data is present (e.g. result ready). The buffer introduces even more flexibility. A buffer speeds up the program execution and allows a module to generate more service request. The buffer can be implemented using resource management or servers. The server solution needs more hardware and an identifier to indicate the number of bytes to transmit.

The shared modules select the correct partner (and so the signal lines) with a multiplexer. Several primitive channels can be multiplexed on one physical line. A multiplexer isn't redundant for buffer communication involving resource management.

Combining the methods we introduced three main techniques. First, the hardwired calls model: the service request identifies the requested shared process. A special module (server) controls the set up and maintenance of the connection and gives the **SPAM** the requested process ID. To simplify the solution we moved the responsibility for the connection's maintenance to the SPAM. The disconnection and connection has to be done in the right moment otherwise signals will be misinterpreted.

In the next solution we introduced the command word. After getting a start signal the server takes care of scheduling the partners (requester and SPAM) and set up of connections by generating an requester's ID. The connection unit is allocated to the SPAM. Still the connection and disconnection of the primitive signal lines between the CPAM and the CP had to be executed very carefully.

In the last model we don't have a direct connection between the CPAM and the SPAM, but a buffer reducing the waiting time of the several modules. Giving the buffer a server requires request and signal lines. and multiplexers, which need more hardware and an identifier to tell the server how many bytes belong together. These problems could be solved introducing resource managers. Beside the resource managers a result primitive and some special DAV/RFD protocols between partners were required. More physical lines are needed in this solution, which is easier to design.

9. AN ARCHITECTURE MODEL OVERVIEW

An architecture model is defined with several modules communicating with each other using synchronization primitives.The final requirements model translates into several processing modules executing parallel processes. *Each final module processes transformations sequentially.* The scheduler module checks process scheduling conditions. The processing modules are designed to execute application specific processes and so they can't be reprogrammed. The program is located in a memory/state table in the module. The bus can't become a bottleneck, since the modules don't need to access shared memory to fetch the program code, but a module can access external resources. Local memory implementation is only possible, because of the transformations allocation to modules reducing flexibility. In the next chapter we discuss a technique introducing more flexibility. A resource manager processes the requests to access shared resources, executing several requests in parallel.

*In the final implementation the calling (*CPAM*) and the shared processing modules (*SPAM*) don't differ. Every module can execute requested and normal processes.* The process execution depends on scheduling and dispatching information in a pure calling module (CPAM) and on a command/request and dispatching in a shared processing module (SPAM).

The SPAM, buffers and CPAM can use the same data bus and bus arbiter, since the different levels introduced for the shared processing buffers don't exist. In the implementation the busses are reduced to one by mapping the two logical busses on a physical bus. If different levels are used, a SPAM module could not call another module unless this module is on a third level or extra server buffers connect the two levels in the opposite direction. We use two levels, if the bus traffic is very heavy requiring a second data bus, allowing introduction of other processing modules at the SPAM level.

The architecture model is recursive (fig 65). Each PAM *can consists of identical modules at the next level. Besides using the mutual bus modules can exchange information in a private channel using R/A synchronization. Therefore info lines, data lines and control lines are implemented between the two modules.*

A difficulty in visualizing the model is that methods are chosen in the allocation (e.g. shared processing, control of shared resources: protocol or multiplexed control lines), but a general model has to contain them all. Our goal is to give an overview of the model and to show the interconnections (channels) between the modules. Depending on the requirements these channels are connected to a module. Figure doesn't contain all possible channel connections.

The model and the synchronization statements are illustrated in appendix A using the producer/consumer problem. The techniques for SAMs and RMAMs implementation are discussed in appendix B. *A bottom level* PAM *consists of a datapath and controller. The datapath contains private resources, shared resources protected by the* RMAM, *shared resource control by the R/A synchronization and shared resource that don't need any protection. The controller generates control signals for its own datapath, for datapaths containing shared resources and for synchronization. The controller tests the information of its datapath, shared resources,* RMAM, BAM *and* SVAM.

The following channels are defined:

BAM	= BGRT+BREQ	* arbiter channel *
RMAM	= {GRT+REQ}	* resource management *
SAM	= {REQ + GRT} + {SIG}	
		* scheduling *
SVAM	= {SVREQ + SVGRT} + {MNREQ + MNACK} +	
	[{PTR + RDY} | {DAV + RFD}]	
		* shared processing *
RAD	= DAV + RFD	* data exchange *
INFO	= {\register info\}	
Z	= {\register control signal\}	
DATABUS	= {\data bit\}	

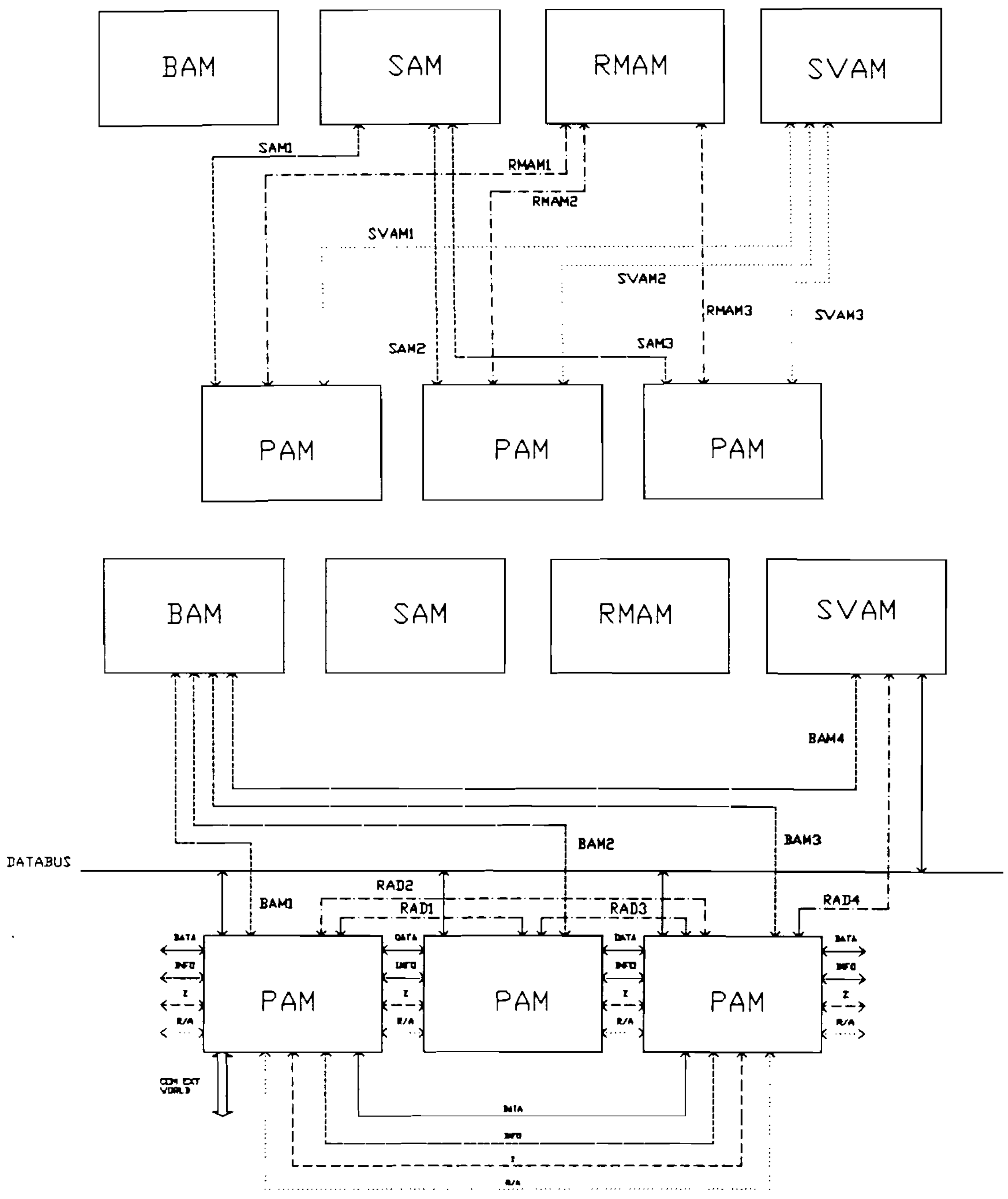


figure 65: general architecure model

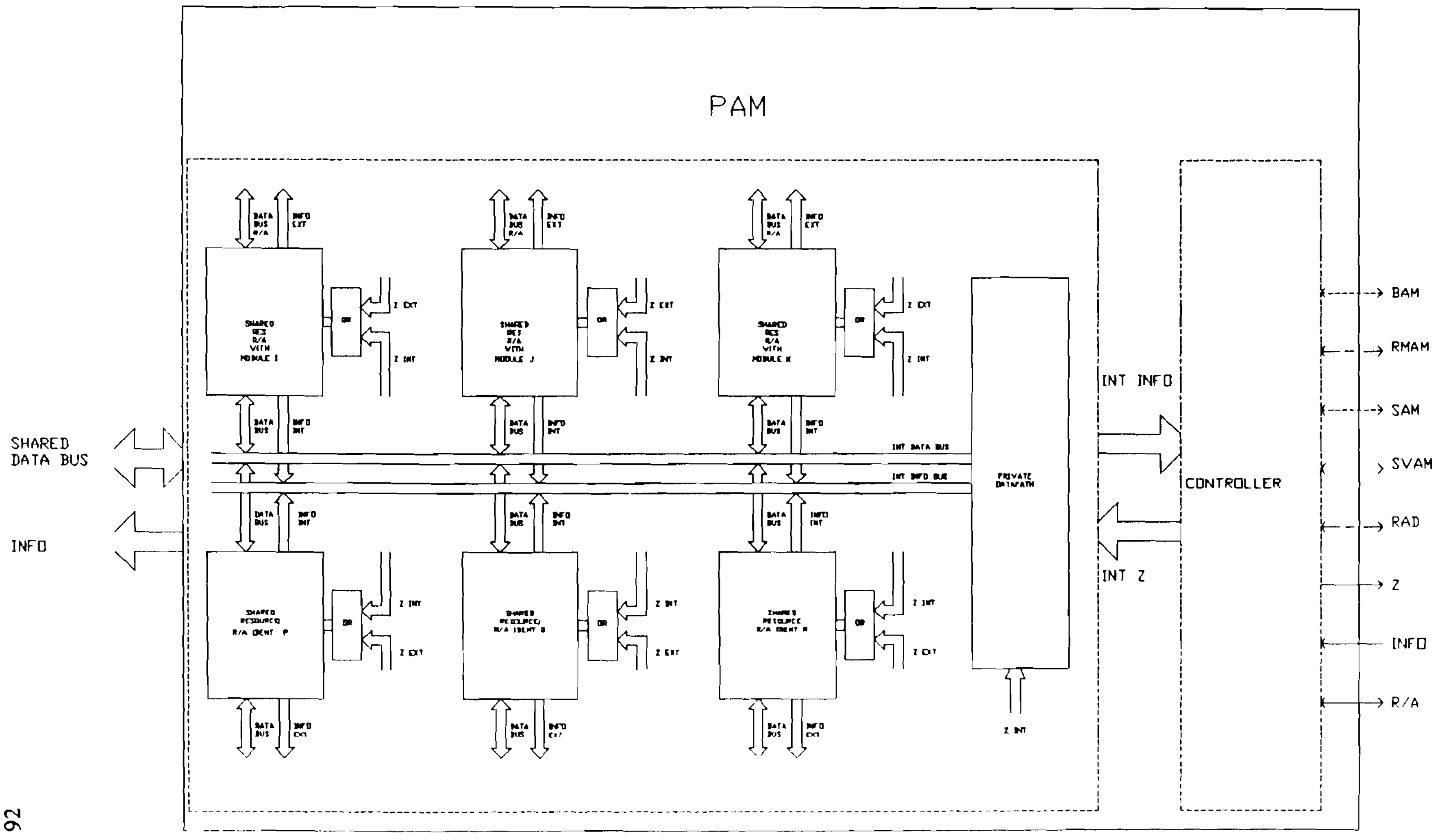

PAM
SHARED DATA BUS
INFO
DATA BUS R/A
INFO EXT
SHARED RES R/A WITH MODULE I
SHARED RES R/A WITH MODULE J
SHARED RES R/A WITH MODULE K
OR
Z EXT
Z INT
DATA BUS
INFO INT
INT DATA BUS
INT INFO BUS
PRIVATE DATAPATH
SHARED RESOURCE R/A IDENT P
SHARED RESOURCE R/A IDENT Q
SHARED RESOURCE R/A IDENT R
INT INFO
INT Z
CONTROLLER
BAM
RMAM
SAM
SVAM
RAD
Z
INFO
R/A

figure 66: general PAM architecture model

10. PROCESSING ARCHITECTURE MODULE IMPLEMENTATION

10.1. introduction

We already discussed a few special architecture modules: the bus arbiter, the resource manager, the scheduler and some units for shared processing. To all these units control transformations executing the synchronization primitives were allocated. Now we will address two problems of the most important architecture module (**PAM**): the tasks execution (dispatching) and the implementation of the primitives.

All final architecture modules are sequential circuits. Concurrency is implemented by allocating the concurrent program to separate modules. This allocation assigns tasks to the module and produces timing requirements for the internal module task execution. The process condition checking using the process timing requirements based on events generated by other modules is done by the scheduling module. The tasks still have to be scheduled in an execution order in the processing architecture module. We will call that selection process dispatching as in operating systems. A special task (monitor) takes care of the task dispatching; a consequence of the allocation trying to optimize resource use. The timing requirements for the task execution are extended with implementation restrictions.

We discuss implementation restrictions caused by the problem definition and by the transformation allocation and also a solution for these problems. The final implementation of the primitives can be deduced from the methods described in appendix B.

10.2. task dispatching

A process describes a functional unit of statements, the result of the functional problem decomposition. A task is an unit used for dispatching and is an implementation result. A process can be a complete task, but can also be divided into several tasks.

We allocated several tasks to one **PAM** and so to one datapath. The monitor task chooses one runnable task using selection criteria depending on the problem and the implementation requirements. Whenever a new task is needed, the monitor is started calling the selected task and checking the tasks scheduling condition status (true or false) or the status (runnable or waiting) of tasks without external scheduling conditions. The scheduling module indicates which task is allowed to be executed after checking the scheduling conditions. A schedule module (SAM) communicates with several architecture modules taking care of asynchronous events. We implement one module for every independent scheduling condition.

Processes are redefined, avoiding idling of the module caused by busy-waiting of synchronization primitives. During the busy-waiting period, another task can be executed, as long as the status of the busy-waiting task is saved. *Processes are split in several tasks using the primitives* **REMTASK** *and* **SIGTASK** *signalling dispatching conditions using other names for process synchronization to stress that these primitives aren't introduced by the problem but by the implementation. They don't require handshaking, since the signals aren't exchanged between two parallel operating modules, but are used in one module.*

```
SIGTASK(ID)        : TASKID:=RUNNABLE;
REMTASK(ID)        : TASKID:=WAITING;
```

The **SIGTASK** primitive signals that a task waiting for the detected synchronization event is now runnable and the **REMTASK** primitive is executed by the running task to reset the runnable flag. A task bit is reset by **REMTASK** and set by the interrupt routine using **SIGTASK** primitives.

10.3. primitive implementation

A synchronization primitive either signals events to other tasks or waits for the occurrence of an event (busy-waiting implementation). Using a control unit and datapath to implement the **PAM**, the control unit checks the event occurrence. Using an interrupt mechanism busy-waiting is avoided. The event is an interrupt starting a task (interrupt routine). After the interrupt handling, the interrupted task is resumed.

The interrupt routine takes care of sensitive program parts, which have to be executed quickly. *A new interrupt during interrupt handling can cause the partner architecture module to stay busy-waiting. Therefore we won't allow that an interrupt routine is interrupted. The event signalled by the interrupt is handled without saving the contents of the datapath (the control status is saved).* This restriction can be dropped and then a special task saves and restores the processing architecture module (datapath and control) status.

Some program parts containing handshake primitives may not be interrupted causing contention and slowing down task execution in other modules. The mutual exclusion section for shared resource protection starting with **WAITRES** primitive and ending with **SIGRES** primitive may not be interrupted (bus traffic including). The request for the shared resources disables and the removal of the request enables the interrupt handling.

```
DIH                 DIH
GET(PAR1)           PUT(RES1)
GET(PAR2)           PUT(RES2)
GET(PAR3)           PUT(RES3)
EIH                 EIH
```

figure 67: example of DIH and EIH

Data communication specially the parameter and result communication of shared processing should not be interrupted. We can't use the data communication primitives (put and get) to disable and enable interrupt handling, since the interrupt handling will be enabled between two primitives. The whole section should be protected by new primitives. We introduce DIH (DISABLE INTERRUPT HANDLING) and EIH (ENABLE

INTERRUPT HANDLING), internal module primitives telling the control unit of the **PAM** what to do with a pending interrupt (see fig 67).

Using the new primitives the mutual exclusion primitives are defined (allowing busy-waiting):

```
    WAITRES :          DIH; (* disable interrupt *)
                       REQ:=TRUE;
                       REPEAT SKIP UNTIL GRT;

    RELRES :           REQ:=FALSE;
                       EIH; (* enable interrupt *)
                       REPEAT SKIP UNTIL NOT(GRT);
```

The interrupt mechanism fits in two situations. During the program execution, a result is available, before the program can process it. To avoid idling of the producer module, the interrupt routine fetches the result and stores it temporally. This technique is used with R/A synchronization and for shared processing.

In the second situation a module program is busy-waiting being at a point, where it needs results, but they aren't ready yet. So the module starts executing another task, which is interrupted when the result is ready. The interrupt routine stores the results temporally. After finishing the required task, the suspended process can be resumed depending on the dispatcher algorithm and timing constraints. Other idling situation are the **SVGRANT** and **WAITPROC** primitive execution and where no call is pending in a **SPAM**. The process has to be split into several tasks.

A module waiting for a service grant of a shared processing module, executes another task and when the grant arrives, an interrupt routine puts the parameters on the bus. A **WAITPROC** primitive will have to be removed by task redefinition, if several processes are executable in one **PAM**.

If a **SPAM** can also execute normal tasks, which aren't called, the result and parameter communication can be given to an interrupt routine. Requesting a new caller, busy-waiting can occur, since no call is pending or the server/buffer is serving another SPAM. The parameters are fetched on interrupt basis, so that another task executes while waiting.

The result communication doesn't cause problems. When no buffer is used, the caller is interrupted. The handshake signal implements an interrupt in a **SPAM** to increase the flexibility. A buffer implementation can cause busy-waiting, serving another module or being full (design error). It is usually a waste of resources to implement interrupt handling, since the buffer is introduced to speed up execution and to reduce waiting. If the buffer is a bottleneck, just expand the size.

An interrupt routine can't be implemented for parameter communication in a **SPAM** using a resource manager to protect the buffer communication. After requesting the command buffer, the module waits for the access grant, but keeps waiting, when the buffer is empty. We can't execute another task, because the shared resource primitives forbid interrupts. We would like to treat the access grant and the communication signal as an interrupt. The interrupt routine fetches the parameters and releases the buffer. By introducing an interrupt routine the **SPAM** can execute an allocated task, which isn't called by another module.

The **WAITRES** primitive changes into a **REQRESI** primitive, stopping the task's execution. The **GRTRESI** primitive generates an interrupt. The partner of the **REQRESI** is the **RELRESI** primitive. The character I indicates that interrupts implements the primitives. Only the parameter communication in a **SPAM** with a command buffer will be on interrupt basis. See for further discussion about this technique appendix D.

10.4. process/task synchronization

To avoid idling we divide processes into several tasks and implement the event as an interrupt. Now we pay attention to the technique and redefine synchronization schemes for primitives.

The **WAITPROC(ID)** primitive requests the scheduling module to check the scheduling condition (see fig 69). A handshake protocol ensures that a grant isn't interpreted more then once. The signalled task implements the synchronization protocol. When in a module only one process can be executed, we will still use **WAITPROC(ID)** and won't implement the monitor. Busy-waiting is allowed, since no other tasks can be executed.

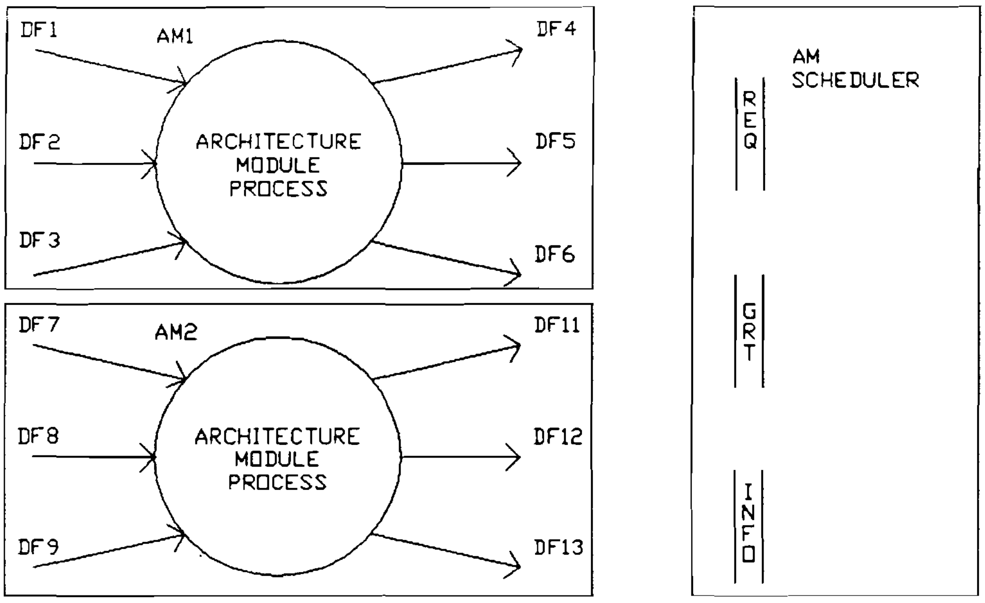
The synchronization schemes are (the corresponding CF can be found in figure 69):

```
TASK:
    REMREQPROC(IDPROC); (* 3; CF1; remove request for scheduling process *)
    WREMGRTPROC(IDPROC); (* 4; CF4; wait for removal of running grant *)
    PROG (* contains no process and task synchronization*)
    REQPROC(IDPROC); (* 1; CF1; start again the check of the scheduling condition *)

THE SCHEDULER: (* parallel running with the processing architecture modules, one for
    every condition, busy-waiting allowed *)
    WHILE TRUE DO
        BEGIN
            WREQPROC(IDPROC);(* 1; CF11; wait for the request to check scheduling
                condition*)
            REPEAT SKIP UNTIL SCHEDULE__CONDITION; (* test timer signal, external
                event, sigproc signal *)
            GRTPROC(IDPROC); (* 2; CF13; signal condition true *)
            WREMREQPROC(IDPROC); (* 3; CF11; wait for acknowledge receipt of
                condition      true signal *)
            REMGRTPROC(IDPROC); (* 4; CF13; remove signal *)
        END
```

TASK PRIMITIVES	SCHEDULE PRIMITIVES
1 REQPROC	WREQPROC
2 -	GRTPROC
3 REMREQPROC	WREMREQPROC
4 WREMGRTPROC	REMGRTPROC

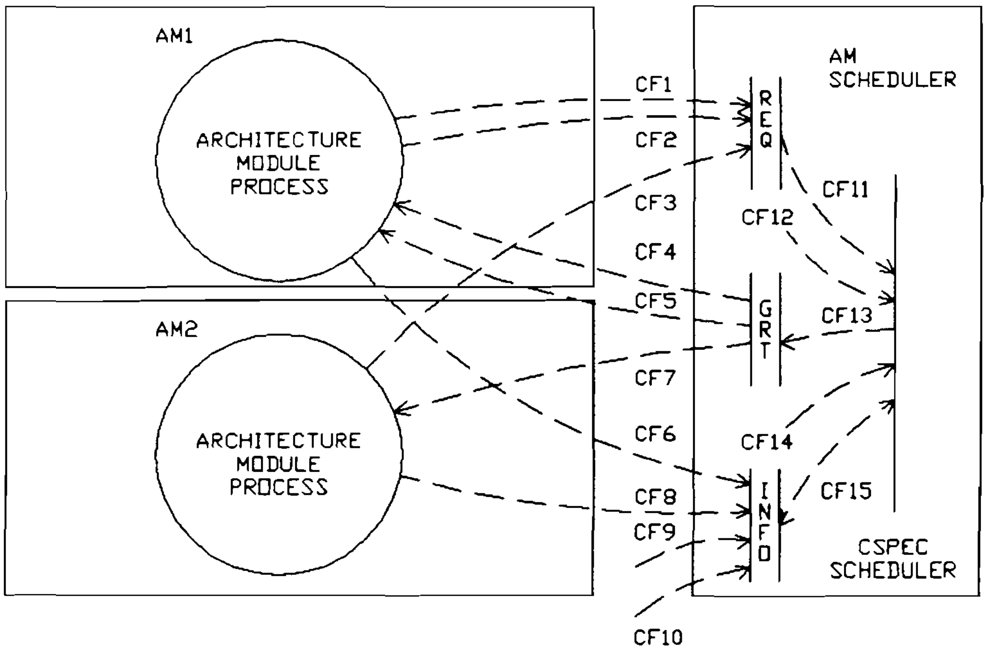
1. start the check of the scheduling condition
2. condition is true
3. task is started
4. removal grant

figure 68: relation of primitive actions



DFD COMBINED
WITH AFD AND AIF

(DF GOES TO OTHER MODULES VIA DATABUS)



CFD COMBINED
WITH AFD AND AIF

figure 69: requirements and architecture model; AFD and AIF for two AMs and a schedule AM (AM 1 contains tow schedule processes)

The task is started after the monitor noticed the grant of the schedule architecture module. The primitive **SIGPROC** doesn't change. In figure 69 the scheduler is the control transformation described with **CSPEC** enabling and disabling processes. Every independent **REQPROC** identifier gets one **CSPEC**.

The internal primitives **SIGTASK** and **REMTASK** remove the **WSIGRSU** (shared processing without buffer), **WSIGRDY** (with buffer), **WSVGRT** (shared processing), **WAITPROC** (process synchronization) primitives.

We discuss an example to illustrate the technique. We have the following requirements model process description:

```
PROC A
BEGIN
    REMREQPROC(IDPROCA);
    WREMGRTPROC(IDPROCA);
    PROG A1
    WSIGRSU(IDSPAM1);
    PROG A2
    REQPROC(IDPROCA);
END
```

Somewhere a request is generated for the shared process, but this is of no interest for the illustration. The implementation description is:

```
TASK A1;
BEGIN
    REMREQPROC(IDPROCA);
    WREMGRTPROC(IDPROCA);
    PROG A1
    SAVE(PROC__STAT); (* save task status *)
END

TASK A2;
BEGIN
    REMTASK(IDTASKA2);
    RESTORE(PROC__STAT); (* restore task status *)
    PROG A2
    REQPROC(IDPROCA);
END
```

An interrupt routine takes care of the SIGRSU signal and the corresponding actions and makes TASK A2 runnable. The primitives SAVE and RESTORE depend on the process and the PAM implementation structure (datapath and control). In complex situation these primitives can be implemented as a task with parameters.

The same technique is used for R/A synchronization executing between two parallel tasks allocated to different architecture modules. It is a simple form of process synchronization in which a condition isn't checked. A request is a start, while in the process scheduling synchronization it only starts the condition check.

The technique is introduced in two applications. Either the data flow between the two processes is in one direction or in both. Dataflow in both directions occurs when one process requests the other process to execute a function requiring result and parameter communication (coprocessing). The control structure is defined in figure 70. The requirements model of coprocessing:

```
PROC 1
BEGIN
   PROG 1A
   PAR COM (DF2)
   REQ(ID)
   PROG 1B
   WACK(ID)
   FETCH (DF3)
   PROG 1C
END
```

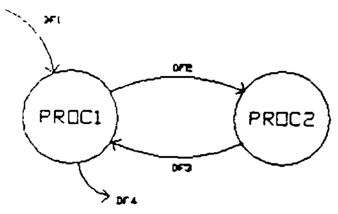
```
PROC 2
BEGIN
   WREQ(ID)
   PROG 2A
   ACK(ID)
   PROG 2B
END
```

The processes don't contain any other synchronization then mentioned. The allocation description is:

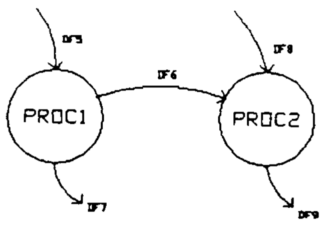
```
TASK 1A
BEGIN
  REMREQPROC(IDPROC1);CF5
  WREMGRTPROC(IDPROC2);CF6
  PROG 1A
  PAR COM
  REQ(IDPROC2); CF4
  PROG 1B
  SAVE(PROC_STAT); DF10
END;
```

```
PROC2
BEGIN
  REMREQ(IDPROC2); CF1
  PROG 2A
  ACK(ID); CF2
  PROG 2B
END;
```

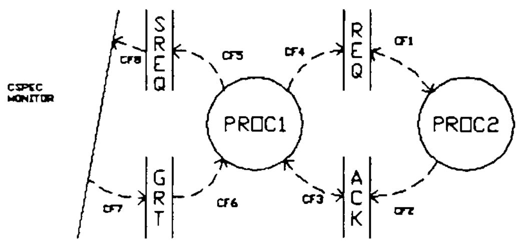
```
TASK 1B
BEGIN
  REMACK(ID); CF3
  RESTORE(PROC_STAT); DF10
  FETCH
  PROG 1C
  REQPROC(IDPROC1); CF5
END;
```



DFD COPROCESSING



DFD PIPELINING



CFD OF COPROCESSING AND PIPELINING

figure 70a: DFD and CFD for r/a synchronization

We introduced a set of four primitives: **REQ**, **REMREQ**, **ACK** and **REMACK**. Because of the dispatching in the modules, the wait primitives are replaced by primitives generating the acknowledge signal. The monitor detects the **REQ** and the **ACK**. We don't need any protection for the control stores (**REQ** and **ACK**) accessed by the primitives. Note that we haven't protected the parameter and result communication, so the implementation must allow this.

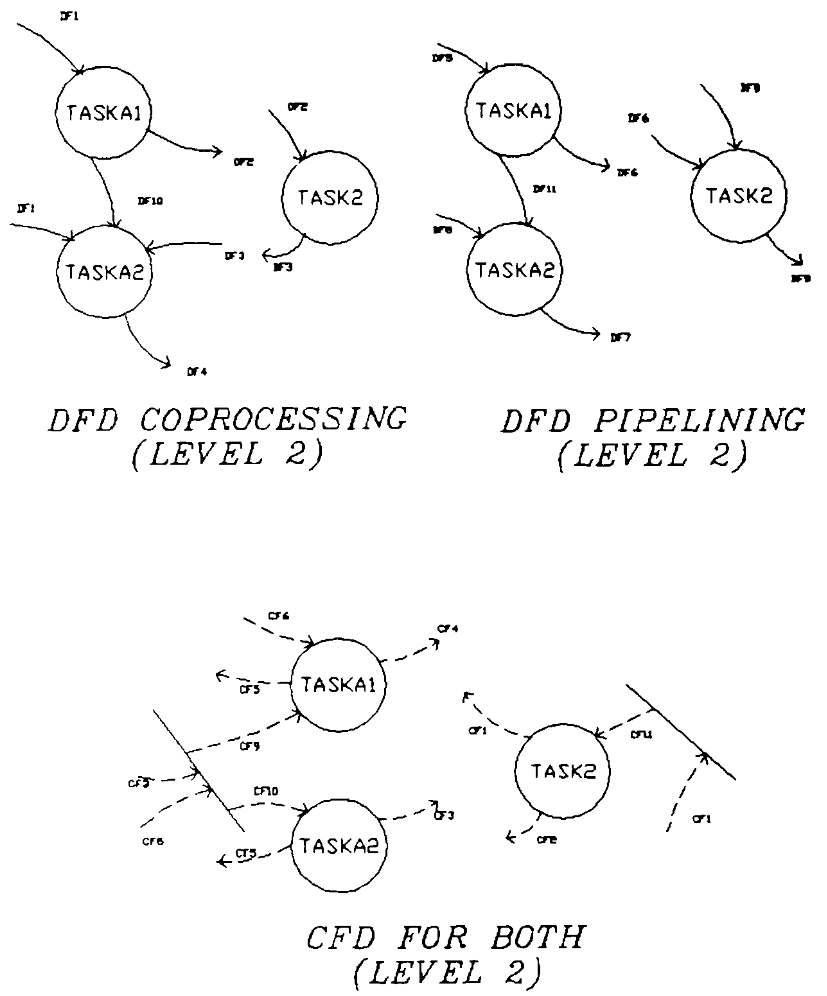


figure 70b: redefined requirements model

The R/A synchronization can be used for pipelining and so to increase the throughput. The allocation description is:

```
TASK 1A                              PROC2
BEGIN                                BEGIN
  REMREQPROC(IDPROC1);                 REMREQ(IDPROC2);
  WREMGRTPROC(IDPROC1);                (NO HANDSHAKE)
  PROG 1A                              PROG 2A
  REQ(IDPROC2);                        ACK(ID)
  PROG 1B                              PROC 2B
  SAVE(PROC_STAT); DF11              END;
END;

TASK 1B
BEGIN
  REMACK(ID);
  RESTORE(PROC_STAT);
  PROG 1C
  REQPROC(IDPROC1);
END;
```

This doesn't differ from coprocessing. Somewhere the result of TASK 1A is given to PROC 2. The program takes care of the correct sequencing. A busy-waiting implementation can be used in proc1 (task 1 and task 2 together), if the architecture module containing proc 2 stores the result of proc1 quickly.

10.5. the monitor

Tasks are runnable after an interrupt occurrence or when the scheduler signals the occurrence of the condition. Selecting a task to run, the monitor tests outputs from the scheduling module and information about internal tasks. Primitive executions changes the state of the task (see figure 71).

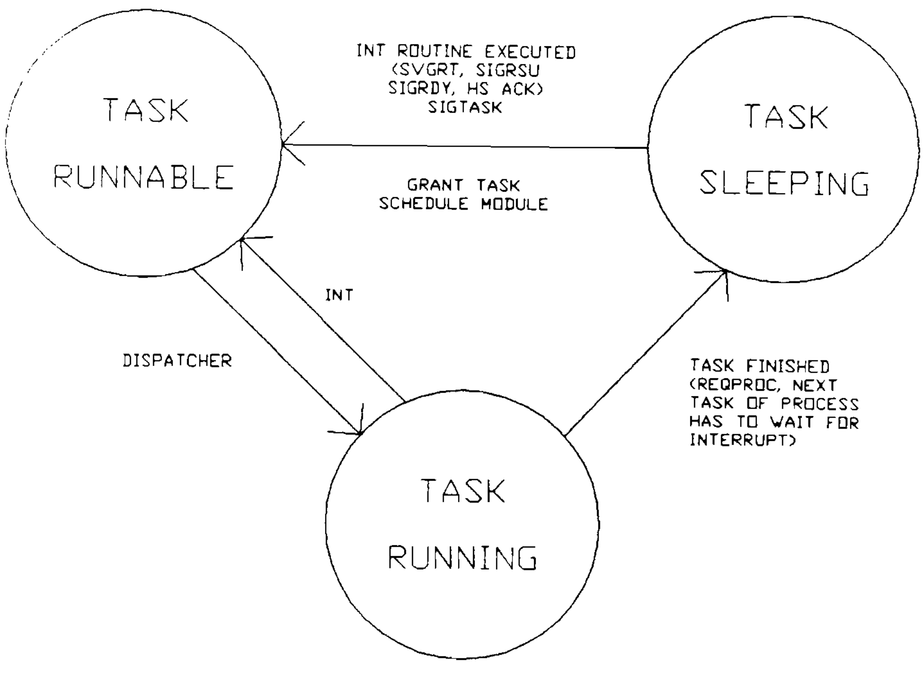

TASK RUNNABLE
INT ROUTINE EXECUTED (SVGRT, SIGRSU SIGRDY, HS ACK) SIGTASK
TASK SLEEPING
GRANT TASK SCHEDULE MODULE
INT
DISPATCHER
TASK FINISHED (REQPROC, NEXT TASK OF PROCESS HAS TO WAIT FOR INTERRUPT)
TASK RUNNING

figure 71: task states

```
MONITOR
BEGIN
    WHILE TRUE DO
    BEGIN
        DIH; (* to avoid that the task info is changed during the execution of the
            dispatch algorithm *)
        DISPATCH(PROCID, SHEDINFO,TASKINFO);
        EIH;
        CASE PROCID DO
            ID1    : TASK ID1
            .......
          END;
    END;
END;
```

The task's selection depends on an algorithm using the timing requirements of the allocated tasks. All constraints must be met and the allocation must make this possible. Several selection algorithms exists in the literature. [Janson,1985] [Kylstra,1984] The solution will be dictated by the problem.

The chosen dispatcher algorithm is called DISPATCH. A task id (PROCID) gives the identity of the selected task. The dispatcher program uses two lists. The list SHEDINFO contains tasks, whose schedule condition are true and the list TASKINFO contains all tasks, for which interrupts have occurred indicating that these tasks are runnable.

The dispatcher should be simple, but it can be implemented in a separate architecture module. The execution is faster requiring protection of the task store (shared resource), extra logic and synchronization signals. Since the dispatching was introduced to save resources, the allocation of the program to another module makes no sense.

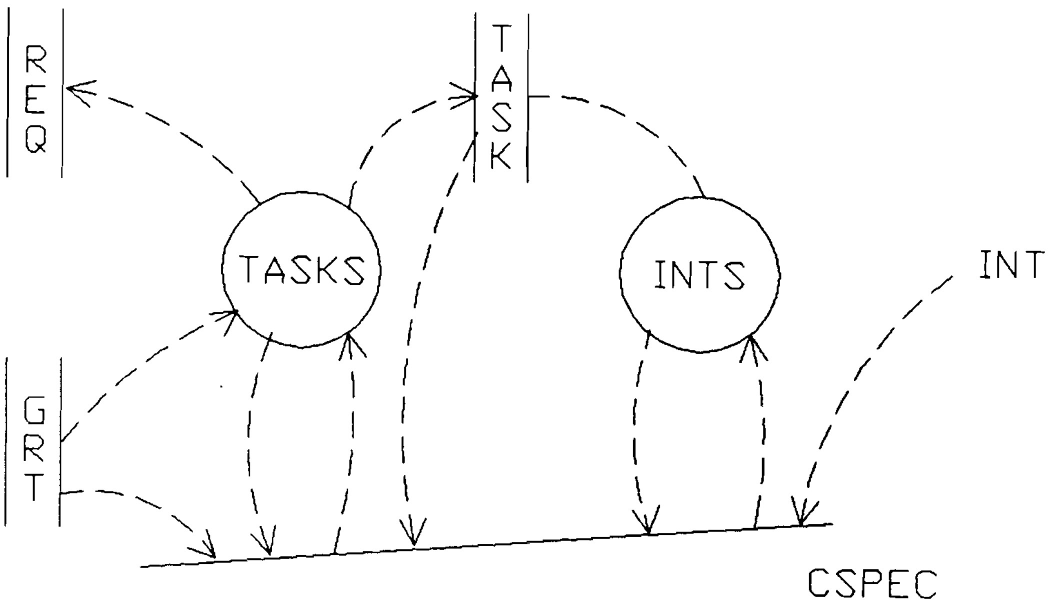


figure 72: CFD of AM allocation.

Beside the monitor the module has an interrupt handler (see figure 72). After detecting the interrupt the control logic starts the main interrupt routine. The literature describes several architectures for the interrupt logic.

The interrupt is removed by the handshaking mechanism implemented in the primitive generating the interrupt.

```
INT MAIN;
BEGIN
    DIH;
    SAVE_ID_INT_TASK; (* save identity and control status of the interrupted task *)
    CASE INTTYPE DO
        1     : TASK INT1; (* procedure call *)
        .....
      END;
    RESTORE_ID_INT_TASK; (* restore task execution *)
    EIH;
END;
```

To keep the interrupt architecture simple and because of the type of interrupt handling programs (communication), we don't allow the interrupt handling to be interrupted. The save and restore implementation depends on the architectural implementation of the module. An interrupt logic block generates an interrupt identity (INTTYPE). Since all possible interrupts are known, we design for every event its own interrupt handler.

The general interrupt task structure:

```
INT TASK;
BEGIN
    PROG
    SIGTASK(IDTASK); (* indicating that the task is runnable now, since the event has
              occurred where the task was waiting for *)
END;
```

Two examples illustrate the interrupt handling technique. Appendix D contains more examples.

```
EXAMPLE
INT_SHARED_PROC_RESULT_COM; (* interrupt is SIGRSU *)
BEGIN
    GET(RES1);
    IF STATUS(RES1)=OK THEN
                        BEGIN
                            GET(RES2)
                            GET(RES3)
                            ......
                        END;
    SIGTASK(ID...);
END;
```

We need extra resource to store the result temporally.

```
EXAMPLE
INT_SHARED_PROC_PAR_COM; (* interrupt is the SVGRT signal *)
BEGIN
    REMSVREQ(IDCP); (* handshake signal to indicate that the interrupt signal is received *)
    PUT(PAR1); (* put command on bus *)
    PUT(PAR2); (* put parameter on bus *)
    SIGTASK(ID...)
END;
```

The **SIGTASK** primitive is not always necessary. In the last example we can skip it, if the main program task isn't stopped at the point, where the parameters are communicated.

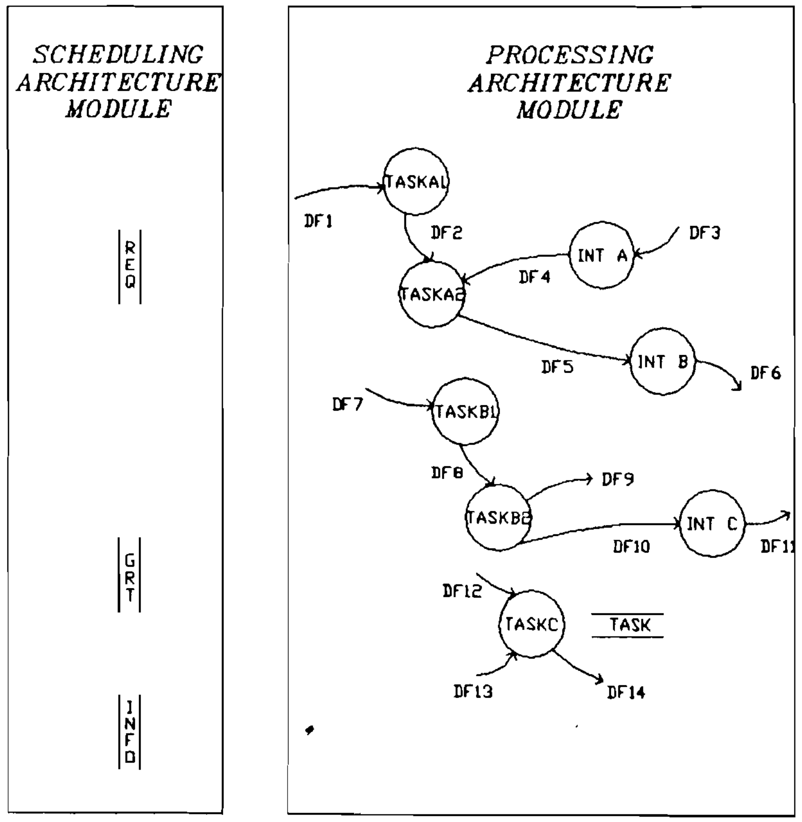
The complete description of the control transformation for a datapath has the following structure:

```
BEGIN
    INTMAIN
    INT1
    INT2
    ....
    DISPATCH
    TASK1
    TASK2
    ...
    MONITOR
    BEGIN
        INIT; (* initialize requests to scheduler *)
        MONITOR;
    END;
END;
```

In figure 73 we see the **CFD** and **DFD** of an imaginary processing architecture module. The figure is a refinement of figure 72. Every task is enabled by the **CSPEC** and generates control flows for the **CSPEC** (e.g. ready). The **CSPEC** sequences the tasks, since the allocation forbids parallel implementation except for interrupt recognition. Two control transformations are introduced, the monitor and the main interrupt routine. No link exists between the two control transformations at this level. The implementation using a control unit and a datapath will connect them.

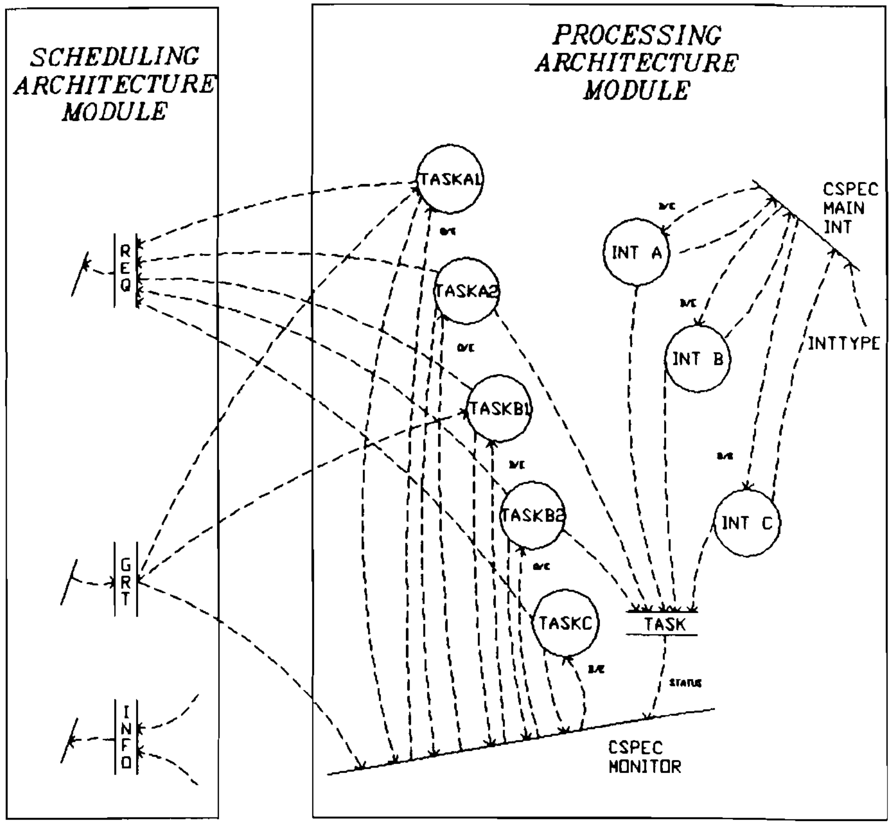
No difference exists between the control and data transformations implementations. At this point we don't know, if a task is implemented in the control unit as a finite state machine or a microprogrammed machine. The basic implementation architecture is shown in figure 74. Several techniques can be used to implement the control unit [Davio,1983] [Stevens,1987]. If we use a finite state machine implementation, all control and data tasks are combined into one big machine generating the control signals for the datapath.

figure 73: AFD of a PAM



AFD/AID SHOWING DATA FLOW

(DF = DATAFLOW)



AFD/AID SHOWING CONTROL FLOW

(D/E = DISABLE/ENABLE)

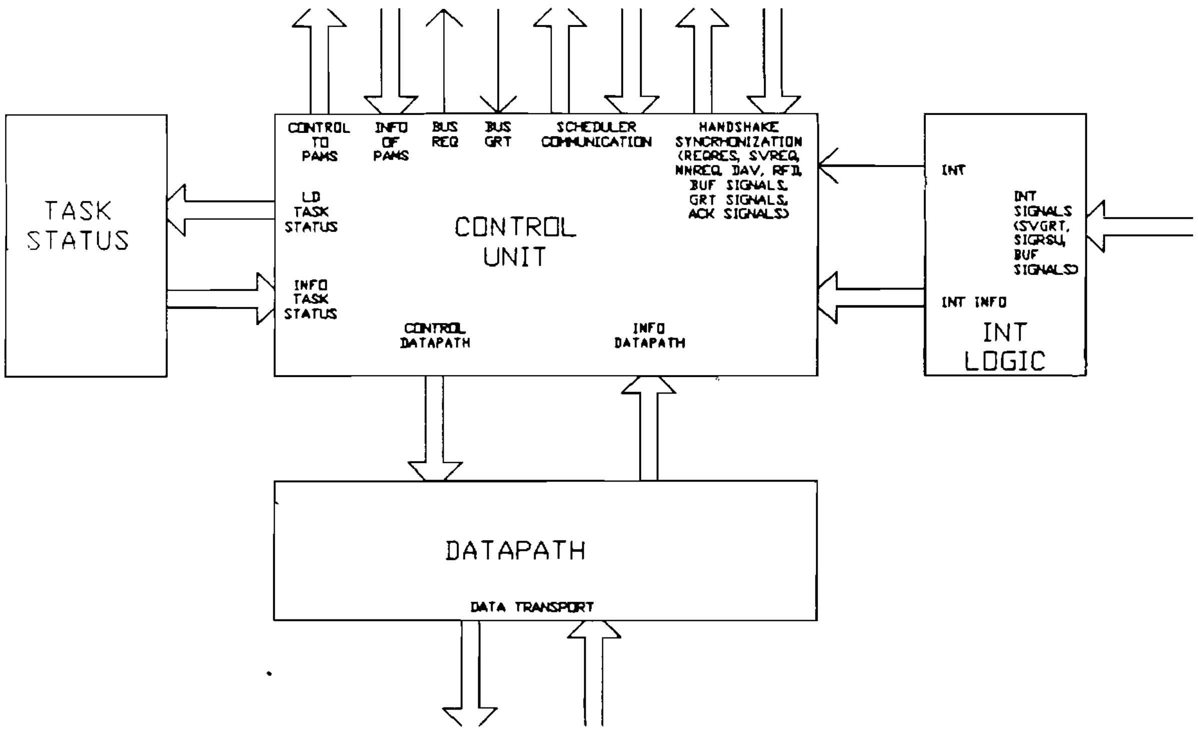
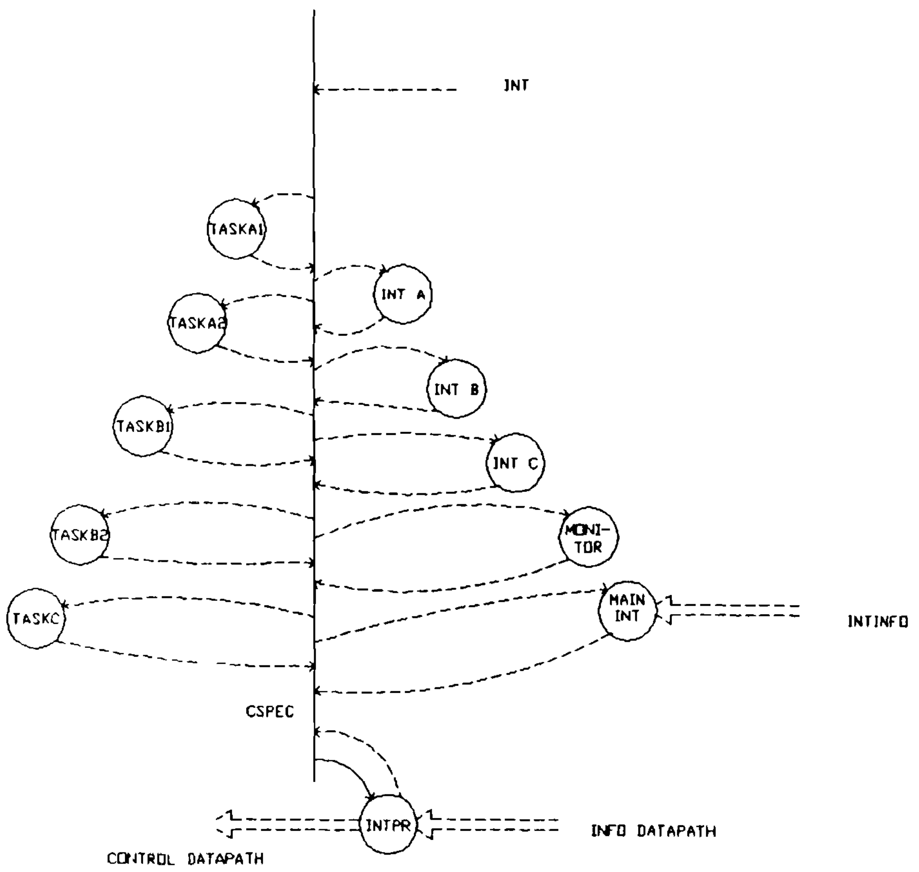


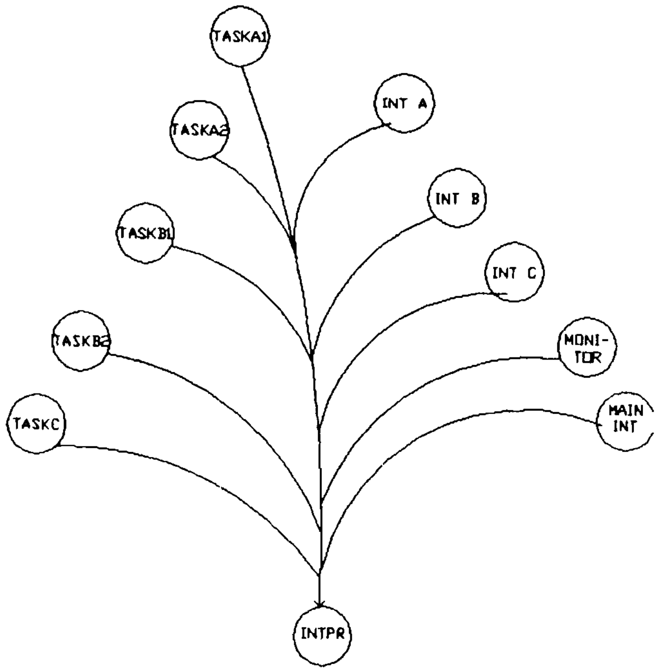
figure 74: basic implementation of the PAM

Figure 75 is the implementation of a control unit using a microprogrammed machine. The controller fetches a microcode and decodes the code to generate the control signals. The CSPEC is the datapath and the execution control. This transformation detects an interrupt and starts the main interrupt routine.

figure 75: microcode finite state machine model



CFD OF MICRO-
PROGRAMMED FINITE
STATE MACHINE



DFD (ONLY BETWEEN
MICRO CODE INTER
PROC AND OTHERS

11. CONCLUSION

The requirements and architecture modeling tool presented is capable of locating concurrent processes and describing synchronization and communication interaction for application specific VLSI design. Both models use leveling to hide unnecessary low level details and concentrate on concurrency. The method avoids process sequencing and an implementation bias.

The final requirements model is a detailed functional specification of the digital system and is the starting point for the architecture model, which is recursive and consists of parallel operating modules. Data transformations, data synchronization, mutual exclusion and scheduling control transformations are allocated to modules implementing data processing, shared processing, coprocessing and pipelining.

The architecture model consists of several modules (**PAM**, **SVAM**, **BAM**, **SAM**, **RMAM**). It is recursive for the **PAMs** introducing several data channels. In the implementation these logical channels can be mapped on one physical channel (one databus). The control modules may consist of other modules, but it introduces too much overhead and delay implementing only basic techniques.

In the easiest allocation one module implements one transformation usually wasting resources. The allocation should fulfill the requirements, optimize silicon and the number of the resources. Concurrent processes can be sequenced as long as the (especially timing) requirements are met. A chosen allocation changes the requirements model, which may change the allocation again. The development process is an iterative loop.

In processing architecture modules, a dispatching process tries to fulfill the timing requirements, when several tasks are allocated. To avoid busy-waiting for an event, a process is divided into tasks, which don't contain any busy-waiting primitives except for resource mutual exclusion. An interrupt routine handles the event occurrence without causing a context switch or even changing the datapath contents and makes runnable a task waiting for that event. The interrupted task is finished and then the dispatcher is started.

The allocation of shared processes introduces special synchronization primitives and a server architecture module. Both partners (requester and executer) must communicate parameters and results and synchronize actions on a direct connection, which is set up by the SVAM after a processing architecture module requests the execution of a shared process. Buffers storing parameters and results and generating synchronization signals are another solution for shared processing increasing flexibility. The actual implementation and techniques depend on the requirements model. To describe the interaction we use logical modules (**CPAM** and **SPAM**), which don't differ in the implementation. Multiplexing synchronization lines can save wiring and a clever arbiter allocation saves resources.

The defined simple primitives describing synchronization between modules and tasks, mutual exclusion, scheduling and data exchange require a handshake protocol, since the communicating parties operate in parallel and can be unable to detect an event during a period. To store the information and to take care of the asynchrounous events the primitives implementation uses flipflops.

All synchronization and communication situations can be described preventing starvation and introducing fairness with software techniques using the defined process synchronization, request/acknowledge synchronization, mutual exclusion and data communication primitives. Requiring synchronization primitives, data is exchanged on private channels or shared channels using a resource manager.

The software synchronization and mutual exclusion techniques are complicated, because the number and character of processes and event are unknown. In application specific design this information is available, so every situation can be implemented. Software techniques are implemented after simplification. The corresponding control transformations are allocated to separate modules (resource manager, bus arbiter, scheduling) containing parallel executing units; one for every independent identifier of a resource/event or a group of resources/events. Some control transformations are sequenced using a centralized implementation to avoid too much overhead and delay. At a certain level parallelism doesn't improve the performance, but even degenerates it.

The synchronization using software techniques should prevent deadlock and starvation and implement serialization. Starvation of requests is avoided by implementing a rotating priority scheme. Deadlock must be solved in the design program text. Critical sections are synchronized with software methods using the hardware primitives. A control transformation selects the module to execute its critical section.

Using application knowledge simplifies the synchronization schemes and implementations. The difference between resource management and scheduling/synchronization for a problem can than be very small. A wrong synchronization technique may cause too much delay, but still several techniques produces suitable solutions and good implementations depending on the application design.

Finally, I recommend to use the tool for specifying a few example applications, to actually implement them, to study the consequences of the allocation decisions especially for the control transformations, to measure the performance and to determine the idling periods of modules. The model rules and checking should be formalized to automate generation of model parts and to check consistency. The **PSPEC** can be simulated to check the functional descriptions, leveling and timing requirements. Unfortunately a large part of the design process (parallel detection) can't yet be captured in a silicon compiler at this stage (e.g. introducing concurrent transformations using an interpretation of **PSPECs**).

GLOSSARY

ACD; **architecture context diagram** gives an overview of how the systems physically fits in its environment. The elements of the ACD are: one architecture module, representing the system; terminators that represents entities in the environment with which the system communicates, and information flow vectors that represent the communications that take place between the external entities in the environment and helps identifying their type.

ACK(IDENT); Acknowledge; R/A synchronization primitive. The module sends an acknowledge to the requesting partner to signal that the expected action identified by IDENT has happened.

WACK(IDENT); Wait for acknowledge; R/A synchronization primitive. The requesting partner waits for an acknowledge generated by the primitive ACK.

ACKRSU(IDCPAM); Acknowledge of buffer. A communication protocol primitive between the BUFRSU and the CPAM granting access to the result buffer.

AFD; **architecture flow diagram** depicts the system architecture. It shows the physical partitioning of the system into its component pieces or modules and the information flow between them. Its main purpose is to allocate the functional processes of the requirements model to physical units of the system and to add more processes as needed to support the new physical interfaces. An architecture flow diagram is a network representation of a system's physical configuration.

AID; **architecure interconnections diagram** shows the actual physical information channels between the AFDs architecture modules and to and from the environment.

AISs; **architecture interconnect specifications** support the AID by textual specifications that specify the characteristics of the communication channels.

AM; **architecture module.** A physical entity that either is a grouping of other physical entities or is a fundamental physical entity to which logical flows and processes have been allocated; could be a hardware or a software unit. A hardware unit operates concurrently with others.

AMSs; **architecture module specifications** define the inputs, outputs and processes allocated from the requirements model for each architecture module.

Architecture model is derived from requirements model by applying design criteria that map functional requirements into an architecture showing physical entities, defining information flow between them.

architecture dictionary captures the allocation of data and control flows to specific interconnect channels (flow name, compose of, origin, destination, channel). The architecture dictionary contains the data and control flow definitions of the requirements dictionary plus the allocation of these flows to architecture models.

BAM; bus arbiter module; module of architecture model. CSPEC implementation for data bus access.

BUFCMD; buffer containing the requested process identity and parameters for shared processes execution. The logical unit BUFCMD contains also the data transport executing processes.

BUFRSU; buffer containing the results of the executed shared processes. The logical unit BUFRSU executes the data transport.

CFDs; control flow diagrams map control flows along the same paths as the data flow may travel. CFDs mirror the processes and the stores of the DFDs, but do not show data flows

CONID; connection identity needed for connection set up between requester and shared process executer for direct connection synchronization.

CPAM calling process architecture modules; logical module identifying a physical PAM containing a task calling a shared process allocated in a logical module SPAM.

CSPECs; control specifications specify control processing controls for the processes on the DFD.

Data condition; Control flow flowing from the PSPEC to a CSPEC.

DAV; data available; information flow/signal used for data transport between concurrent modules.

DFDs; data flow diagrams decompose the system and its functions showing transformations and data flows between those transformations.

GRTRSU(IDSPAM); Grant access for buffer BUFRSU. A communication protocol primitive between the SPAM and BUFRSU for getting access to the buffer BUFRSU granting access. IDSPAM identifies the SPAM getting access.

MNACK(IDSPAM); Monitor acknowledge. Shared process synchronization primitive. IDSPAM identifies the shared process module. Primitive (CF) for signaling the monitor that a shared process can be started.

MNREQ(IDSPAM) monitor request. Monitor request. Shared process primitive (CF) of PAM requesting a new shared process to execute.

monitor; (hardware) a control process of the processing module testing all grant signals for allocated processes and selecting a process. (software) synchronization mechanism based on abstract data types.

PAM; processing architecture module; am with allocated data transformations.

Process; a process is the transformation of incoming data flow into outgoing data flow.

PSPECs; process specifications specify the system's functional requirement. Control flows generated inside PSPECs through test on data flows are called data conditions. They flow to CFDs (control flow diagram), where they are treated as any other control flow.

RELRES(ID); Release resource. Primitive which releases the locked resource or group of resources identified by ID.

REMACKRSU(IDCPAM); Remove acknowledge for buffer access. A communication protocol primitive between the BUFRSU and the CPAM removing the access grant.

REMGRTRSU(IDSPAM); Remove grant for buffer access. A communication protocol primitive between the SPAM and BUFRSU for getting access to the buffer BUFRSU removing the grant.

REMMNACK(IDSPAM); Remove monitor acknowledge. Shared process synchronization primitive removing the monitor acknowledge. IDSPAM identifies the shared process module.

REMMNREQ(IDSPAM); Remove monitor request. Shared process synchronization primitive removing the monitor request. IDSPAM identifies the shared process module.

REMREQRSU(IDSPAM); Remove request for access to buffer BUFRSU. A communication protocol primitive between the SPAM and BUFRSU for getting access to the buffer BUFRSU removing the request.

REMSIGRSU(IDCPAM); Remove signal result ready and access for buffer. A communication protocol primitive between the BUFRSU and the CPAM removing the access request.

REMTASK; Primitive for task synchronization in a sequential PAM executed by a running task to reset the runnable flag.

REQ(IDENT); Request. Primitive for R/A synchronization sending a request to the partner requesting execution of an action identified by IDENT.

requirements model describes what the system must do (essential activities) and what data it must store so that the description is true regardless of the technology used to implement the system.

requirements dictionary; It contains an alphabetical listing of all the data and control flows in the DFDs and CDFs along with their definitions.

response time specification defines the limits on response time allowed between events at the system input terminals and the resulting events at the system output terminals.

RFD; ready for data; Information flow/signal used for data communication between concurrent modules.

RMAM; resource manager module. An AM implementing the resource management CSPECs regulating access to resources, while implementing identifiers relations and types of request.

SAM, scheduling architecture module. AM implementing CPSEC for process condition checking after receiving a request of a PAM.

shared process; primitive process appearing several times on a DFD and implemented only a few times in the architecture module.

SIGPROC(IDENT); Signal process. Primitive for process synchronization signalling an event (CF) to the SAM making a part of a condition identified by IDENT true.

SIGRSU(IDCPAM); Signal result ready. A communication protocol primitive between the BUFRSU and the CPAM for the result signalling.

SIGTASK; Task synchronization primitive in a sequential PAM signalling that a task waiting for an event is now runnable.

SPAM; **shared processing architecture modules**; logical module identifying a PAM containing only shared processes requested by a CPAM.

SPB; **shared processing block** A logical module containing buffers and SPAMs.

SPID; executable shared process identity.

SVAM; **server architecture module.** Architecture module implementing a server necessary for shared process execution. The unit is responsible for caller and executer selection and the execution start.

SVGRT(IDCPAM); Service grant. Primitive generating a service grant for a calling PAM; IDCPAM identifies the caller.

SVREQ(IDCPAM); Service request. Primitive requesting a service (execution of a shared process). ICPAM identifies the requester.

Task. A set of instructions manipulated (stopped, interrupted and resumed) as an unit by the hardware and software implementation of synchronization primitives.

WAITPROC(IDENT). Process synchronization primitive. The process waits until it receives a SAM signal to continue, which depends on the condition where to the identifier refers.

WAITRES(ID); Wait for access resource granted. Primitive requesting access to a resource or a group of resources identified by ID processed by the RMAM.

WGRTRSU(IDSPAM); Wait for grant access to buffer BUFRSU. A communication protocol primitive between the SPAM and BUFRSU for getting access to the buffer BUFRSU; It first generates a request and than waits for access grant.

WMNACK(IDSPAM); Wait for monitor request. Shared process synchronization primitive waiting for monitor acknowledge. IDSPAM identifies the shared process module.

WMNREQ(IDSPAM); Wait for monitor request. Shared process synchronization primitive waiting for monitor request. IDSPAM identifies the partner.

WREMGRTRSU(IDSPAM); Wait for removal grant access to buffer. A communication protocol primitive between the SPAM and BUFRSU for getting access to the buffer BUFRSU waiting for the removal of the grant.

WREMMNREQ(IDSPAM); Wait for removal of monitor request. Shared process synchronization primitive waiting for removal of monitor request. IDSPAM identifies the shared process module.

WREMSIGRSU(IDCPAM); Wait for removal of signal result ready and access to buffer. A communication protocol primitive between the BUFRSU and the CPAM waiting for the removal of the request.

WREMSVREQ(IDCPAM); Wait for removal service request. Shared process synchronization primitive waits for acknowledge (wait remove service request).

WREQ(IDENT): Wait for request. R/A synchronization primitive. The partner waits for a request of an action identified by IDENT.

WSIGRSU(IDCPAM); Wait for signal result ready and access to buffer allowed. A communication protocol primitive between the BUFRSU and the CPAM waiting for an access request.

WSVGRT(IDCPAM); Wait for service grant. Shared process synchronization primitive. (wait service grant) executed by the caller.

A. SYNCHRONIZATION AT THE LOWEST ALLOCATION LEVEL: CONSUMER PRODUCER PROBLEM

1. introduction

After allocating all transformations to modules the implementation phase starts. At this level several problems occur especially when modules containing one transformation access shared resources. We study the consumer producer problem and using software techniques we try to solve the problems [Dykstra,1982], [Raynal,1986] [Andre, 1985]. Discussing these solutions we find criteria to choose implementations or even to change allocations.

2. problem definition

We have two transformations at level n: a consumer and a producer process. The processes share a resource (a buffer). The process READ, a transforation of the consumer fetches an item out of the buffer and the process WRITE, a transformation of the producer stores an element in that buffer. The number of buffer locations is limited. The processes are executed in parallel and independent of each other.

We introduce a pointer PNTR that points to the buffer location where the next element can be read. Also, we introduce a pointer PNTW pointing to the buffer location where the next element can be written. After reset the pointers have an identical value indicating that an element should be put in the buffer. When the pointers are identical again, the buffer is empty or full. If READ was the last active process, the buffer is empty. Otherwise it is full.

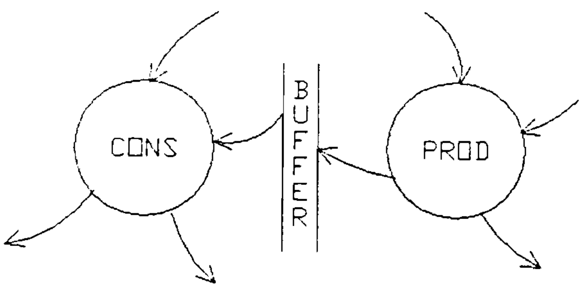


figure 76: requirements model

The READ transformation can ask for an element, when the buffer is empty. The transformation waits until the producer has put a new element in the buffer.

The process execution is shown in figure 77 (timing chart). RREQ: read request, RACK: acknowledge of the read, WREQ: write request, WACK: acknowledge of the write action. We call this semi-parallel execution, because the processes access the memory after each other (mutual exclusion). Accessing the buffer by the two processes at the same instant, the execution is parallel (process synchronization). We concentrate on the subprocesses studying the interaction and introducing several solutions.

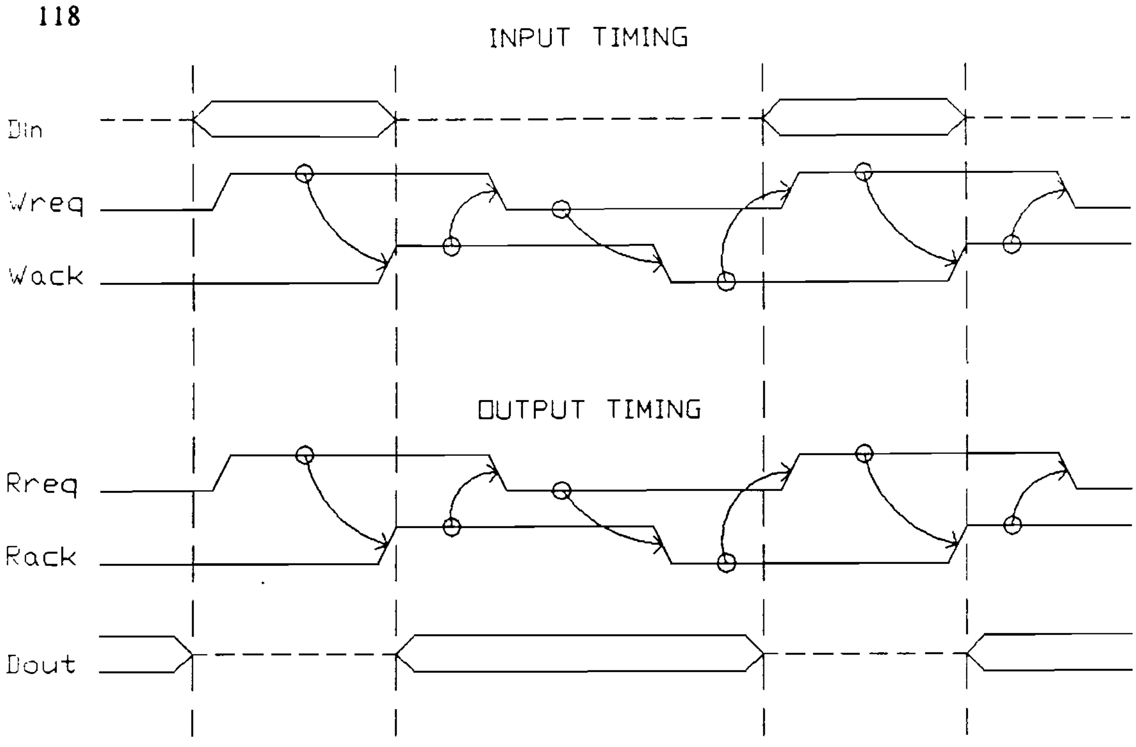


figure 77: semi-parallel process execution

3. structural model design

A simple allocation stressing the existence of levels is shown in figure 78. In the first level (n) we find the processing modules (consumer and producer). At this level a resource and scheduling architecture module controls the module synchronization. In every processing module we see at level n+1 more processing architecture modules and again a resource and scheduling module.

A part of the consumer and producer process is allocated to a separate module, because of the many interactions and the sharing of the datapath (a buffer and some pointers). The shared resources are controlled by the **RMAM** or **SAM** at level n. The buffer module isn't sequential indicating that a better allocation to put the buffer and the controllers in separate module. But the buffer can't operate on his own. The datapaths of the processes are interconnected so tightly, that we have to allocate them to one module to get a comprehensive allocation. In the final implementation we get four modules (the datapath, both controllers and the synchronization module).

The module READ executes only one process, started after reset. If just one process asks for an element, we use R/A synchronization to start the READ process. Otherwise a server (arbiter) will be needed. This last allocation is more logical, since it makes no sense to allocate the READ process to a separate implementation module to fetch an element once in awhile.

A solution containing mistakes will be presented to show problems occurring when the synchronization isn't efficiently introduced.

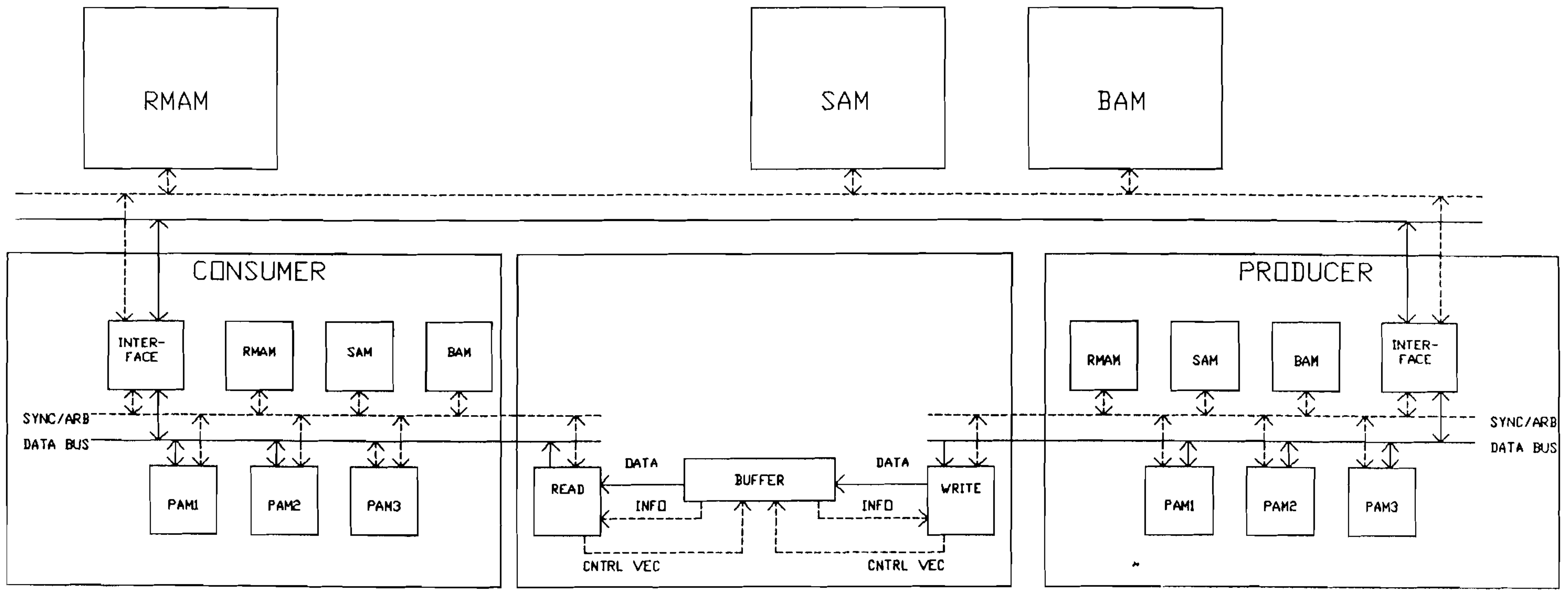


figure 78: allocation

3.1. THE STRAIGHT FORWARD SOLUTION

First we describe the process WRITE without synchronization except for the request synchronization. (the READ has the same structure.)

R/A SYNCHRONIZATION IDENTIFIERS

IWR	= identifier for the process write
IRD	= identifier for the process read

LOCAL VARIABLES

SUC	= boolean indicating if process action has succeeded
REG	= variable containing the data

SHARED VARIABLES

PNTW	= integer, value of memory location where the next element will be written
PNTR	= integer, value of memory location where the next element will be read
RPRIO	= boolean, indicating which process had last access to the memory (True indicates that the WRITE process was the last active process, RPRIO=read priority)

GLOBAL CONSTANT

SIZE	= the number of memory locations

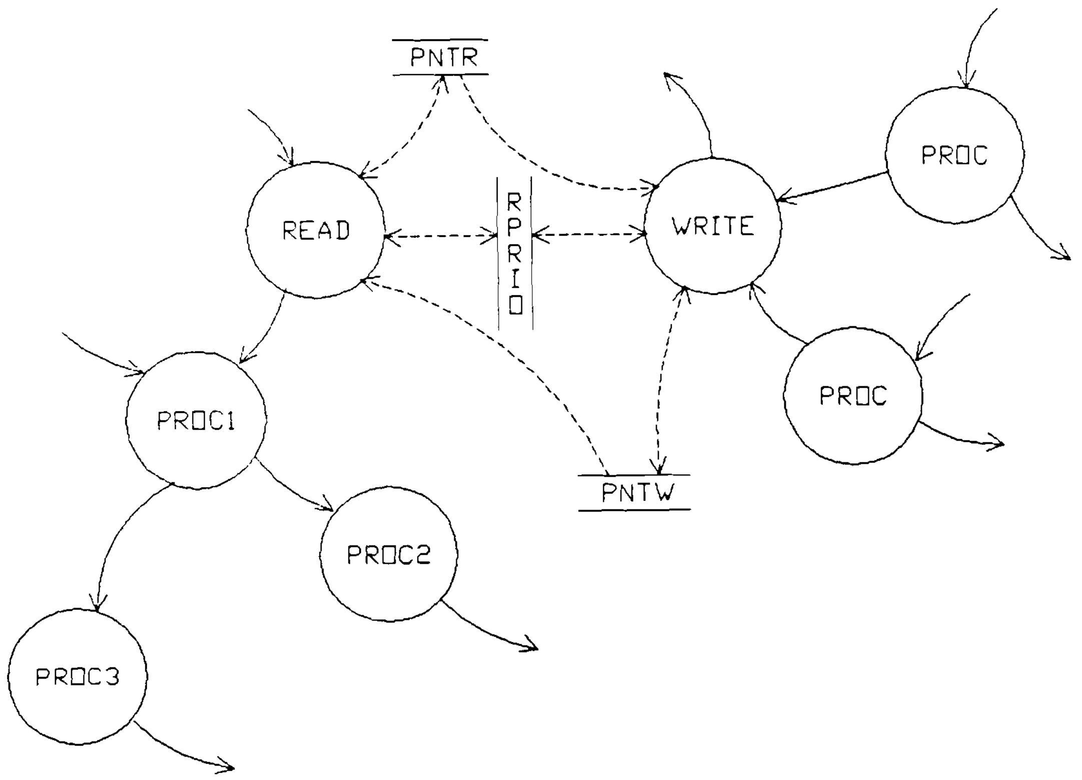


figure 79: requirements model

```
WRITE;
while true do
      begin
        wreq(iwr); (* request for write action *)
        suc:=false; (* no access succeeded *)
        while not(suc) do
            begin
             if ((pntw≠pntr) or not(rprio))
                   then
                    cobegin
                          rprio:=true;
                          suc:=true;
                          begin
                            buf[pntw]:=reg;
                            pntw:=(pntw+1) mod size;
                          end;
                    coend;
            end;
            od;
        ack(iwr); (* write succeeded *)
      end;
  od;
```

Now we introduce synchronization for the shared resources. Every shared resource gets an identifier.

RESOURCE MANAGEMENT IDENTIFIERS:

IPNTR	= identifier for the pointer PNTR
IPNTW	= identifier for the pointer PNTW
IRPRIO	= identifier for the variable RPRIO
IMEM	= identifier for the buffer memory
IALL	= identifier for a group of resource (PNTW,PNTR,RPRIO)

WRITE PSPEC:

```
while true do
      begin
        wreq(iwr); (* request for write action *)
        suc:=false;
        while not(suc) do
            begin
             cobegin
                    waitres(ipntw);
                    waitres(irprio);
                    waitres(ipntr);(* request access to the pointer of the other process *)
             coend;
             if ((pntw≠pntr) or not(rprio))
                    then
                       cobegin
                          relres(pntr);
                          rprio:=true;
                          suc:=true;
                          relres(irprio);
                          begin
                                 waitres(imem);
                                 cobegin
                                        buf[pntw]:=reg;
                                        relres(imem);
                                 coend
                                 cobegin
                                        pntw:=(pntw+1) mod size;
                                        relres(ipntw);
                                 coend;
                          end
                       coend
                    else
                       cobegin
                          relres(ipntw);
                          relres(irprio);
                          relres(ipntr);
                       coend;
            end;
            od;
        ack(iwr);
      end;
  od;
```

READ PSPEC:

```
while true do
     begin
       wreq(ird); (* request for read action *)
       suc:=false;
       while not suc do
           begin
             cobegin
                 waitres(ipntw);(*request access to the pointer of the other process*)
                 waitres(ipntr);
                 waitres(irprio);
             coend;
             if ((pntr≠pntw) or (rprio))
                 then
                   cobegin
                       relres(ipntw); (* release resource *)
                       rprio:=false;
                       suc:=true;
                       relres(iprio);
                       begin
                         waitres(imem);
                         cobegin
                             reg:=buf[pntr];
                             relres(imem);
                         coend;
                         cobegin
                             pntr:=(pntr+1) mod size;
                             relres(ipntr);
                         coend;
                       end
                   coend;
                 else
                   cobegin
                       relres(ipntw);
                       relres(ipntr);
                       relres(iprio);
                   coend;
           end;
           od;
       ack(ird);
     end;
  od;
```

A request for the buffer memory isn't necessary, because the other process can't be active. The active process uses RPRIO, so the other process has to wait until RPRIO is released. We will differentiate between the read and write access to allow more processes to read one variable.

A deadlock problem could be introduced, when several resources are requested one after another and the other process needs the same resources. A process has already a few variables and is waiting for some others, which have been granted to another process waiting for this process to release its variables. The design should avoid this. Therefore the resource identifier IALL is introduced representing a cluster of resources (RPRIO, PNTR and PNTW). This identifier isn't independent of other identifiers and this relation has to be implemented in the resource manager module.

WRITE PSPEC:
```
while true do
    begin
      wreq(iwr);
      suc:=false;
      while not(suc) do
          begin
            wrdres(iall); (* request to read all shared var*)
            if ((pntw≠pntr) or not(rprio))
                then
                  begin
                  (* when allowed to write the read process can't change the condition *)
                  cobegin
                    rlrdres(iall);
                    waitres(imem); (* wait until read proc is finished *)
                  coend;
                  cobegin
                    buf[pntw]:=reg;
                    relres(imem);
                  coend;
                  cobegin
                    begin
                        wwrres(ipntw);
                        cobegin
                              pntw:=(pntw+1) mod size;
                              rlwrres(ipntw);
                        coend;
                    end;
                    begin
                        wwrres(irprio);
                        cobegin
                              pprio:=true;
                              rlwrres(irprio);
                        coend;
                    end;
                    suc:=true;
                  coend
                else rlrdres(iall);
                fi
          end;
      od
      ack(iwr);
    end;
  od;
```

This implementation requires protection of the buffer.

READ PSPEC:

```
while true do
   begin
     wreq(ird);
     suc:=false;
     while not(suc) do
       begin
         wrdres(iall); (* request to read all shared var*)
         if ((pntw≠pntr) or rprio)
              then
                begin
                  (* when allowed to read the write  process can't change the condi-
                  tion *)
                  cobegin
                        rlrdres(iall);
                        waitres(imem); (* wait until write proc is finished *)
                  coend
                  cobegin
                        reg:=buf[pntr];
                        relres(imem);
                  coend;
                  cobegin
                        begin
                          wwrres(ipntr);
                          cobegin
                                pntr:=(pntr+1) mod size;
                                rlwrres(ipntr);
                          coend;
                        end;
                        begin
                          wwrres(irprio);
                          cobegin
                            rprio:=false;
                            rlwrres(irprio);
                          coend;
                        end;
                        suc:=true;
                  coend;
                end;
              else rlrdres(iall);
              fi
       end;
     od
     ack(ird);
   end;
 od;
```

Both processes have a long critical section excluding each other's activity. A solution is to copy critical variables. The copy is only used during the manipulation of the original by the other process. This seems possible in the programming text, but we shouldn't forget that for example PNTR and RPRIO are changed together. To use the pointer's copy effective, RPRIO should be copied also. At least two clock cycles (see appendix B) are needed (claim, copy and release) to make these copies. The other process uses at least two clock cycles (claim and copy and release) to increment PNTR. These two clock cycles could be won implementing a copy, but to produce the copies two extra cycles are needed.

Figure 80 contains the AFD/AID/IMPLEMENTATION schematic of the modules and the necessary communication and synchronization signals. This schematic is a result of the program description and the implementation of the communication and synchronization primitives.

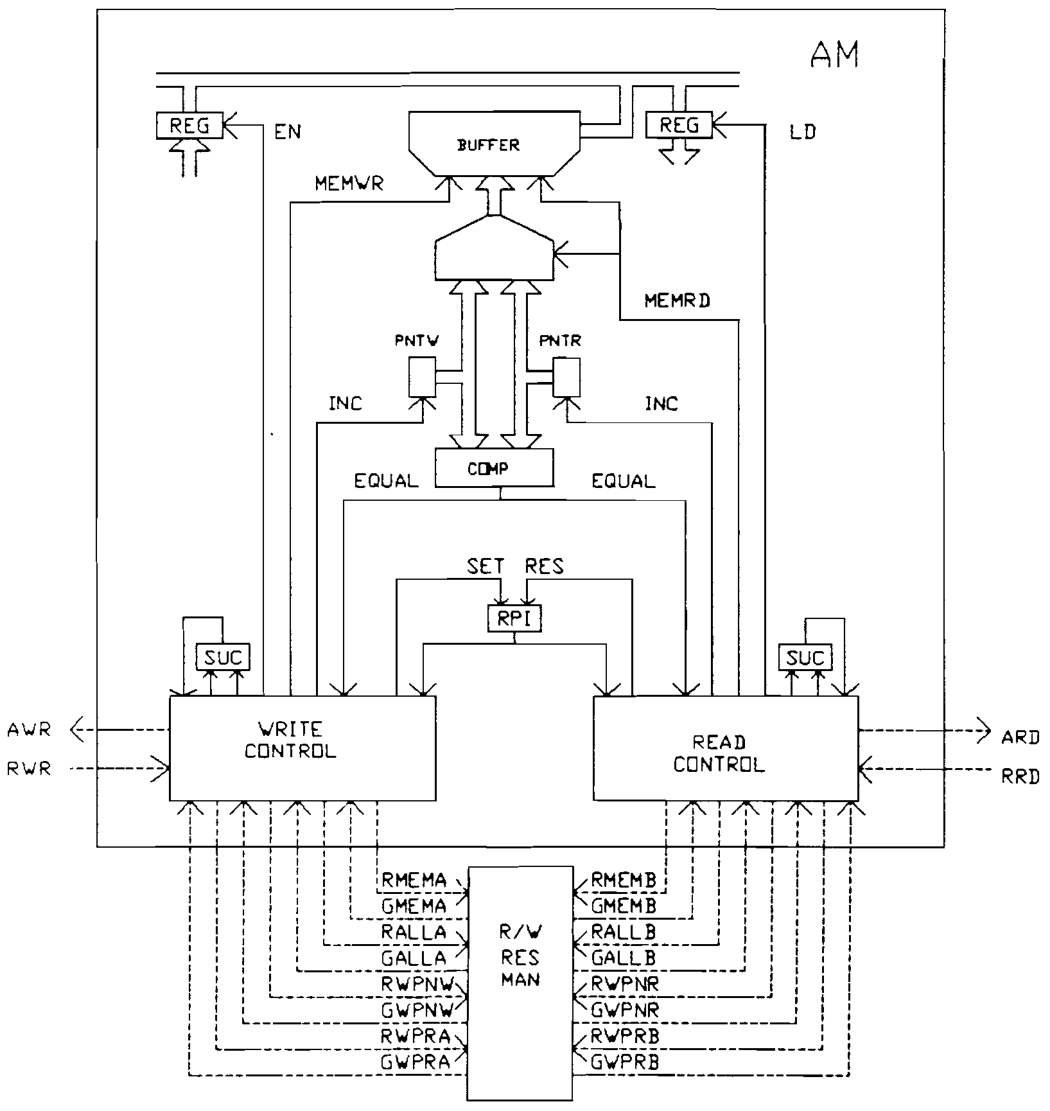


figure 80: implementation

The following identifiers are used:

RWR	: request write
AWR	: acknowledge write
RRD	: request read
ARD	: acknowledge read
RMEMA/B	: request of partner A/B for memory
GMEMA/B	: grant for partner A/B for memory
RALLA/B	: request of partner A/B for all resource
GALLA/B	: grant for partner A/B for alle resource
RWPNW	: request to write in pointer PNTW
GWPNW	: grant to write in pointer PNTW
RWPNR	: request to read in pointer PNTR
GWPNR	: grant to read in pointer PNTR
RWPRA/B	: request of partner A/B to write priority FF
GWPRA/B	: grant for partner A/B to write priority FF

The resource manager implements the difference between a read and write request and will grant several read requests when possible.

This simple program requires a lot of synchronization statements. So we start searching for a solution that doesn't use so much synchronization. (see simulation figure 86)

In the parallel implementation the buffer can be accessed by the two processes as long they don't access the same location. This won't happen, because the problem definition doesn't allow it. When the pointers have the same value, the buffer is either empty or full. When the buffer is full the WRITE process has to wait and when the buffer is empty the READ process has to wait. The WAITRES(IMEM) and RELRES(IMEM) statements are redundant.

3.2. INTEGER SEMAPHORE SOLUTION

Now we will present another solution using integers semaphores controlling the resource buffer and at the identical moment the process execution. The process synchronization doesn't allow access to the same buffer location. So the buffer doesn't need any protection. A counter CRD contains the number of locations, where an element can be read and another counter CWR contains the number of locations, where an element can be written. We introduce process execution conditions. The process WRITE can be executed, if CWR≠0. Then the counter CWR should be decreased and the counter CRD should be increased. So one element more can be read. The process READ can be executed if CRD≠0. Then the counter CRD should be decreased and the counter CWR should be increased.

```
IDENTIFIERS
     ICRD  : identifier for read semaphore
     ICWR  : identifier for write semaphore

WRITE PSPEC:
begin
     while true do
       begin
          wreq(iwr);
          waitproc(icwr);
          cobegin
               sigproc(icrd);
               begin
                  buf[pntw]:=reg;
                  cobegin
                     pntw:=(pntw+1) mod size;
                     ack(iwr);
                  coend;
               end;
          coend;
        end;
       od;
end;

READ PSPEC:
begin
     while true do
       begin
          wreq(ird);
          waitproc(icrd);
          cobegin
               sigproc(icwr);
               begin
                  reg:=buf[pntr];
                  cobegin
                     pntr:=(pntr+1) mod size;
                     ack(ird);
                  coend;
               end;
           coend;
        end;
       od;
end;
```

The corresponding counters should be initialized to CWR=SIZE OF THE BUFFER and CRD=0.

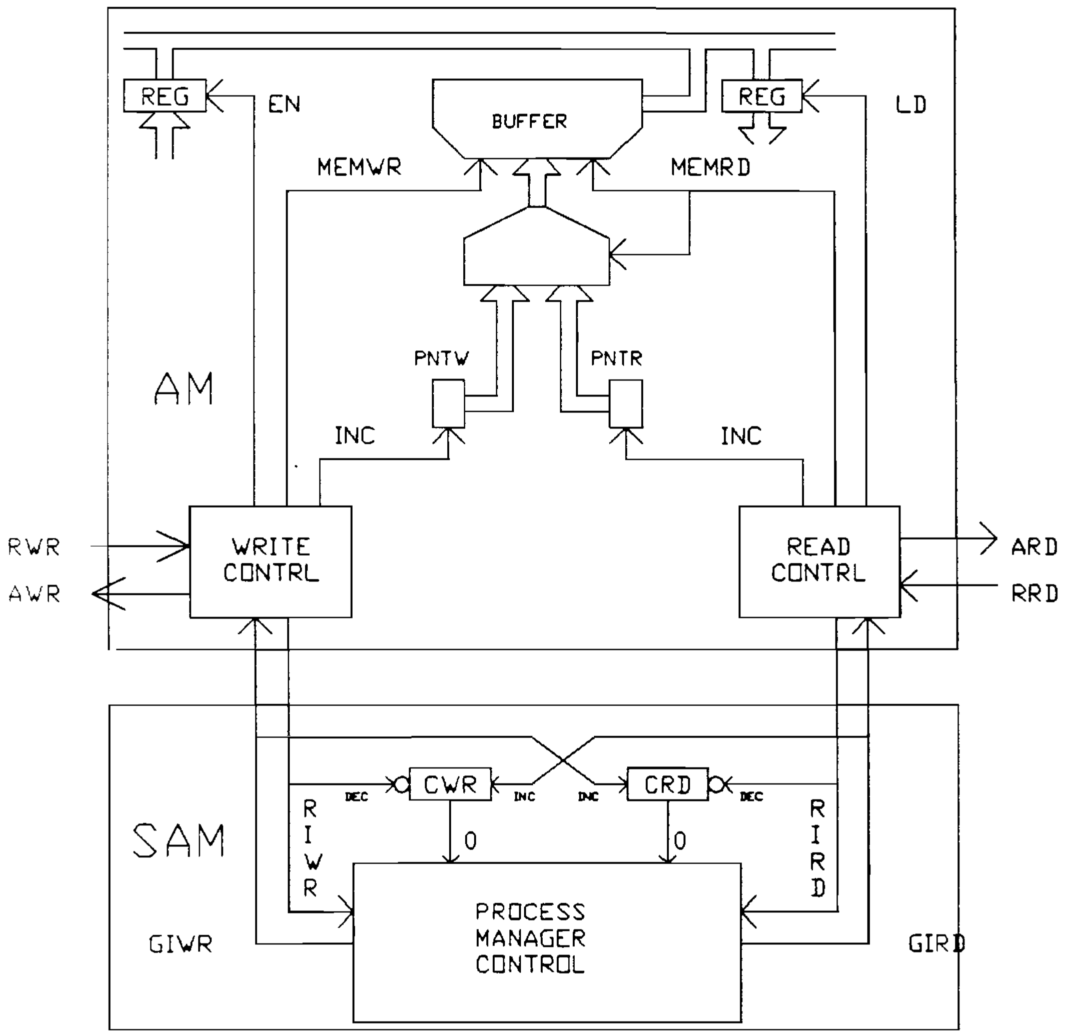


figure 81: AFD/AID/IMPLEMENTATION

The semi-parallel schematic can be found in figure 81 using a R for request and a G for grant.

3.2.1. process management

The process synchronization implementation depends on the concurrent buffer access. The process management synchronization regulates the semi-parallel memory access. Accessing the buffer together the process management module (SAM) implementation is difficult. A counter can be incremented and decreased at one instant requiring a resource manager.

The semi-parallel solution is very simple to implement. The schematic is presented in figure 82. The counter is incremented, when an acknowledge is given. The logic decides which process is active.

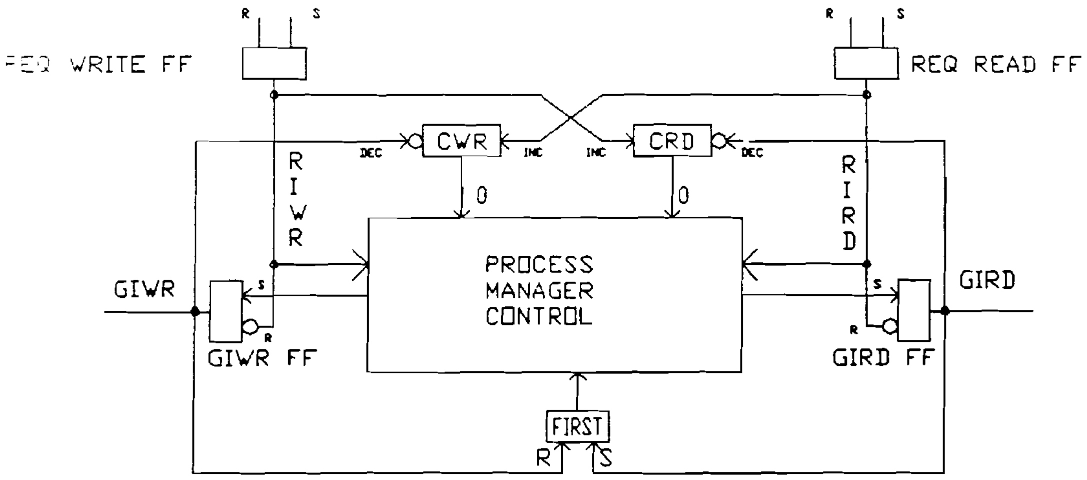


figure 82: implementation of SAM for semi-parallel access

The logic functions:
S_{GIWR}:= NOT(ZEROCWR) AND (RIWR AND (RIRD OR FIRST))
S_{GIRD}:= NOT(ZEROCRD) AND (RIRD AND (RIWR OR NOT(FIRST)))
R_{GIWR}:= NOT(RIWR)
R_{GIRD}:= NOT(RIRD)
S_{FIRST}:= GIRD
R_{FIRST}:= GIWR

In the parallel situation we have to introduce resource management. As the increment can't make the zero line true, the increment can be executed in parallel with a test on the zero line. The zero line won't be tested, when the counter is decreased.

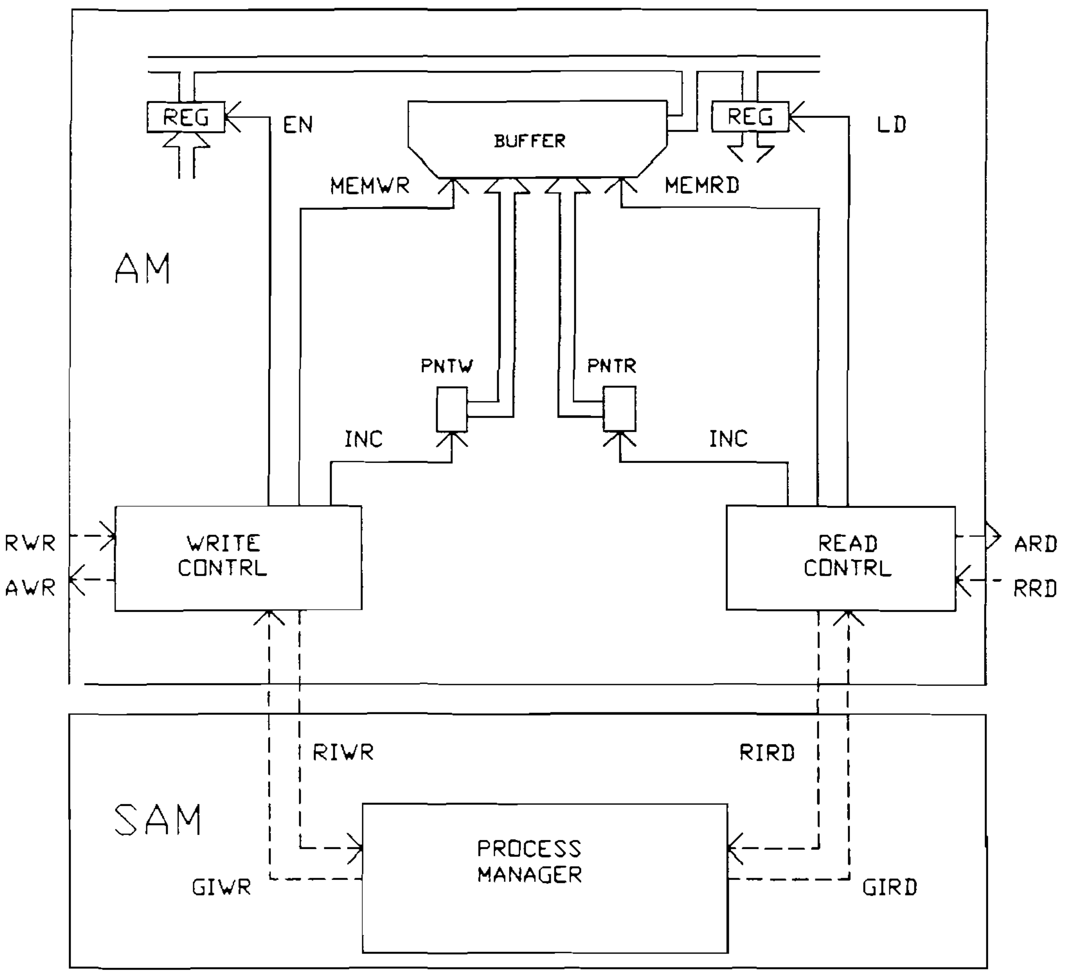


figure 83: parallel buffer access with process management

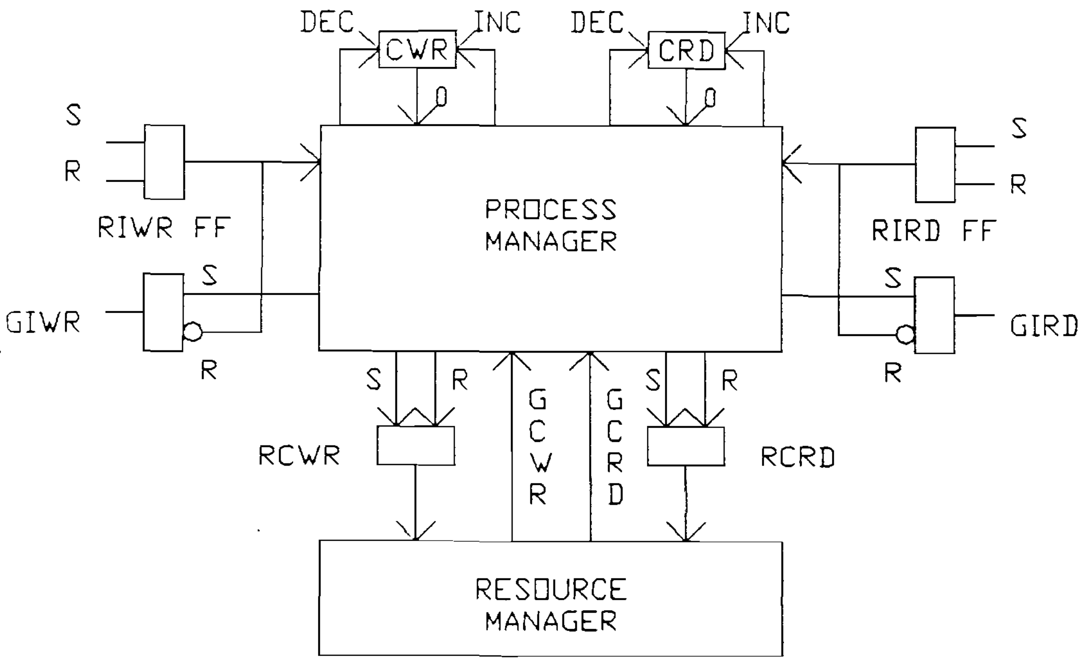


figure 84: implementation of SAM

mutual exclusion identifiers:

```
ICRD  : counter read
ICWR  : counter write
```

PROCESS MANAGER PSPEC:

```
cobegin
  while true do
     begin
        repeat skip until riwr; (*wait until request to execute process write *)
        repeat skip until not(cwr=0); (* check condition *)
        cobegin
           giwr:=true; (* grant proc write *)
           begin
              waitres(icrd);
              cobegin
                 crd:=crd+1; (* update read counter *)
                 relres(icrd);
              coend
           end
           begin
              waitres(icwr);
              cobegin
                 cwr:=cwr-1;
                 relres(icwr);
              coend;
           end;
         coend;
        end;
     od;
  while true do
     begin
         repeat skip until rird;
         repeat skip until not(crd=0);
         cobegin
           gird:=true;
           begin
              waitres(icwr);
              cobegin
                 cwr:=cwr+1;
                 relres(icwr);
              coend
           end
           begin
              waitres(icrd);
              cobegin
                   crd:=crd-1;
                   relres(icrd);
              coend;
           end;
         coend;
     end;
     od;
coend;
```

The process manager executes two processes concurrently. The resource management prohibits the parallel access of the buffer.

3.3. critical section synchronization

The analysis of the solution A.3.1, which uses read and write claims for the resource management shows that quite a lot of time is wasted by requesting access. The memory access shifts the parallel requests in time, so that PNTR and RPRIO also are accessed together not requiring different resource identifiers.

In our model every process state has outputs controlling the datapath. Introducing synchronization means that new states are introduced. Before we synchronize a problem, we fist minimize the description containing unsynchronized problem uses only three states.

The minimized program READ is:

```
READ;
begin
  while true do
    begin
      wreq(ird);(* request a new process*)
      while not(pntw≠pntr or rprio) do skip; (***)
      cobegin
        reg:=buf[pntr];
        ack(ird);(* this is allowed, because only in the next clockcycle a new request
            can be assigned *)
        rprio:=false;
      coend;
      pntr:=(pntr+1) mod size;
    end;
end;
```

The critical sections contain statements executed in two clockcycles. We have seen that synchronizing statements of a small critical section introduces a large amount of overhead. So now we will synchronize the complete critical section. One identifier (ICS) is used for all shared resources.

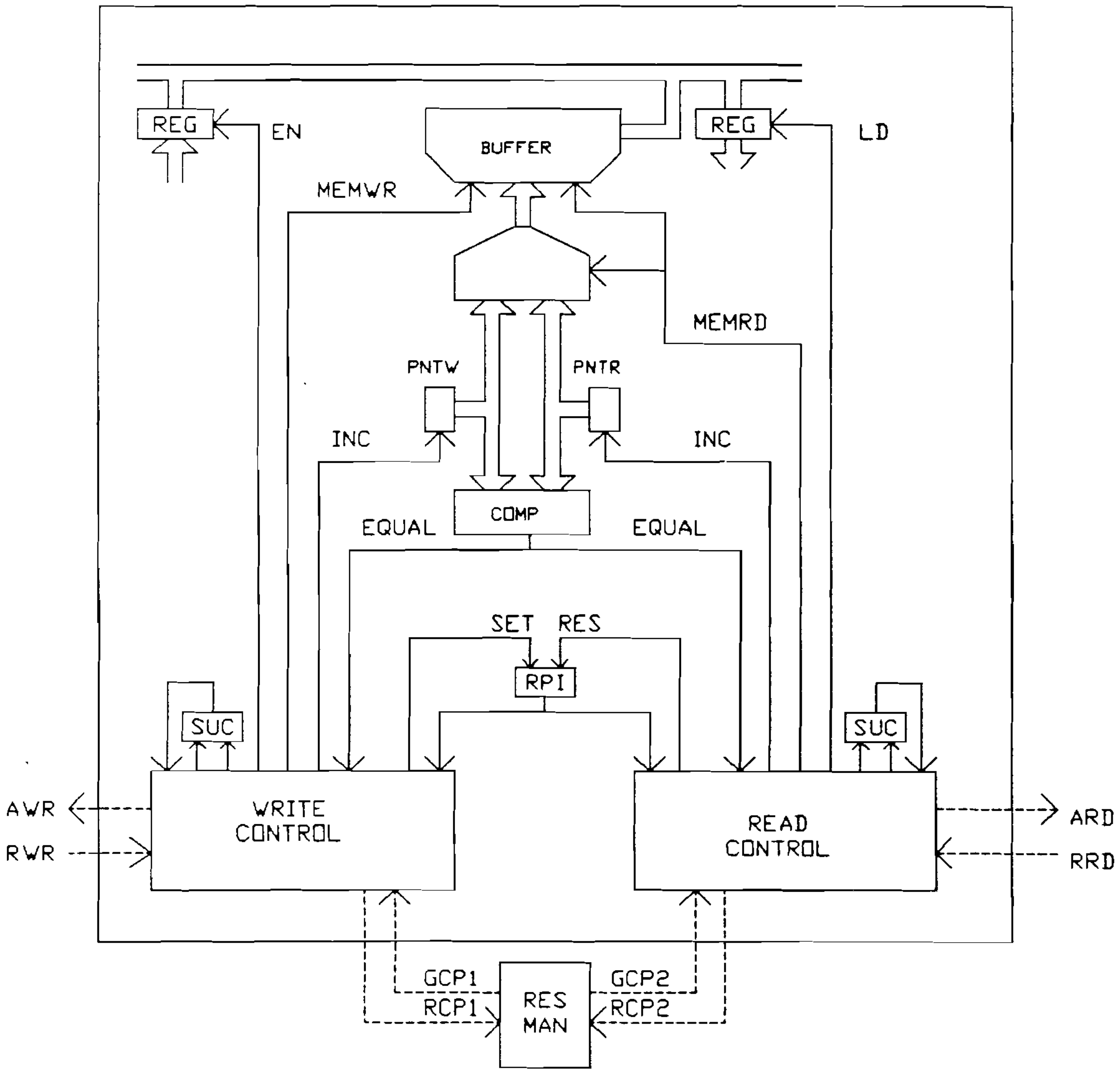


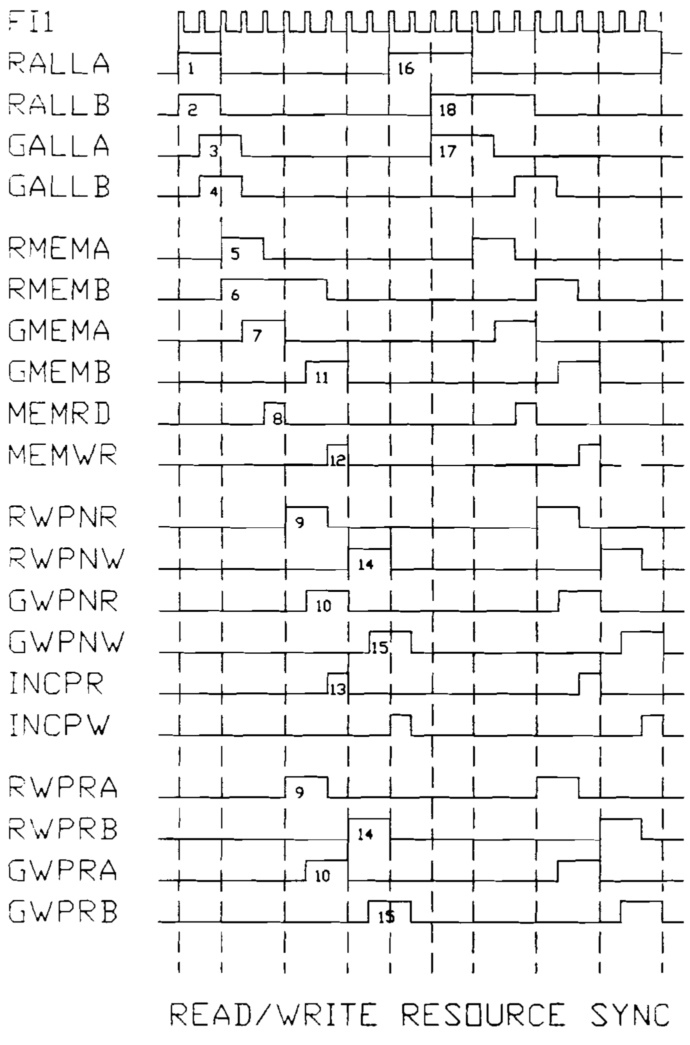
figure 85: critical section implementation

```
IDENTIFIERS:
        IRD   : READ process identifier
        ICS   : critical section of READ and WRITE
        IWR   : WRITE process identifier

READ;
begin
   while true do
      begin
         waitproc(ird);(* request a new process*)
         waitres(ics);(* request the resources *)
         while not(pntw≠pntr or rprio) do
            begin
               relres(ics);
               waitres(ics);(* give other process a chance *)
            end;
         od;
         cobegin
            reg:=buf[pntr];
            sigproc(ird);
            rprio:=false;
         coend;
         cobegin
            pntr:=(pntr+1) mod size;
            relres(ics);
         coend;
        end;
     od;
end;

WRITE;
begin
   while true do
      begin
         waitproc(iwr);(* request a new process*)
         waitres(ics);(* request the resources *)
         while not(pntw≠pntr or rprio) do
            begin
               relres(ics);
               waitres(ics);(* give other process a chance *)
            end;
         od;
         cobegin
            buf[pntw]:=reg;
            sigproc(iwr);
            rprio:=true;
         coend;
         cobegin
            pntw:=(pntw+1) mod size;
            relres(ics);
         coend;
      end;
     od;
end;
```

Unfortunately, we must divide an one clock cycle statement (marked ***), otherwise we introduce deadlock. The resources are assigned, but the process is waiting until the resources contents is changed. This will never happen, since the resources are exclusive for this process. The other process can change the contents, but can't execute its CS, because this process is in its critical section.The solution using critical section is pictured in figure 85. GCP1: grant process 1, RCP1: request of process1.

3.4. program progress

Figure 86 contains the timing of the straight forward solution and the critical section solution.



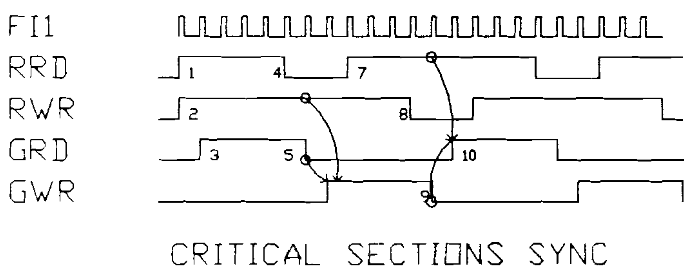


figure 86: timing charts

Explanation of the timing chart.

The controller of the resource manager knows that it is executing a read/write resource management scheme. The writing of PNTR and PNTW are independent. The requests to read all resources are implemented in the priority scheme as one request. After a grant, the read request is assigned the lowest priority.

Read/write synchronization.

1/2	request of read (A) and write (B) process to read all variables
3/4	both processes are granted the resource
5/6	both processes request memory access
7	process A is granted access to the memory
8	the memory action
9	process A requests new actions (to write pointer and pri)
10/11	process READ is granted new actions and process WRITE is granted the memory access
12/13	memory action of process WRITE and update pointer
14	request to update pointer of WRITE and pri
15	request granted
16	new request of A to read all variables
17	granted
18	new request of B

Critical section synchronization

1/2	request for access to critical section
3	cs granted for process Read
4	process READ is ready
5	the grant for process READ is removed
6	process WRITE is granted
7	new request for cs of process Read
8	process WRITE is ready
9	critical section is released
10	process READ is allowed to execute the critical section.

No solution is really faster. But the logic to implement R/W synchronization is enormous compared to the critical section implementation. The logic for the R/W synchronization is tricky, because the relations between the several identifiers have to be implemented correctly and starvation should be avoided. So using process knowledge in the description phase can be advantageous (introducing critical sections).

4. van de weij solution

A solution is presented in figure 87, which doesn't use the synchronization primitives. The description of the modules is as follows:

ACT	= boolean, indicating that an action has to be performed
ACK	= boolean, indicating that the action is completed
REQ	= boolean, indicating that the action is requested
CONT	= boolean, indicating that the action has succeeded

HS MODULE (HS= HANDSHAKE)

```
while true do
    begin
        cobegin
            act:=false;
            ack:=false;
        coend;
        repeat skip until req; (* wait until request *)
        act:=true;
        repeat skip until cont; (* wait until action succeeded *)
        cobegin
            act:=false;
            ack:=true;
        coend;
        repeat skip until not(req); (*wait until request is withdrawn *)
      end;
    od;
```

This can be implemented as a state machine.
The description of the logic in van de Weij solution is:

```
PROC LOGIC;
cobegin
        while true do
                cobegin
                 if (wr and (not(rprio) or not( eq or rd)))
                        then wsucc:=true
                        else wsucc:=false;
                 if (rd and (rprio or not( eq or wr)))
                        then rsucc:=true
                        else rsucc:=false;
                coend
        od;
        while true do
                cobegin
                   if wsucc
                        then
                          begin
                             cobegin
                                rprio:=true;
                                ram[pntw]:=in;
                             coend;
                          pntw:= (pntw+1) mod size;
                          end;
                    if rsucc
                        then
                          begin
                             cobegin
                                rprio:=false;
                                out:=ram[pntr];
                             coend;
                             pntr:= (pntr+1) mod size;
                          end;
                coend;
        od;
coend;
```

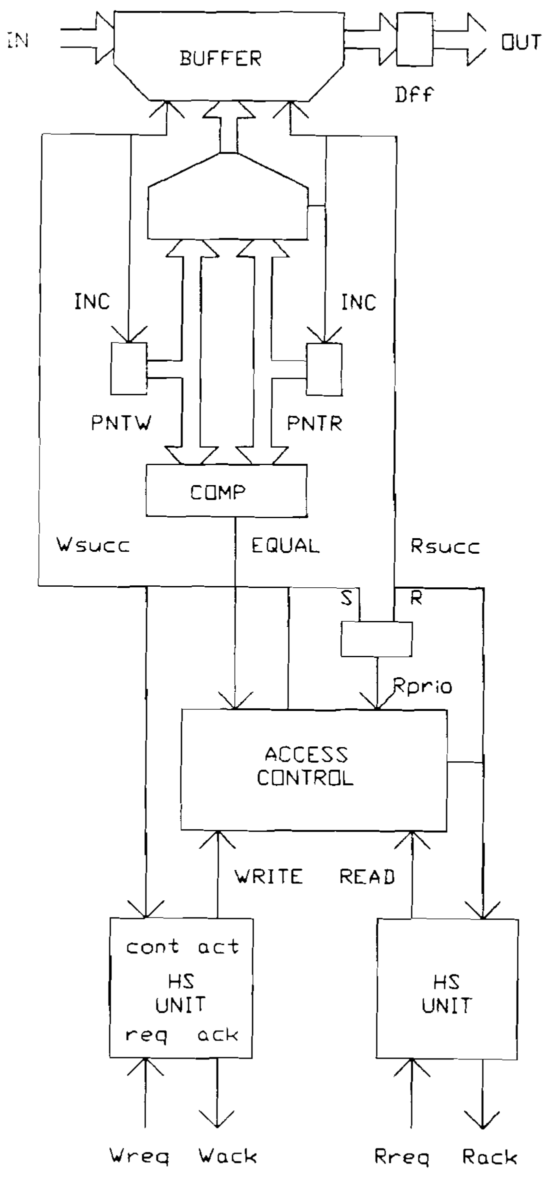


figure 87: van de weij solution

The block ACCESS CONTROL is just logic. The logic functions are:
Wsucc= WRITE AND ((NOT(RPRIO) OR NOT(EQUAL OR READ))
Rsucc= READ AND ((RPRIO OR NOT(EQUAL OR WRITE))

The ASM chart in figure 88 shows the difference between the solution of critical section synchronization and the van de Weij solution (the numbers indicate corresponding actions). We can't concluded from this figure which is the best solution. The van de Weij solution is probably the fastest.

In the normal operation the execution of a process using an arbiter takes at least six clockcycles. The reason for the long execution time is, that the parallel processes need communication. Communication requires a request and an acknowledge, which takes two clock cycles each.

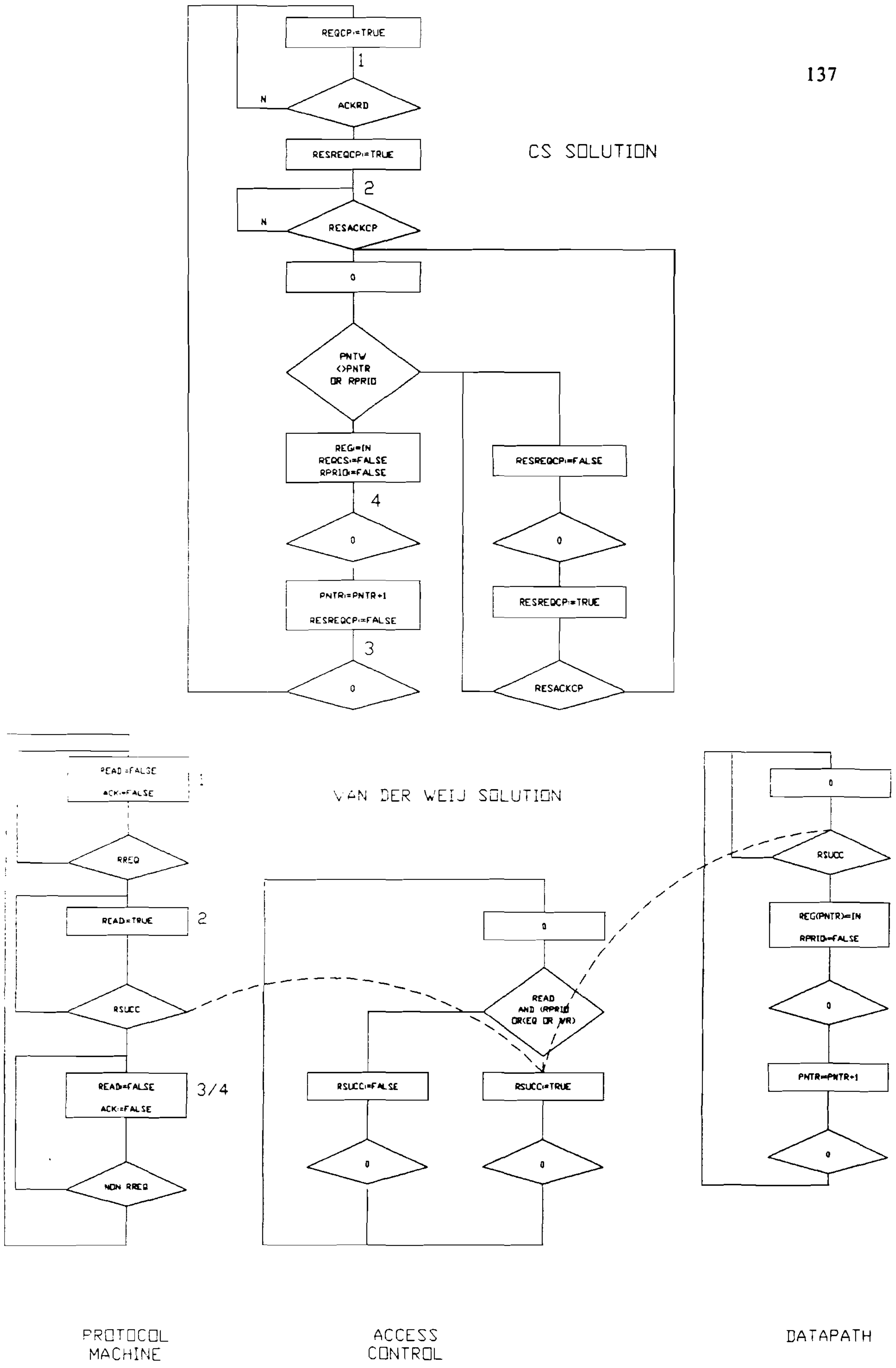


figure 88: ASM charts

5. comparison of the solutions

It is difficult to compare the several solutions, since no information is available about the timing. We have seen two groups of solutions: semi and real parallel. The semi-parallel solutions are the Critical Section solution, the resource management solution, the process manager solution and the van de Weij solution. In each solution the arbiter (RMAM/SAM) has knowledge about the processes. Parallel solutions are the process and the resource manager implementation. The best solution depends on the design goals. If the manager may only be an arbiter, then only cs and resource management can be used. (process knowledge isn't required)

In the final implementation phase we have knowledge about which resources will be shared, what kind of modules are allocated and what the processing conditions are. We have several tools to synchronize the processes and processing modules. The boundaries between several solutions aren't clear. It depends on the responsibility given to a module and if copies of some vital information are kept. A trade-off appears between parallelism and synchronization overhead. At some point the increase of parallelism causes only more overhead and more delay.

We have seen that we should cluster resources, which are used together. This causes application knowledge to be implemented in the resource manager. It is also quite useful to cluster resource that are requested at short intervals after each other. We can use critical section synchronization to reduce overhead.

The solution of van de Weij is projected on our model in figure 89. He has integrated several modules by using process knowledge. It is a very clever solution, that can be used for buffer design, since buffers are needed frequently in parallel processing to speed up the processing. Our solutions use a structural design method with pre-defined building blocks (arbiters and manager).

The van de Weij solution can be modeled as follows: the access control executes several functions: server, processing condition control. Two processing modules (a producer and consumer) request a service (read or write). The server starts one of the processes after arbitration. The processing module containing the buffer requests the SAM to check if the processing conditions are satisfied. If no grant is given another condition will be checked. Only when a process is executed, the service acknowledge will be given.

A central module (access control) controls the process progress. In our solutions the resource management is executed separate, while the process and resource management in van de Weij are identical. Because of this separate management the priority is saved at two locations. Decentralization requires more hardware. The advantage of decentralization is that the manager doesn't need specific process knowledge allowing a straight forward implementation and allowing perhaps silicon compilation.

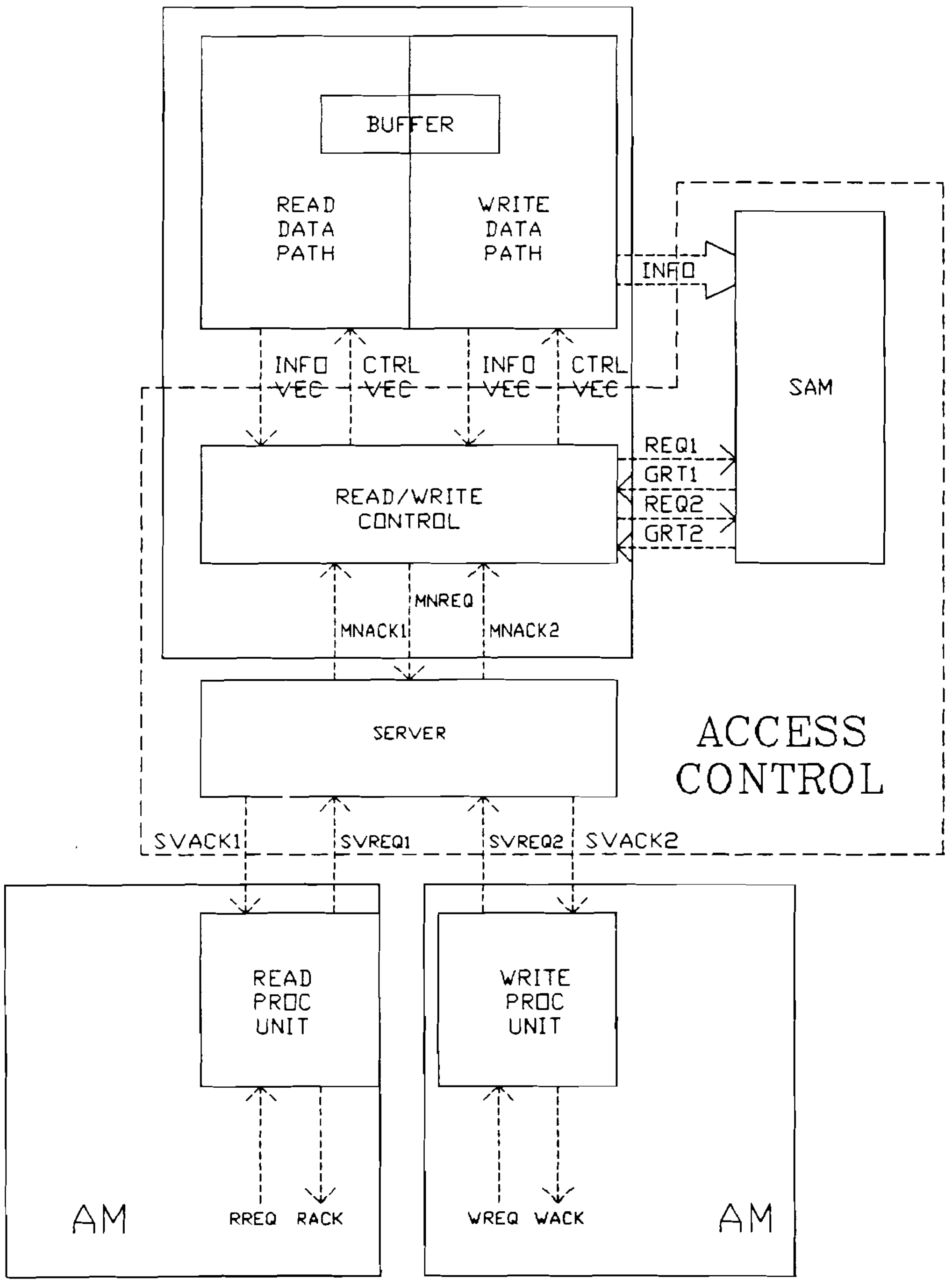


figure 89: modeling solution van de Weij

The presented interpretation using the building blocks of our model to represent the solution of van de Weij is a tricky and cumbrous one. It was only to show that the solution mixes several functions, which speeds up the processing, but is difficult to find using a structural functional approach.

The main conclusion is that for synchronization problems the access solutions don't differ from synchronization solutions. They can be implemented in the SAM as well as in the RMAM. We also saw that it is important to analyze the synchronization/mutual exclusion problems correctly. Very often a software synchronization technique a better result, then a straight forward solution using hardware primitives for every mutual exclusion access.

B. THE SYNCHRONIZATION MANAGEMENT IMPLEMENTATION

1. introduction

The first step is to optimize the process description with only the requirements synchronization. After analyzing the critical sections and the necessary mutual exclusion synchronization the synchronization statements are chosen, so that as few as possible restrictions are introduced. The different names for the several synchronization kinds make the program text readable and even easier to check.

All software techniques for mutual exclusion and synchronization are implementable in hardware, but the limited number of process and the dedicated process execution require only simple forms. In this appendix we translate the basic techniques into hardware. The application is discussed in the literature [Dykstra,1982], [Raynal,1986], [Andre,1985]. We should keep in mind, that a technique can be implemented in several ways, but one implementation can be faster than another.

We discuss only one identifier implementation. In the architecture module, more identifiers are implemented executing concurrent. The execution can be sequenced slowing down the reaction time and restricting the concurrent execution of processes (the module becomes a bottleneck in the processing). In appendix A, we concluded that the difference between SAM and RMAM is not great, especially when implementing process knowledge. We won't discuss this further, since it depends on process specification and relations between identifiers.

In our design we requires at least a two phase clock. After the generation and clocking of the control signals for the datapath, the result of an earlier action in the datapath or external info can be tested. This information is needed for the determination of the next state (action). Programming the synchronization we use information about the response time of the synchronization. The response on a request will arrive in the next clock period or later, so the removal of a request for a resource can be executed at the moment the final clock cycle access to the resource is started.

2. resource management architecture modules/arbiters.

We did define the following mutual exclusion primitives:

```
WAITRES(ID):        REQ:=TRUE;
                    REPEAT SKIP UNTIL GRT;
```

The request and the waiting for the grant are one statement, so that the resource won't be requested before it is really needed.

```
RELRES(ID):         REQ:=FALSE;
```

The implementation assures, that the GRT is removed before the request is raised.

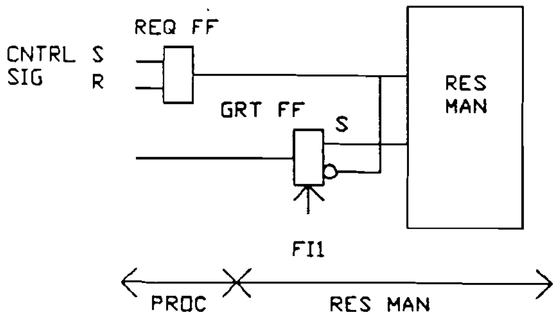


figure 90: implementation of the mutual exclusion synchronization

2.1. TWO REQUESTERS FOR ONE RESOURCE.

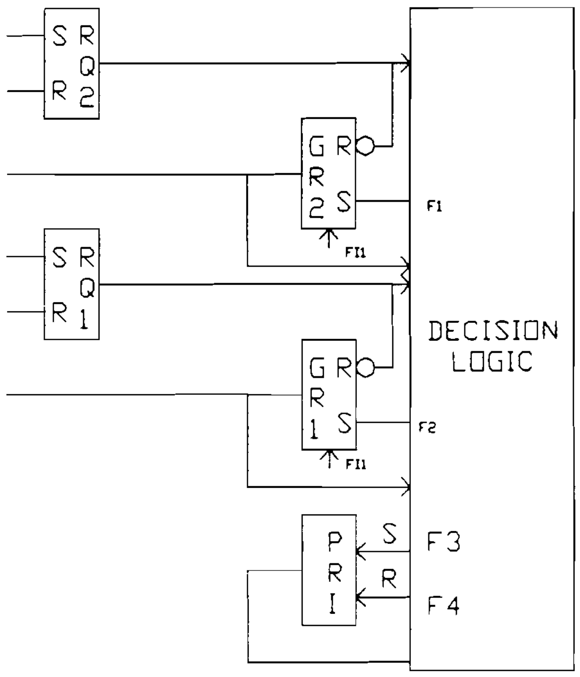
The situation where two requesters want access to one resource is encountered very often. We introduce a resource management logic functional description of a set of resources identified by an identifier IDRQ. Two transformations access these resources. One requester is identified by IDRQ1 and the other by IDRQ2.

IDENTIFIERS:

REQID1:	IDRQ 1 REQUEST
GRTID1:	GRANT OF THE IDRQ 1 REQUEST
REGID2:	IDRQ 2 REQUEST
GRTID2:	GRANT OF IDRQ 2 REQUEST
PRIOID:	BOOLEAN, INDICATING WHO HAS THE HIGHEST PRIORITY

```
PROC 2REQBINSEM(REQID1,REGID2,GRTID1,GRTID2,PRIOID);
cobegin
    while true do
        if (reqid1 and not(grtid2) and (prioid or not(reqid2))) then grtid1:=true;
        od;
    while true do
            if (reqid2 and not(grtid1) and (not(prioid) or not(reqid1))) then grtid2:=true;
        od;
    while true do if grtid2 then prioid:=true;
    while true do if grtid1 then prioid:=false;
    while true do
           begin
            while reqid1 do skip;
            grtid1:=false;
           end;
        od;
    while true do
           begin
            while reqid2 do skip;
            grtid2:=false;
           end;
        od;
coend;
```

This program contains six parallel processes, which are easily implemented in hardware.



TWO REQUEST RESOURCE MANAGER

$F1 := (RQ2 \text{ AND } \overline{GR1}) \text{ AND } (\overline{PRI} \text{ OR } \overline{RQ1})$

$F2 := (RQ1 \text{ AND } \overline{GR2}) \text{ AND } (PRI \text{ OR } \overline{RQ2})$

$F3 := GR2$
$F4 := GR1$

figure 91: 2REQBINSEM implementation

RQ1	**: RSFF CONTAINING REQID1**
RQ2	**: RSFF CONTAINING REQID2**
GR1	**: RSFF CONTAINING GRTID1**
GR2	**: RSFF CONTAINING GRTID2**
PRI	**: RSFF CONTAINING PRIOID, BOOLEAN INDICATING WHICH REQUEST HAS HIGHEST PRIORITY**

In this simple solution Reset Set flipflops are used. Because the GRT flipflops are clocked on FI1 (on FI2 the processing module preforms test on inputs and grants), the reset and set lines are never active together.In a faster solution this can happen, so JK flipflops should be used. If the set is still present during the arrival of the reset, the JK flipflop inverts its state. The GRT flipflop is clocked to ensure a stable test input.

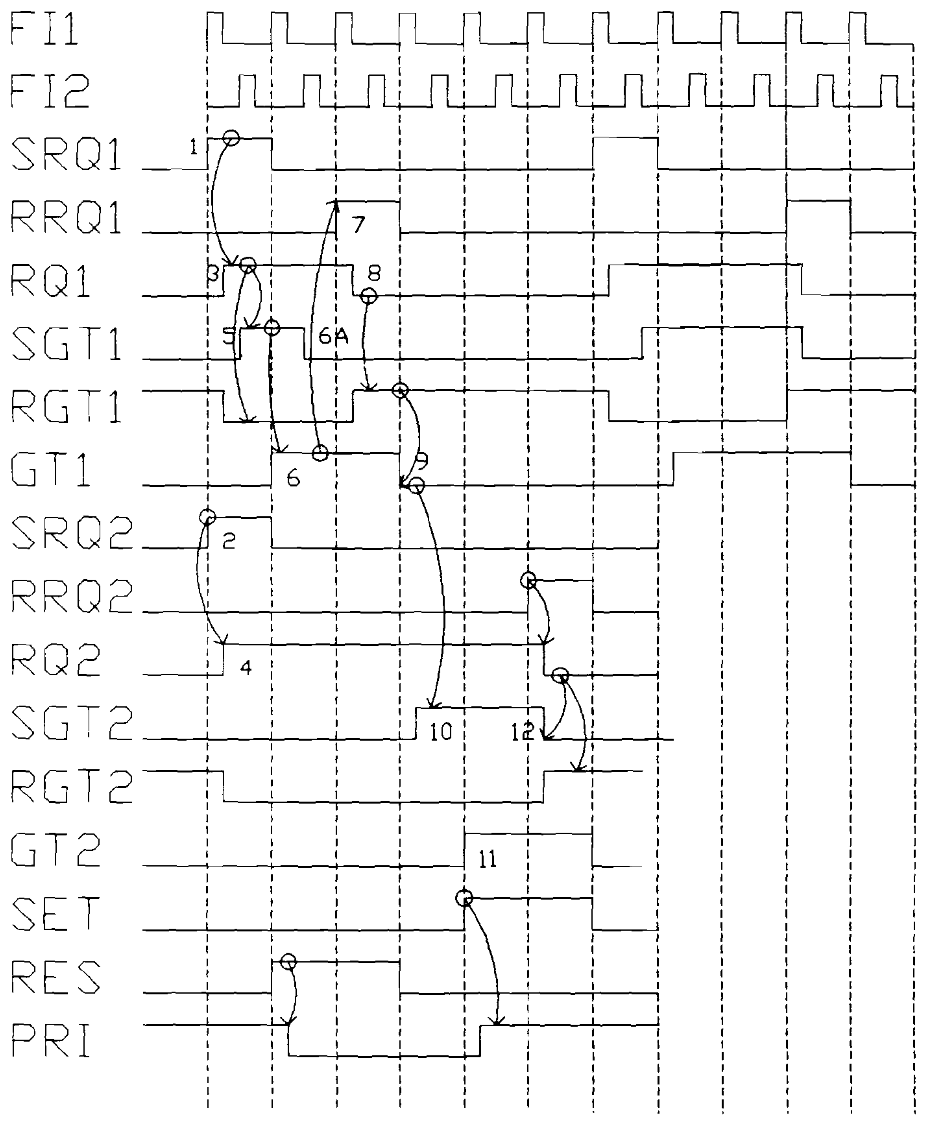


figure 92: timing of 2REQBINSEM

ACTION:

1.	set req 1
2.	set req2
3/4.	response on set req
5.	grant given to 1
6.	clocking grant
6a.	the set is not true anymore, because pri changed
7.	removal of request (release)
8.	req removed
9.	release
10.	set grt2
11.	grt2 clocked
12.	set grt2 removed.

The periods:

4-5	decision time
6-7	delay of resource
7-8	response time ff
8-9	clocking delay
10-11	clocking delay.

The waiting time of the processing module is at least two clock cycles (1,6). Serving two requests, the second request is served in four clockcycles and the number of clockcycles used by the other requester (2,11). The response on a release is given in one clockcycle. In a RMAM several 2REQBINSEM units can be implemented, operating parallel. If the response time is unimportant, a sequential implementation is possible. The module checks one after another the requests of every set of resource identifiers allocated to the RMAM.

2.2. SEVERAL REQUESTERS

The most encountered situation with several requesters is the access to a databus. Starvation should be avoid in granting requesters using a rotating priority scheme. Hardware generates a grant in one clockcycle. The result should be ready on FI2 (tests are executed in the PAM).

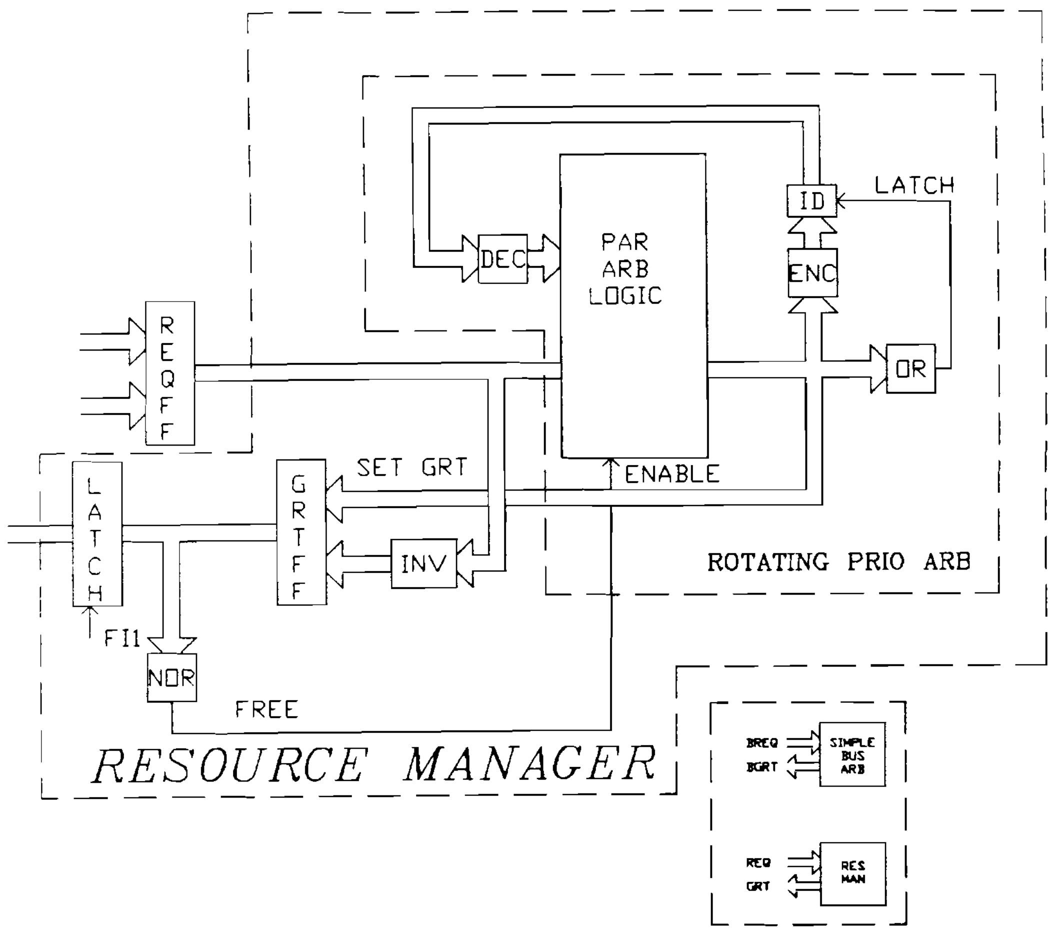


figure 93: implementation of simple RM or bus arbiter

A rotating priority arbiter is started by an enable line. Otherwise the result can change, when a new request arrives, the ID is changed and a new grant is set. To store the arbiter's result, the logic is disabled. During the arbiter response time no new request arrives, since the result is generated in one clockcycle. A resource manager can be used in **PSPEC** writing RESMAN(REQ,GRT). Every time a grant is generated its identity is latched to determine the new priority scheme.

The encode block (ENC) translates the outputs into a request identity, which will be translated into a new priority scheme by the decoder (DEC). The granted request will get the lowest priority.

```
FREE:=NOT( GRT1 OR GRT2 OR GRT3 ...)
LATCH:= SGRT1 OR SGRT2 OR SGRT3 ....
```

The ID is latched on the first edge of signal **latch**, which is delayed to give the encode logic enough time to generate the id. The **set grant** signal has to be active long enough to set the grant FF. The logic has to be disabled before the next FI1 on which a new request can arrive. The ID's latching and ID's decoding into a new priority scheme may not change the set input before the ff is set. The arbiter is at least during one clock period idle, because the resource has to be released.

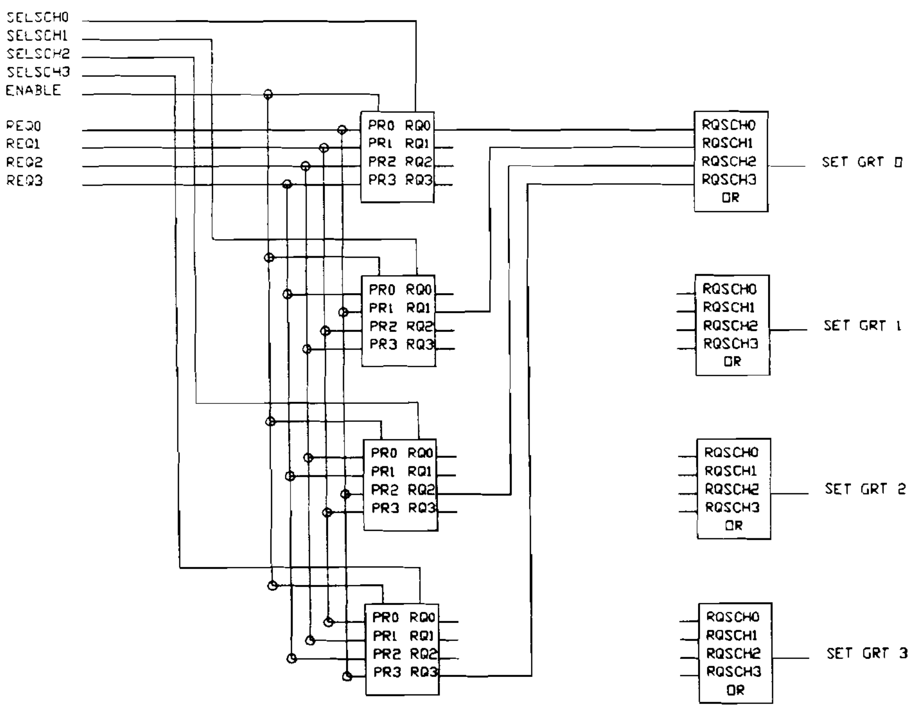


figure 94: pla implementation

The arbitration in one clockcycle can be implemented using two schemes. The first one uses building blocks containing a priority scheme is implemented. Depending on the last served request, a building block having the correct priority scheme is selected. The second scheme uses a daisy chain. The start number of this chain is selected by decoding the last served request. This solution uses less hardware, but the number of requests is limited, because of the ripple through time. (see figure 95)

A building block contains the following functions:
input requests: PR0, PR1, PR2, PR3 and the outputs: RQ0, RQ1, RQ2, RQ3.
RQ0:=(EN AND SELSCH) AND PR0
RQ1:=(EN AND SELSCH) AND NOT(PR0) AND PR1
RQ2:=(EN AND SELSCH) AND NOT(PR0) AND NOT(PR1) AND PR2
RQ3:=(EN AND SELSCH) AND NOT(PR0) AND NOT(PR1) AND NOT(PR2) AND PR3

Logic can be saved by implementing the enable in the decoder.

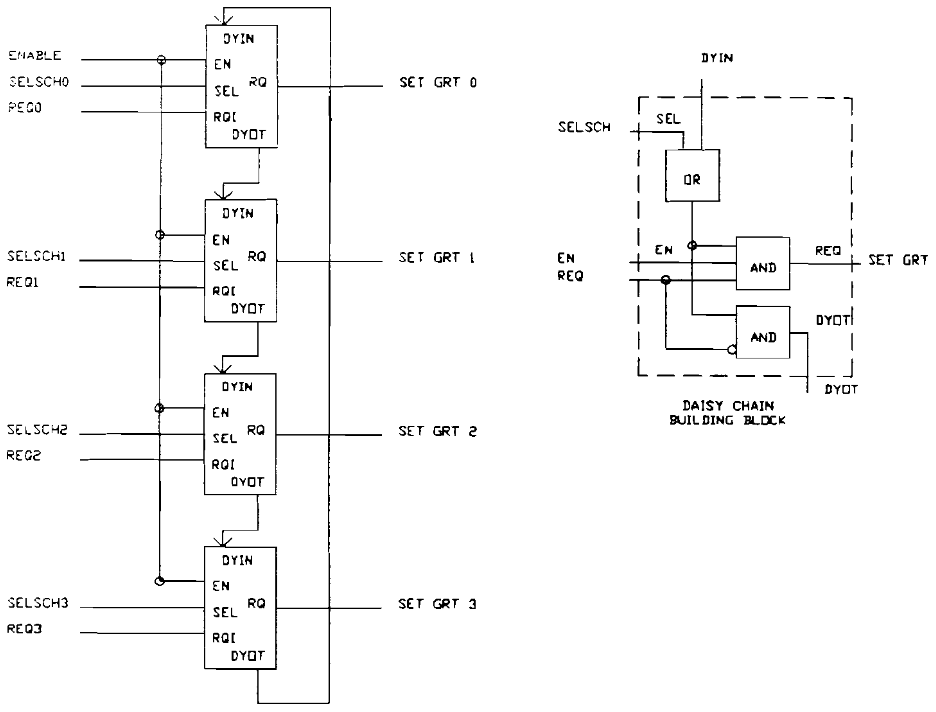


figure 95: daisy chain

The building block of the daisy chain has the following inputs and outputs:
DYIN := daisy in
DYOT := daisy out be given before the signal is reset.
DYOT := NOT(REQ) AND DYIN.

When no request is present, all DYOTS and DYINS are active. Several SET GRTS become active, when several requests arrive. The DYOT will go low, so the grant ff will have to be clocked on FI2. The enable can be moved to the decode logic.

Now we introduce a special arbiter called server arbiter defining a process SERVARB(REQ[1..M],STR,ID,RDY). Often a request is selected. Sometimes we only want the request id, which will be used by other modules. STR is the start/enable signal. The RDY signals the production of an ID, the identity of the process to serve. The RDY should be latched and be implemented with a handshake with STR. The result must be ready before FI2.

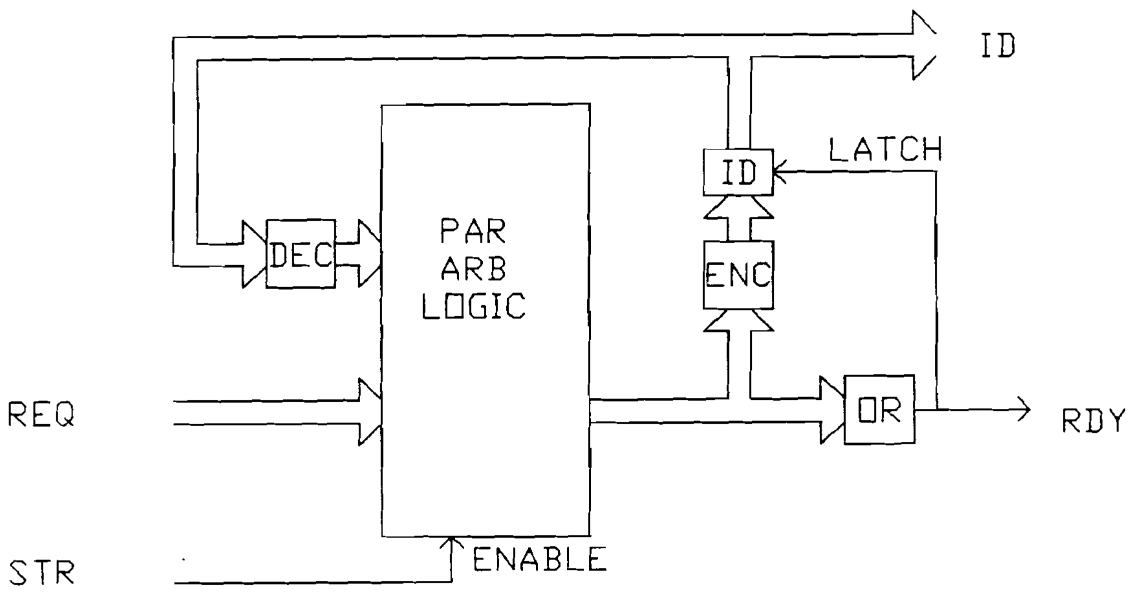


figure 96: server arbiter

2.3. THE SEQUENTIAL ARBITER

The next solution checks the requests one after another (fig 97a). This check is started, when the resource is free. After a request selection the grant will be set. When FREE is true again, the last served request will be checked last using a circular scheme. The request are checked, until one is found to serve.

IDRQ	: identifier of selected request.
FREE	: boolean indicating if the resource is free
SIZID	: number of ids

```
PROC MORREQSEMONEID(REQ[1..N],GRT[1..N])
cobegin
   while true do free:=not(grt[1..N]);
   while true do
     begin
        while not(free) do skip;
        idrq:=(idrq+1) mod sizid;
        while not(req[idrq]) do idrq:=(idrq+1) mod sizid;
        grt[idrq]:=true;
     end;
     od;
   while true do
          (* reset grant if request is removed *)
     od;
coend;
```

Each group of resources will be checked sequentially. After inspecting all possible requests of an identifier and finding no request the module starts checking another identifier resource group avoiding that processes keep waiting to get a free resource. The request of that ID group is not granted, because the processor would keep checking another identifier wherefor no requests are generated.

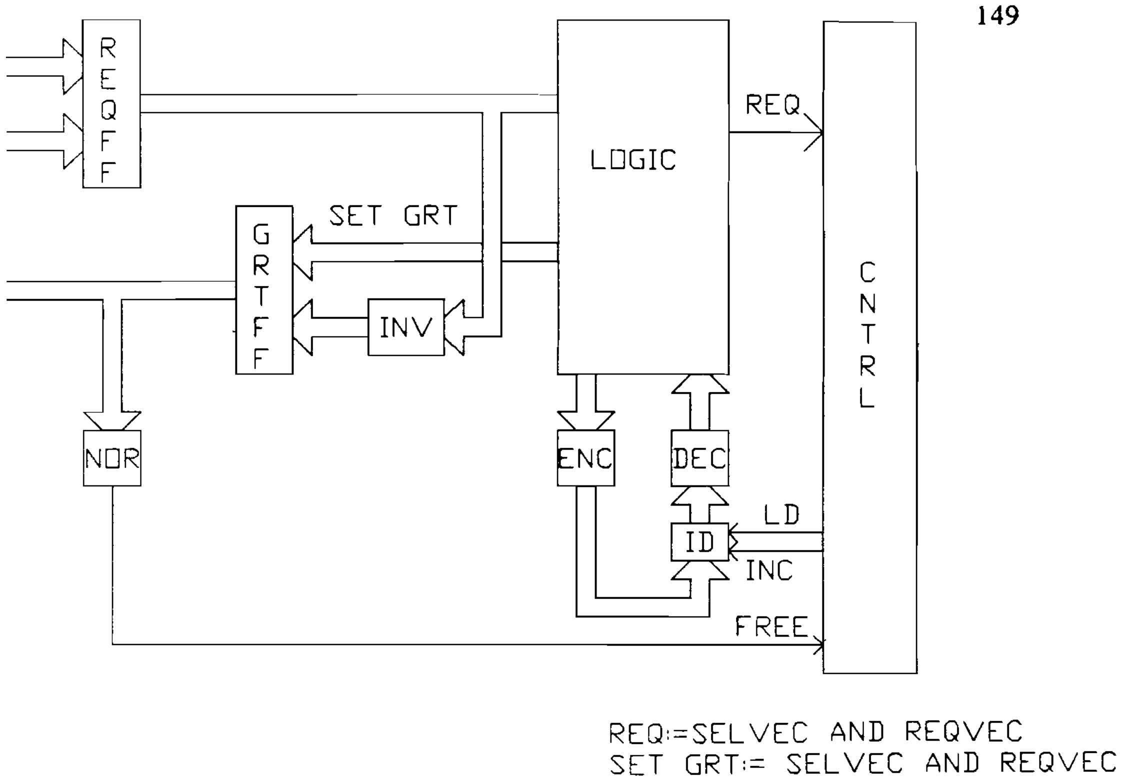


figure 97: sequential arbiter for one identifier

SELSET	: number of resource identifier group
SIZSELSET	: total number of existing groups
FREE	: boolean indicating if selected group is free
RAM[I]	: memory containing the pointers of the resource groups
DUMP	: variable holding the pointer start value to check if all requests are checked

SEQ RESOURCE MANAGER PSPEC:

```
proc resman(req,grt);
begin
   while true do
     begin
       selset:=(selset+1) mod sizselset; (* select next group of semaphores *)
       if free then
            begin
              dump:=ram[selset]; (* copy start number *)
              ram[selset]:=(ram[selset]+1) mod sizram[selset]; (* set pointer to new set of
                                  resources *)
              while (not(grt[ram[selset]]) and ram[selset]≠dump) do
                          (* stop condition: a grant is given or all request are checked *)
                          if req[ram[selset]]
                                  then   grt[ram[selset]]:=true
                                  else   ram[selset]:= (ram[selset]+1) mod sizram[selset];
                   od;
            end;
     end;
     od;
end;
```

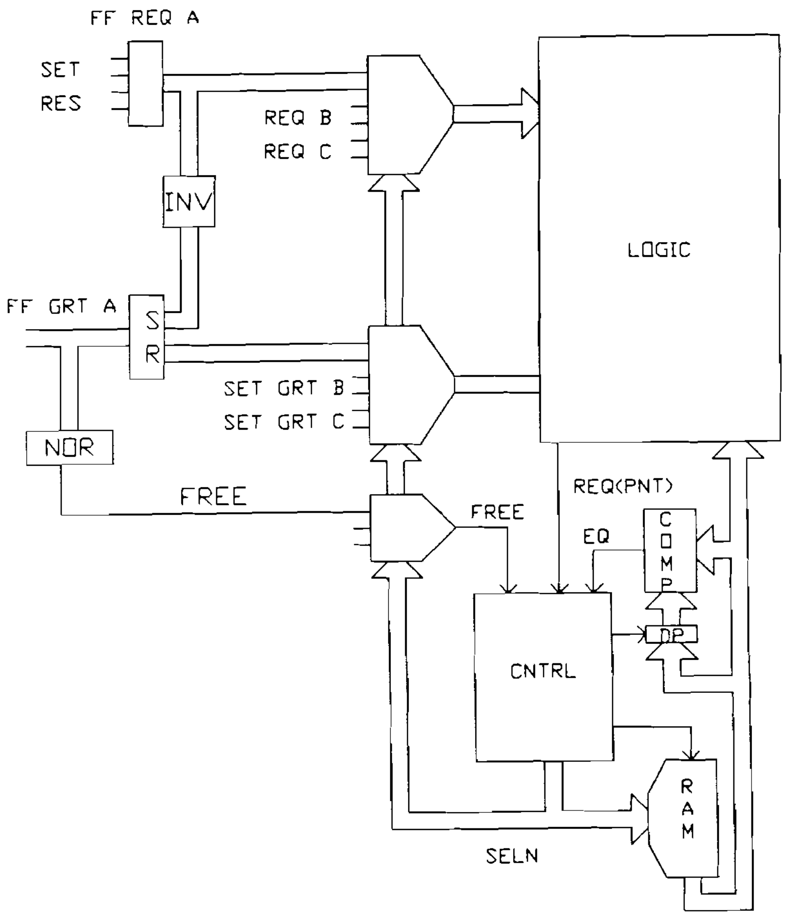


figure 98: sequential arbiter

2.4. READ/WRITE RESOURCE MANAGEMENT

The read/write synchronization allows several processes to read a shared variable together. A process requesting to write, will be the only process allowed to access the resource. Therefore all read requests are translated into one requested. (see figure 99)

```
RDWRRM(REQR[1..N],GRTR[1..N],REQW[1..M],GRTW[1..M]);
cobegin
    while true do free:=nor([grtrd,grtw[1..M]]);
    resman([reqw[1..M],reqrd],[grtw[1..M],grtrd]);
    while true do
       begin
          while not(grtrd) do reqrd:=or(reqr[1..N]);
          COfor i:=1 to m do grdrd[i]:=reqrd[i];
          while not free do skip
          reqrd;=false;
          while grdrd do skip;
       end;
       od;
coend;
```

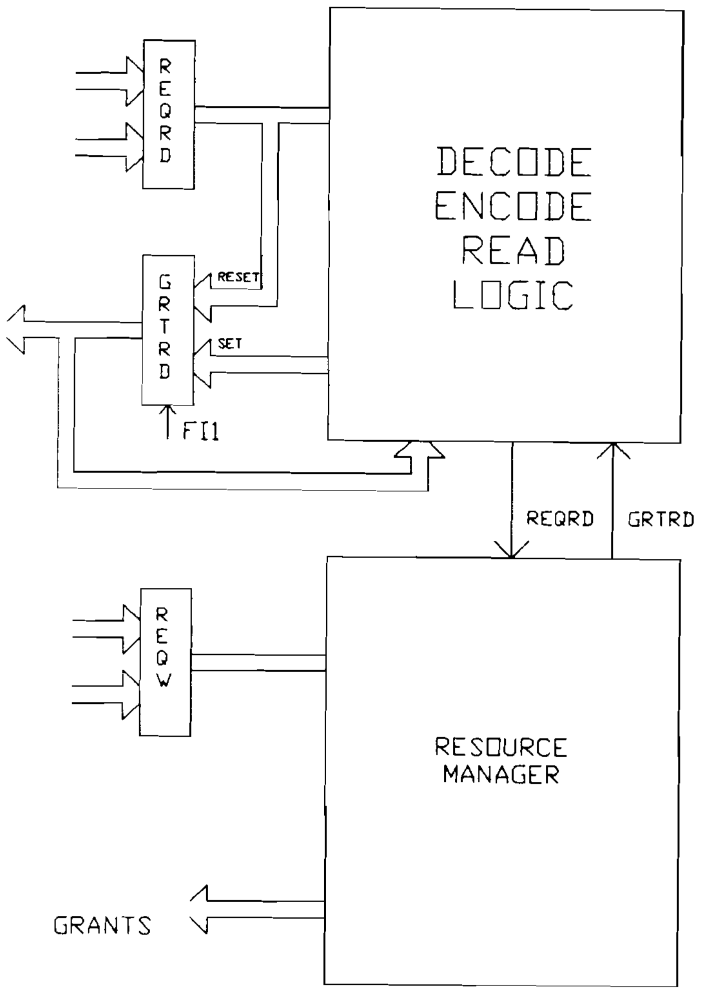


figure 98: implementation of r/w management

A handshake protocol is used otherwise a newly arrived read request could keep the REQRD active and cause starvation for the write process.

3. scheduling architecture modules

In the scheduling architecture module several identifiers combinations and even data/control transformations are allocated. Every identifier is processed in parallel. In the following paragraphs we discuss the basic technique for signalling and requesting, required to implement process synchronization techniques introduced in the literature for parallel processes, restricted to signal and wait, semaphore techniques. The SAM has at least two partners: a requester and a signaller.

The processing module statements.

```
WAITPROC(ID):        WHILE GRT DO SKIP;
                     REQ:=TRUE;
                     WHILE NOT (GRT) DO SKIP;
                     REQ:=FALSE;
```

In the implementation the GRANT signal is removed in the clock period following the removal of the request making the first statement redundant.

The request may be placed in the wait loop, since a request once set won't be reset until the GRANT is set. The statement signalling to the SAM that a part of the condition is true, is:

```
SIGPROC(ID):         SIG:=TRUE;
```

THE SCHEDULER PSPEC FOR ONE ID:

```
WREQPROC(ID); (* wait for request for condition checking*)
TEST
GRTPROC(ID); (* grant Id *)
WREMREQPROC(ID);(* wait for removal request *)
REMGRTPROC(ID);(* remove grant *)
```

We will start with a boolean implementation of process synchronization, that can be simplified to implement a boolean semaphore. This semaphore is an important tool for the split binary semaphore used to implement mutual exclusion and synchronization of critical sections [Dykstra]

3.1. THE BOOLEAN PROCESS SYNCHRONIZATION.

The scheduler receives a request (req) and several information signals (sig1, sig2,....). The grant is given to the requesting process when all signal info is correct.

Using a boolean implementation restricts the signaling. No new signal may be given before the signal is reset. One process can't signal two processes together using one signal, so a second identifier is needed. If one identifier signals several processes, it is implemented with separate identifiers (the number is equal to the number of processes). Depending on the trade off between speed, silicon area an integer semaphore can be used.

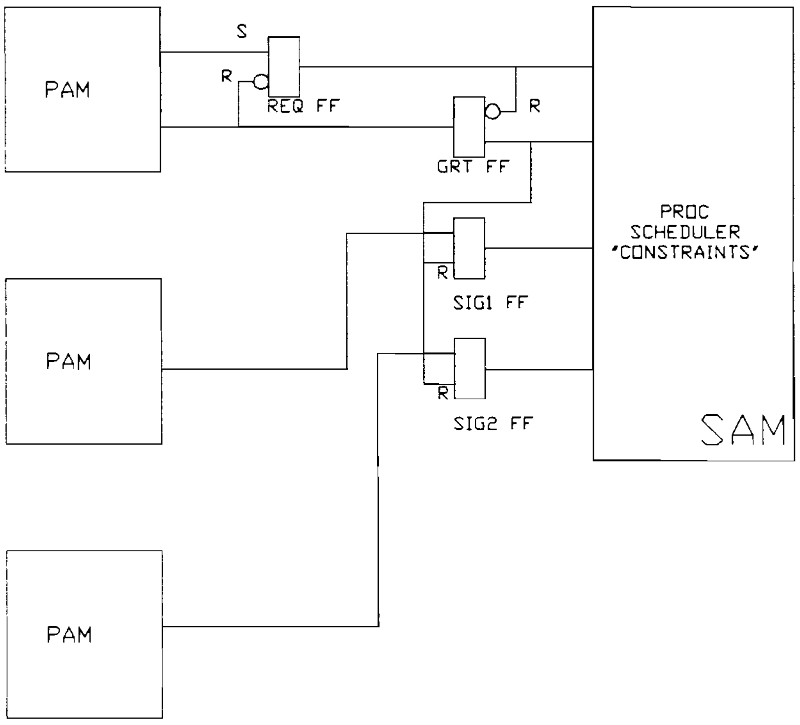


figure 100: one request, several signals

```
PROC BOOLEAN SCHEDULING PSPEC;
cobegin
  while true do
    begin
      while not(req) do skip;
      while not(sig1 and sig2 and sig3.....) Do skip;
      cobegin
            grt:=true;
            sig1,sig2,......:=False;
            req:=false;
      coend;
    end;
  od;
  while true do
    begin
      while not(ssig1) do skip;
      sig1:=true;
    end;
  od;
  while true do
    begin
      while not(ssig2) do skip;
      sig2:=true;
    end;
  od;
coend;
```

3.2. THE INTEGER SEMAPHORE SCHEDULER

```
WAITPROC(ID)        : WHILE CNT=0 DO SKIP;
                      CNT:=CNT-1;
SIGPROC(ID)         : CNT:=CNT+1;
```

The independent actions (sigproc, waitproc) use the same resources, which can be accessed at the same moment needing resource management. The test cnt>0 doesn't have to be protected. The parallel process doesn't change the status cnt>0. The variable SSIG (set signal) is a control signal setting the signal FF and increasing the counter. We describe only one signal source.

```
PROC INTEGER SCHEDULER;
cobegin
  while true do
     begin
       while not(req) do skip;
       while not(sig1>0) do skip;
       cobegin
          grt:=true;
          req:=false;
          begin
            waitres(sigid1);
            cobegin
               sig1:=sig1-1;
               relres(sigid1);
            coend;
          end;
       coend;
       grt:=false;
     end;
   od;
   while true do
     begin
       if ssig1
          then
            begin
               waitres(sigid1);
               cobegin
                  sig1:=sig1+1;
                  ssig1:=false;
                  relres(sigid1);
               coend;
            end;
     end;
     od;
coend;
```

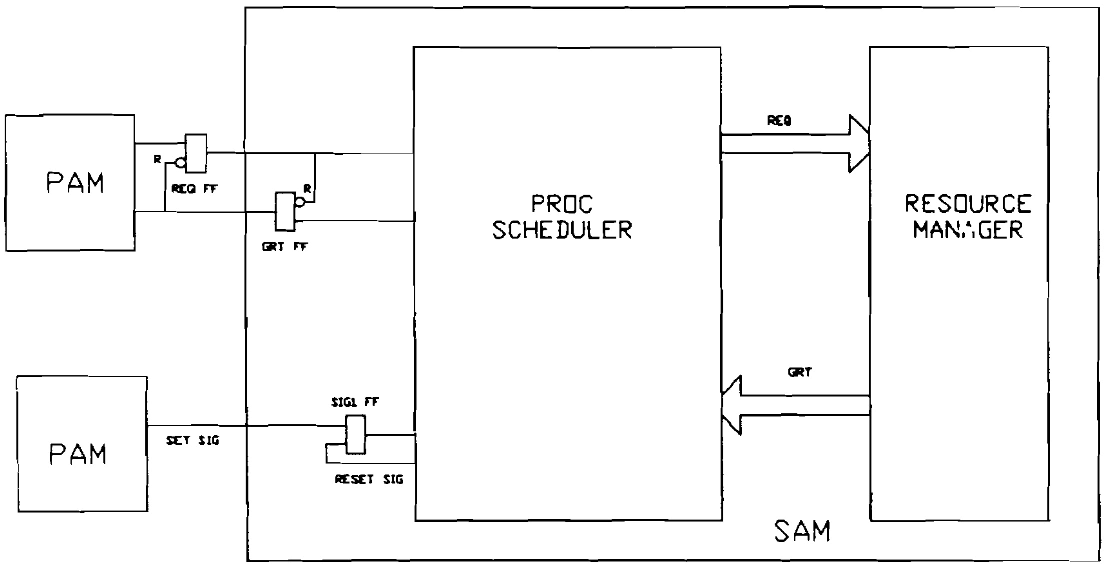


figure 101: integer semaphore SAM.

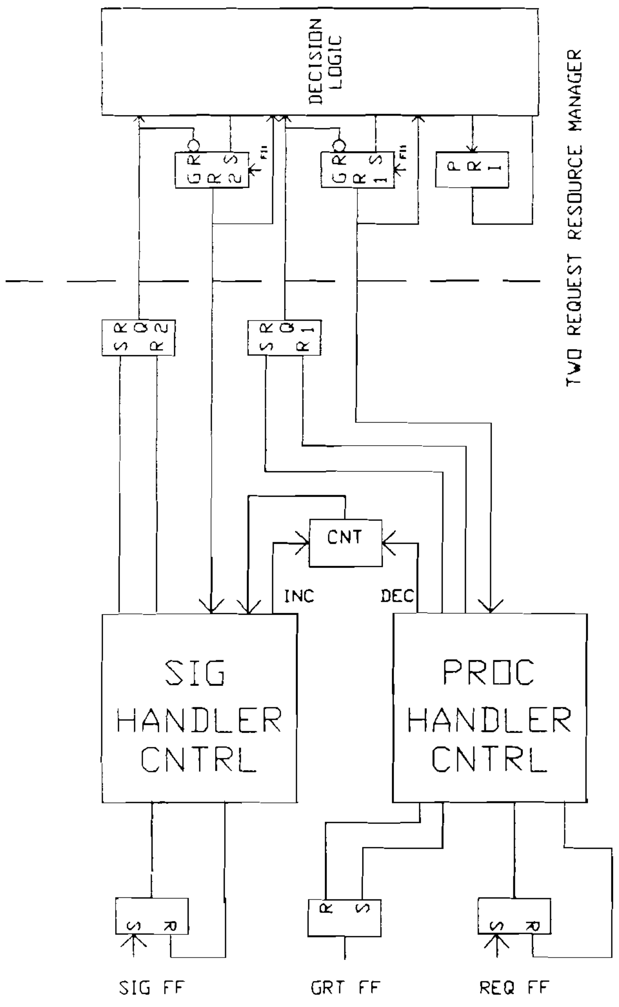


figure 102: implementation

One identifier can also be signalled by several processes. To process these signals we introduce an arbiter. But the signalling process never knows, if its signal is processed. In this configuration it takes at most 8 clockcycles to process the signal. If it is necessary to generate an acknowledge, an identical protocol as for the **waitproc** can be used. This scheduler is an example where parallelism introduces so much overhead, that the centralized control scheduler is faster.

centralized solution.
PROC CENT_ID_SCH PSPEC:

```
while true do
  begin
    if req and sigl>0
          then
            cobegin
              grt:=true;
              req:=false;
              sigl:=sigl-1;
            coend;
          fi;
    if ssigl
            then
              begin
                sigl:=sigl+1;
                ssigl:=false;
              end;
          fi;
  end;
od;
```

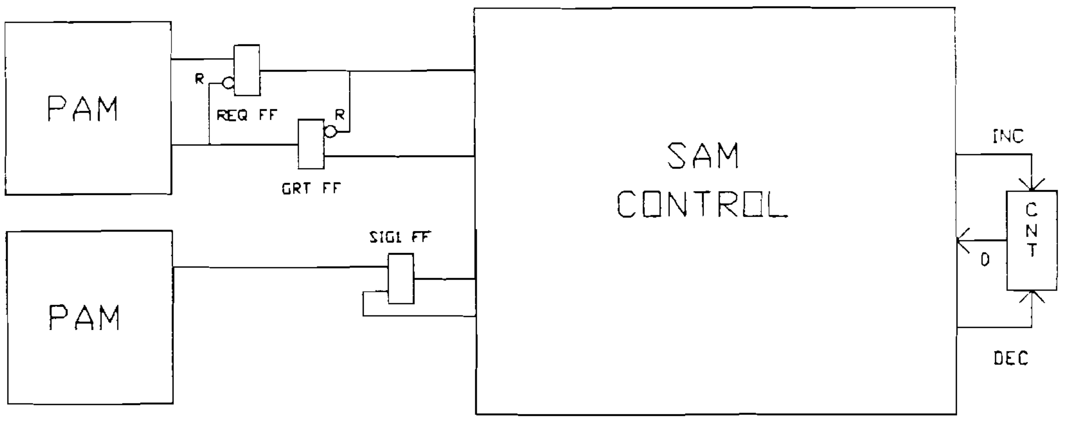


figure 103: implementation centralized integer semaphore control

In this solution the signal can be processed every two clockcycles. Finally, where several identifiers should have an correct value. For one requester and several identifiers the program description is:

```
PROC INTEGER SCHEDULER;
cobegin
  while true do
     begin
       while not(req) do skip;
       while not(sig1>0 and sig2>0) do skip;
       cobegin
          grt:=true;
          req:=false;
          begin
            waitres(sigid1);
            cobegin
               sig1:=sig1-1;
               relres(sigid1);
            coend;
          end;
          begin
            waitres(sigid2);
            cobegin
               sig2:=sig2-1;
               relres(sigid2);
            coend;
          end;
       coend;
       grt:=false;
     end;
   od;
   while true do
     begin
       if ssig1
          then
            begin
               waitres(sigid1);
               cobegin
                  sig1:=sig1+1;
                  ssig1:=false;
                  relres(sigid1);
               coend;
            end;
     end;
   while true do
    begin
       if ssig2
          then
            begin
               waitres(sigid2);
               cobegin
                  sig2:=sig2+1;
                  ssig2:=false;
                  relres(sigid2);
               coend;
            end;
     end;
     od;
coend;
```

Several identifiers can be used in the split binary semaphore solutions. When a process isn't allowed to start, before processes have signaled an event, the count initialization of the integer semaphore should be negative.

3.3. SEVERAL REQUESTERS AND SIGNALERS ON ONE IDENTIFIER

The most used forms of scheduling are the boolean implementation and the integer semaphore with one ID, where processes execute a **P** and V operation. Several processes can execute a **P** operation on the same semaphore, requiring a module to serve the requesting processes. The server sees the request, gives the request to the scheduler, waits until a grant arrives and signals this to the requesting process and tries to serve another requester. Starvation is avoided.

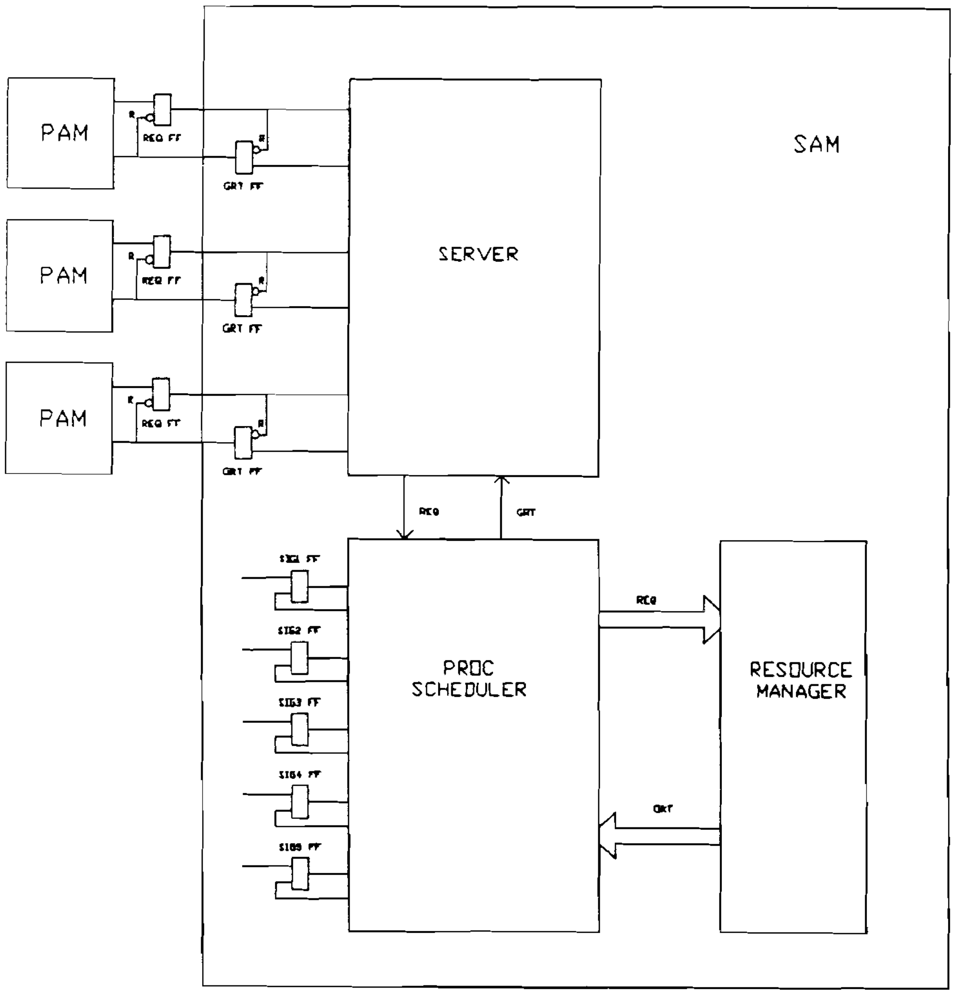


figure 104: SAM using server

Generalizing the last remarks and also introducing several processes signalling the same identifier, we get the model pictured in figure 105.

All solutions can be reduced to a centralized system by removing all resource management statements by putting the processes in a sequential order. The server arbiter will be used to avoid starvation. The centralized solution is the best solution, because it avoids resource management which causes too much overhead.

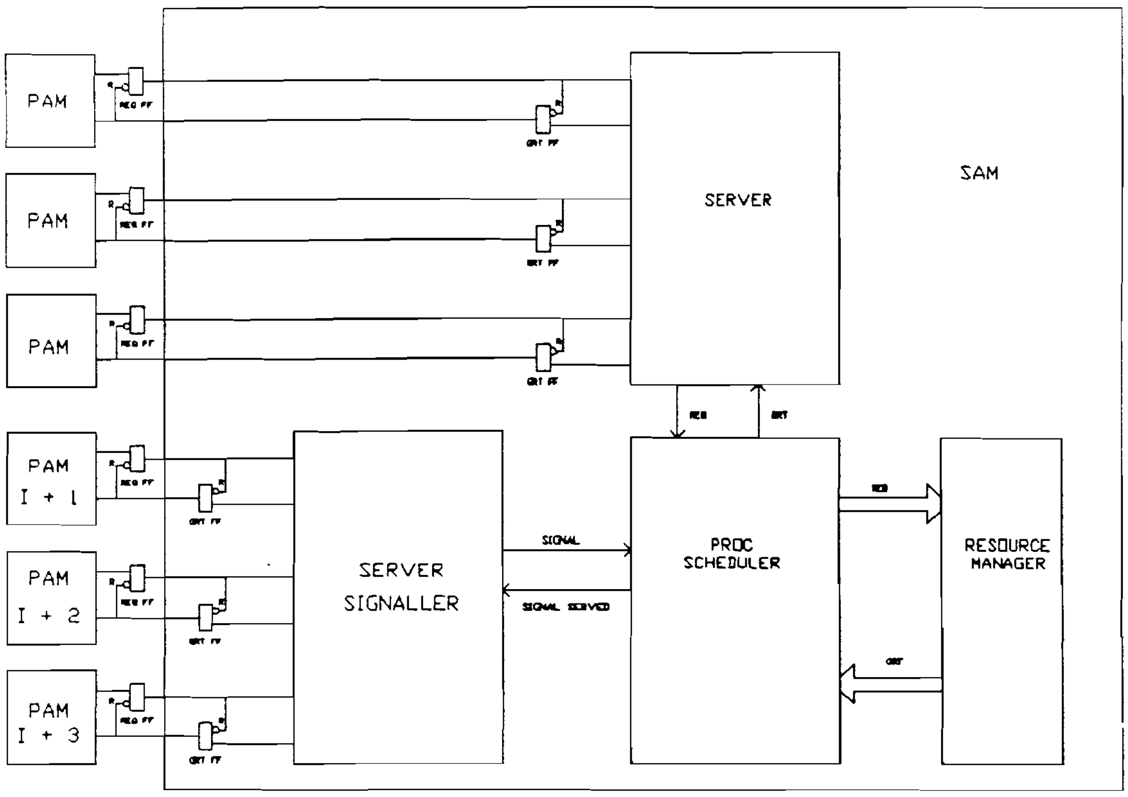


figure 105: SAM using servers for one ID

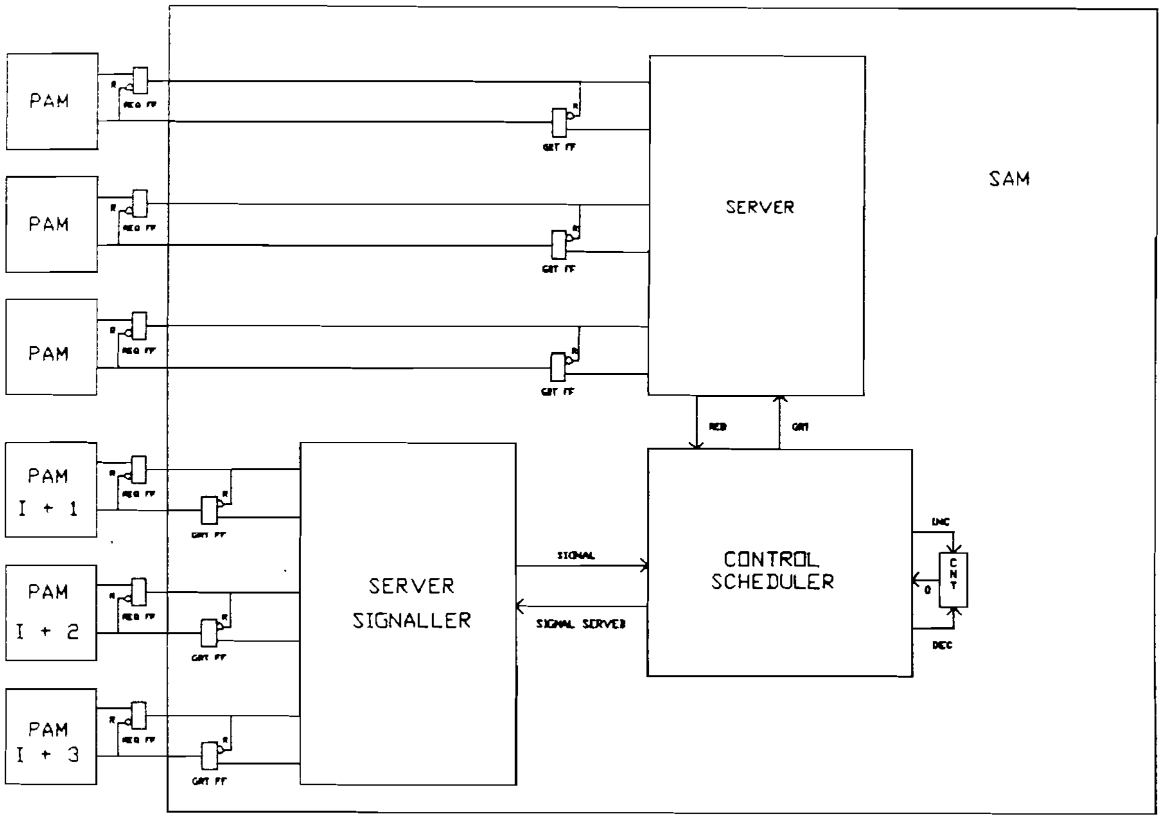


figure 106: SAM using servers and centralized control

Signalling an identical sig identifier, signals could be lost. A server arbiter could be used, giving probable long serving time. It is easier to introduce several boolean identifiers, one for every signalling process. (this is only true for parallel implementation) When a signal can be lost, then the signaling processes should be acknowledge after processing the signal.

4. request acknowledge synchronization.

```
REQ(ID):        REQ:=TRUE;
                ACK:=FALSE
WACK(ID):       WHILE NOT(ACK) DO SKIP;
WREQ(ID):       WHILE NOT(REQ) DO SKIP;
ACK(ID):        ACK:=TRUE;
                REQ:=FALSE;
```

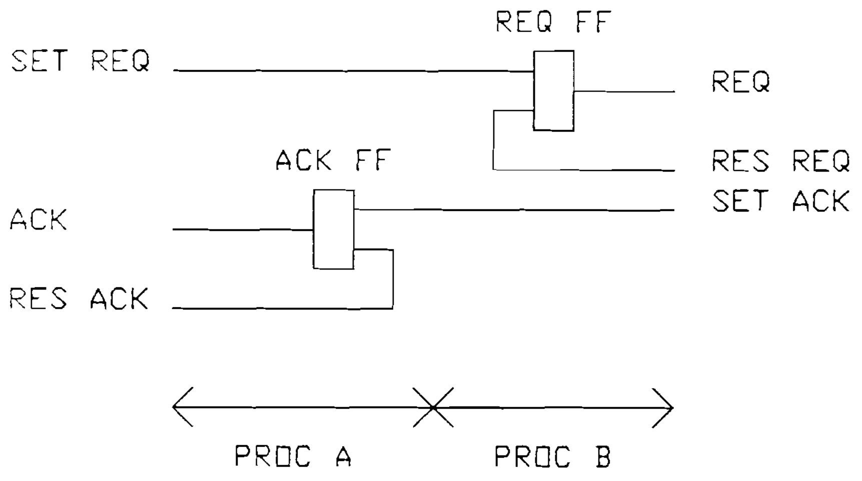


figure 107: implementation of r/a synchronization

This implementation can be simplified by just using one FF and wire or the set and reset inputs.

C. SHARED PROCESSING PROTOCOL EXECUTION AND SERVER IMPLEMENTATION

1. introduction

In this appendix the protocol execution between the server, the **CPAM** and the **SPAM** is discussed. For every allocation we try to multiplex lines between modules, to simplify the protocol and to optimize the resource allocation. An implementation of the server is presented. We also introduce a parallel server. This appendix is a supplement to the chapters 5,6,7.

2. direct connection shared processing

2.1. HARDWIRED PROCESS CALL MODEL

2.1.1. one spam
We discuss the timing of the allocation presented in figure 108.

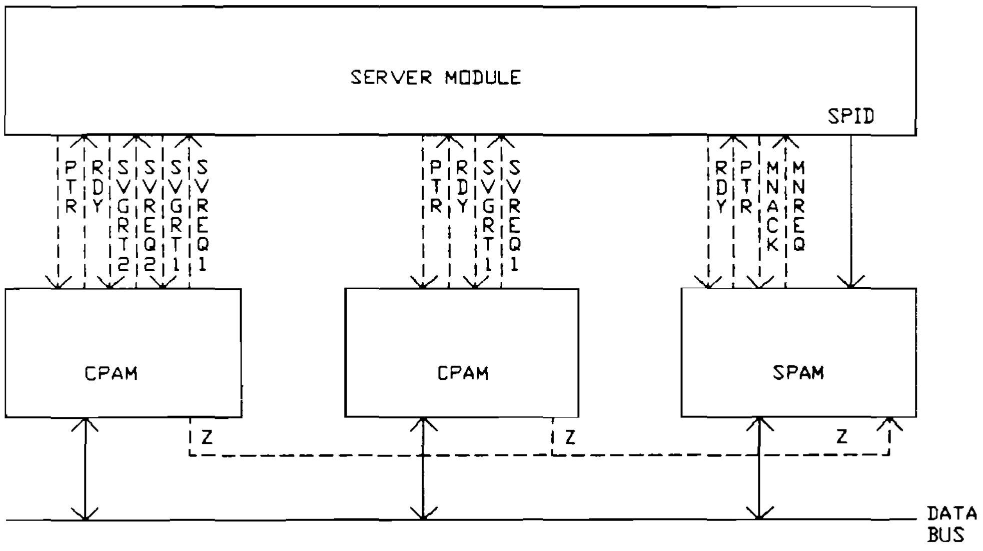


figure 108a: hardwired call process AFD

When a caller isn't ready to fetch the results, the shared processing module will be idle for awhile. In the timing diagram we see that it is possible to multiplex PTR en RDY line with the REQ and ACK lines, as long as we can interpreted the signals correctly.

After selecting the requester the protocols between the server and SPAM and the server and CPAM are executed independently. So action 8 can happen before action 4. This can give misunderstanding during multiplexing.

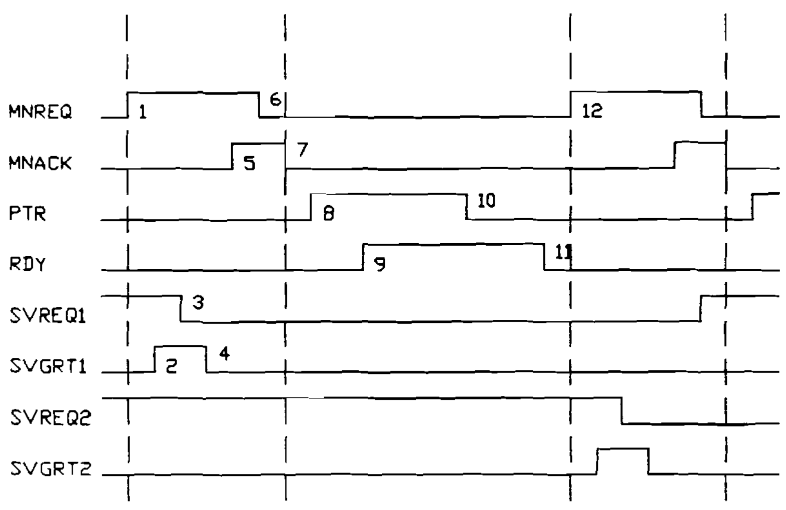


figure 108b: hardwired call process timing

action:

1.	request of the shared process monitor for a new partner
2.	signal to new partner
3.	calling process acknowledges receipt of grant
4.	remove grant
5.	acknowledge to monitor that partner is found
6.	monitor acknowledges receipt of acknowledge
7.	remove acknowledge
8.	request for parameters
9.	parameters transmitted
10.	result ready
11.	result copied
12.	new request

periods:

1-2	decision time of the server (one clockcycle if request is pending)
2-3	response time depending on the calling process was already waiting for the grant or not
3-4	one time unit, just response
3-5	one time unit; sequential processing
2-5	set up communication between caller and processor
5-6	depends on monitor program (could be one clock)
6-7	one time unit, just response
7-8	response time depends on initialization
8-9	minimal time for parameter communication, which can contain waiting time
9-10	processing time
10-11	result communication and perhaps some waiting time depending on program
11-12	response time

We introduce a server and a connect block in the server architecture module (SVAM). The server has the structure of the server arbiter presented in appendix B. Some hardware is required to translate the identity of the called process request into the correct SPID. The request lines can be used for other signals, since the server won't monitor them unless it is in the correct state. (see figure 109).

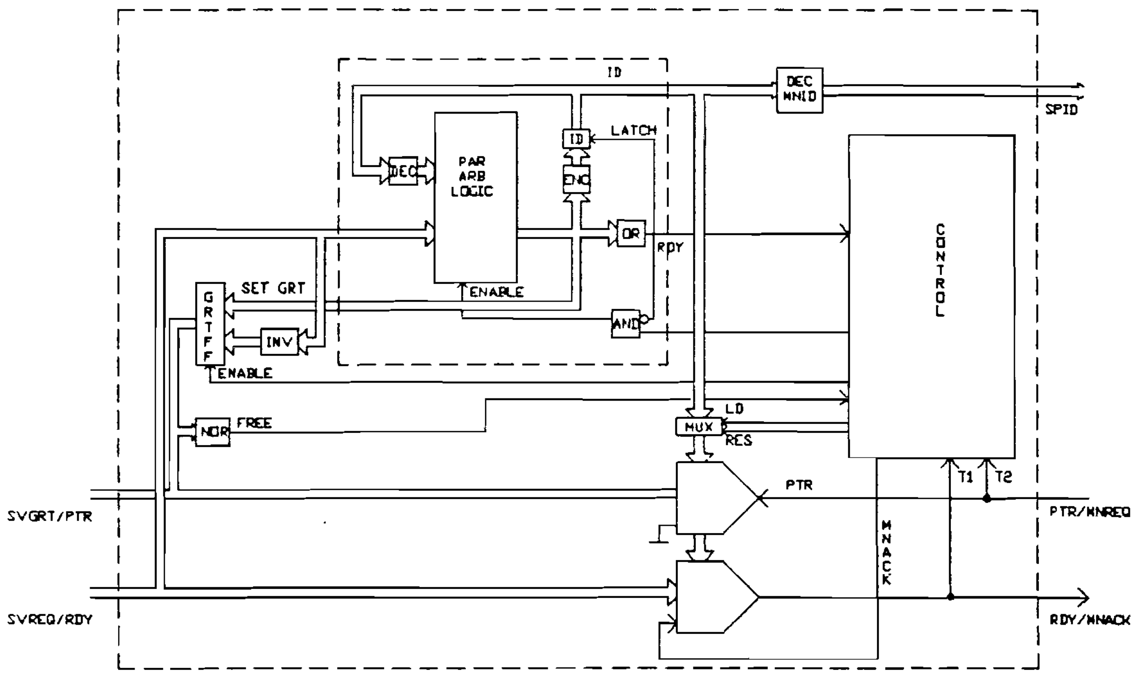


figure 109: server module implementation

After the calling process and monitor request are removed, the pointer of the multiplexers will be set, so that the MNREQ, MNACK, SVREQ and SVGRT lines can be used for the parameter and result synchronization. The numbers in the description refer to the timing chart (fig 110).

```
EXTENDED SVAM PSPEC:
cobegin
  while true do (* grant logic *) od;
  while true do
     begin
        wmnreq(idspam); (* 4 wait for the request for anew process to serve*);
        while not(rdy) do servarb(svreq[1..M],true,id,rdy);
        svgrt[id]:=true; (* 4a; correct process spid is ready *)
        cobegin
           engrt:=true; (* 5; set ack up line *)
           mnack(idspam);
        coend;
        cobegin
           wremmnreq(idspam); (* 6*)
           wremsvreq(idcpam); (* 7; while not(free) do skip, so no wrong interpretation is
           possible. When the connect is set-up before the svreq is removed, the svreq
           will be seen as rdypa*)
        coend;
        cobegin
           engrt:=false; (* 9*)
           remmnack(idspam);
        coend
        mux:=id; (* 10; set up link, now reset mnack *)
        wrdy; (* 12; the controller monitors the changes of mnack/rdy line *)
        wremrdy; (*14 necessary to keep track of the state *)
        mux:=0;(*3/15; so no svreq will be seen as mnack*)
     end;
  od;
coend;
```

The specification produces the following timing:

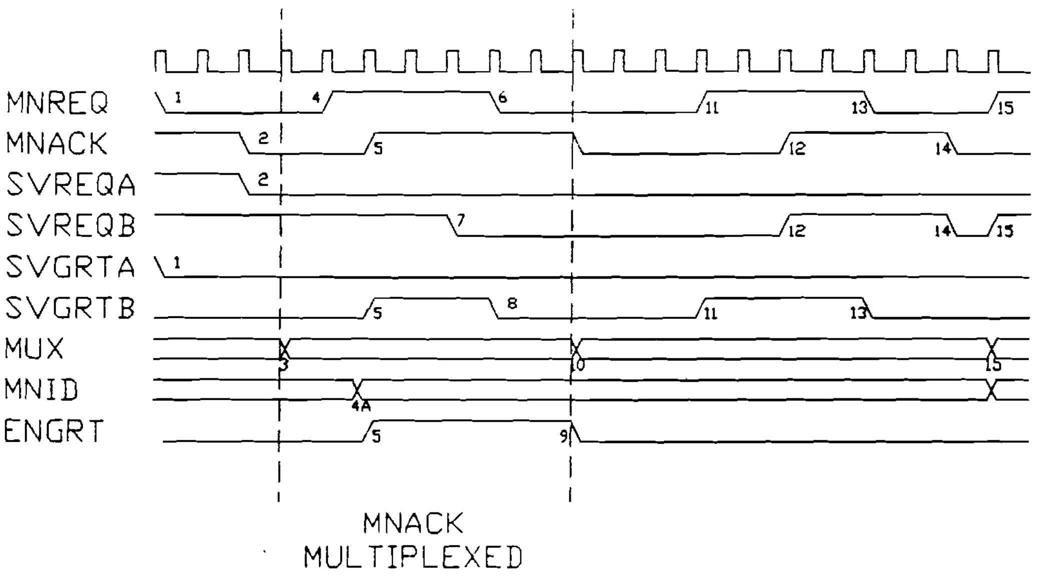


figure 111: timing hardwired server

action:

1.	result ready
2.	result fetched
3.	reset of pointers;(tristate for PTR/MNREQ and the RDY line goes high)
4.	MNREQ
4a.	SPID ready
5.	SVGRT and MNACK given
6.	removal of MNREQ
7.	removal of SVREQ
8.	removal of SVGRT
9.	enable low because of 6 and 7
10.	set multiplexer, MNACK goes low
11.	request for parameters
12.	parameter transported
13.	result ready
14.	result fetched
15.	connection break down

The periods:

1-2	unknown
2-3	one clockcycle
3-4	depends on SPAM
4-5	one clockcycle, if requests depending
5-6	depends implementation to start up a process
5-7	depends on implementation; 8 is the bottleneck
7-8	one clockcycle
8-9	zero clockcycles (see implementation)
10-11	can be zero, depends on the implementation of the SPAM, This doesn't introduce any problem
14-15	one clockcycle

No problems arise the multiplexing SVREQ with PTR, because when PTR is active the arbiter is always disabled. Some signals can be changed into pulses, since the handshake isn't used. When a module is waiting for a signal, a pulse is sufficient. The MNACK and the start signal for the server can be a pulse.

2.1.2. parallel operating spams

2.1.2.1. USING RESOURCE MANAGEMENT

The connect module of the SPAM can be found in figure 112. Simplifying the server's control logic by moving the connect module to the SPAM should be introduced in the one-one situation. The connection module doesn't need a multiplexer for the z's signals in the one-one situation.

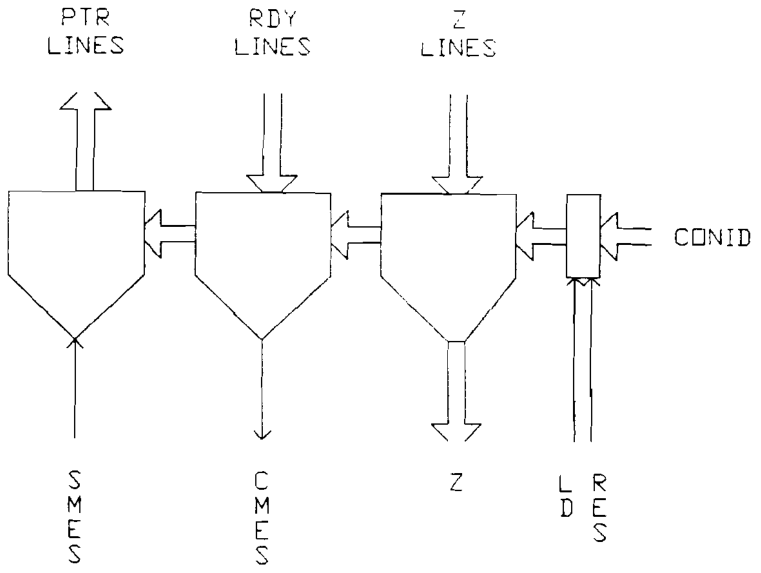


figure 112: connect module

Since we separated the connect module from the server, we get a new server description, which will be presented after discussing a few further improvements. The server won't be given free by the SPAM, before the SPID and CONID are fetched.

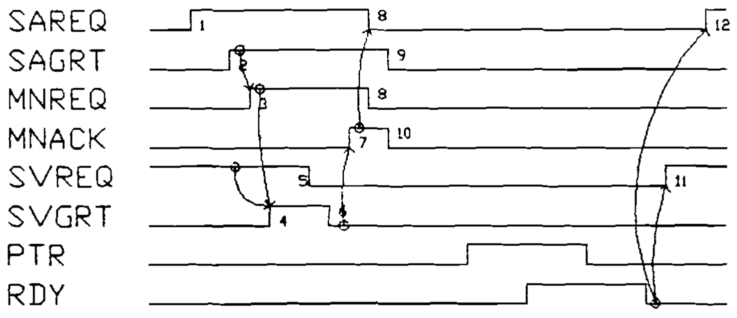


figure 113: the communication protocol using resource management.

action:

1	: request for the shared resource: the server
2	: request granted
3	: MNREQ starts arbiter
4	: the pending SVREQ is granted
5	: acknowledge of CPAM for the SVGRT
6	: SVGRT removed
7	: MNACK given
8	: request for server removed. This is only possible, when the SPID and CONID are clocked at the rising edge of the MNACK
11	: the result are fetched. The CPAM requests a new service
12	: the SPAM requests the server. It wants a new process. (NO relation exists between eleven and twelve)

Remark: the period between seven and eight can be longer than one clockcycle. This depends on the identifier's storing implementation.

The resource management allows the several MNACK and MNREQ lines to be multiplexed becoming low active lines. The resource management timing chart indicates that even more lines can be multiplexed. SVGRT, MNACK and SAGRT can be multiplexed, because the receiver knows which answer to expect. SAREQ, SVREQ and MNREQ can't be multiplexed. Every new pulse is interpreted as a new request. We will multiplex the SVGRT and PTR lines (low active). We use "\" to indicate a low active signal.

The data bus can be used for the SPID and CONID transport. The MNREQ line is a RFD (ready for data), the MNACK line is a DAV (data available). This is allowed, since the sequencing is known. When already RFD and DAV lines exist, they can be used. The advantage of this solution is a simpler server and less lines. Several sources activate the RFD and DAV line, so these lines are low active.

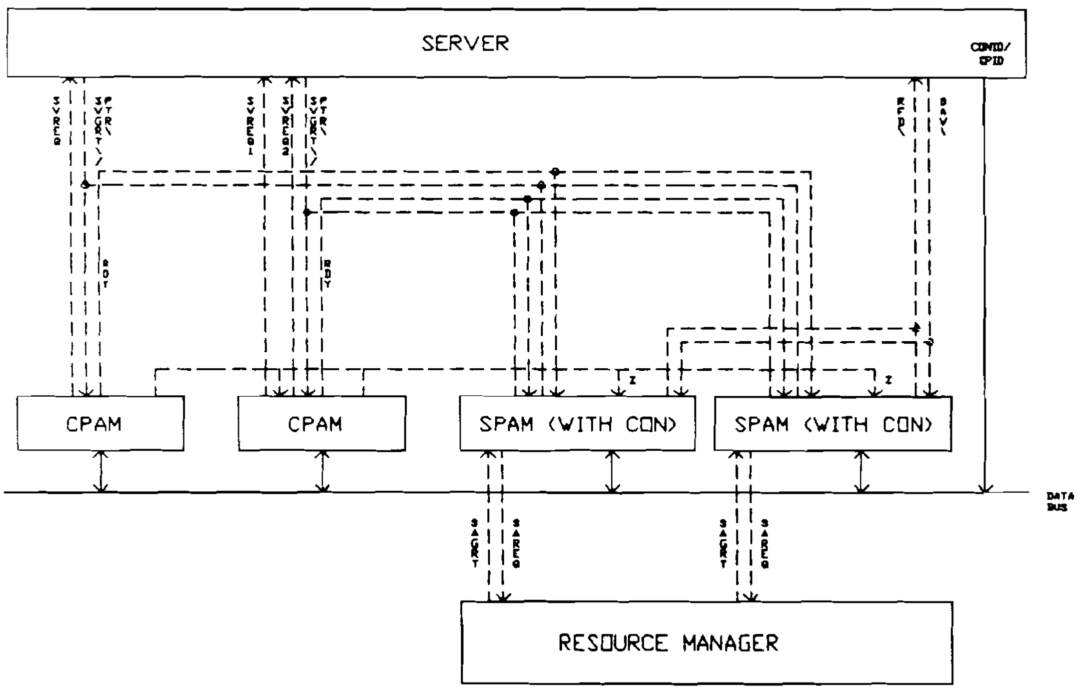


figure 114: AFD/AID optimized configuration

The calling process description doesn't change, only the shared process monitor.

SPAM MONITOR PSPEC:

```
while true do
    begin
      cobegin
            res__conid;
            waitres(sa);
      coend
      get(spid);
      get(conid);
      relres(sa);
      case spid do
            id1     : proc id1;
            id2     : proc id2;
            ..    .....
          end;
    end;
od;
```

SERVER CONTROL SPEC:

```
while true do
      begin
            while not(rdy\) do skip; (* wait until request *)
            str__sv:=true; (* start svreq logic *)
            while not(rdy__sv) do skip; (* wait until a service request is found *)
            str__sv:=false;
                  (* rdy\ already active *)
            cobegin
                  en__spid:=true;
                  dav\:=true; (* spid on the bus *)
            coend
            while rfd\ do skip; (* wait until loaded *)
            cobegin
                  en__conid:=false;
                  dav\:=false;
            coend;
            while not(rfd\) do skip;
            cobegin
                  en__conid:=true;
                  dav\:=true;
            coend
            while rfd\ do skip;
            cobegin
                  en__spid:=false;
                  dav\:=false;
            coend;
            while not (free__sv) do skip;
                  (* wait until acknowledge of svgrt *)
      end
od;
```

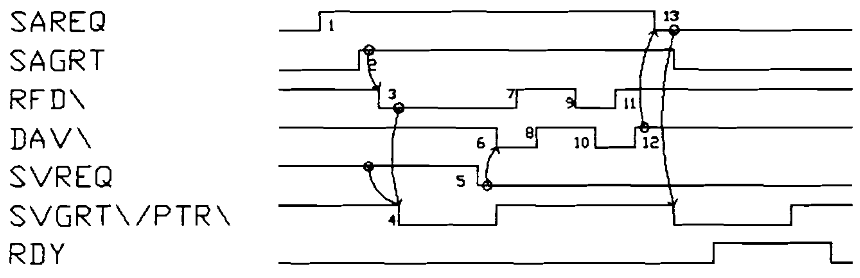


figure 115: hardwired call AFD using multiplexed lines

action:

1	: request for the shared resource: the server
2	: request granted
3	: SPAM ready for data, starts arbiter
4	: the pending SVREQ is granted
5	: acknowledge of CPAM for the SVGRT
6	: data byte put on bus
7	: data clocked
10	: next data byte
13	: request for server removed after clocking the SPID and CONID.

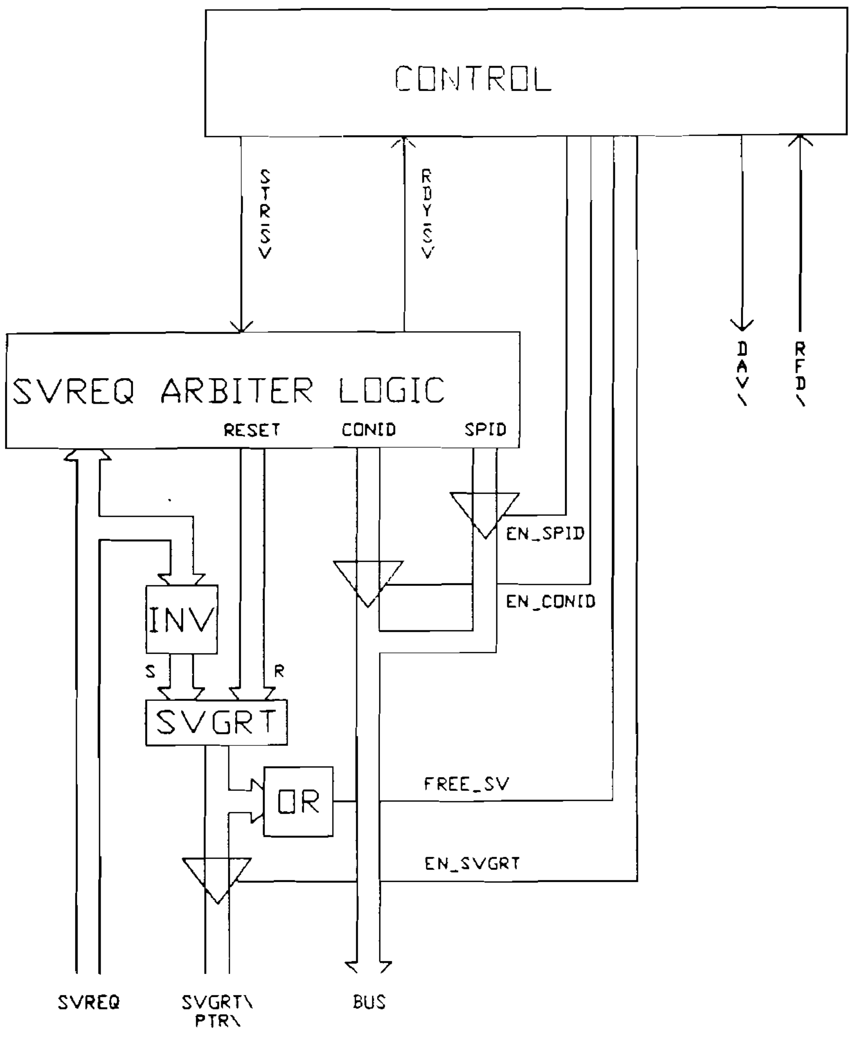


figure 116a: server with resource management

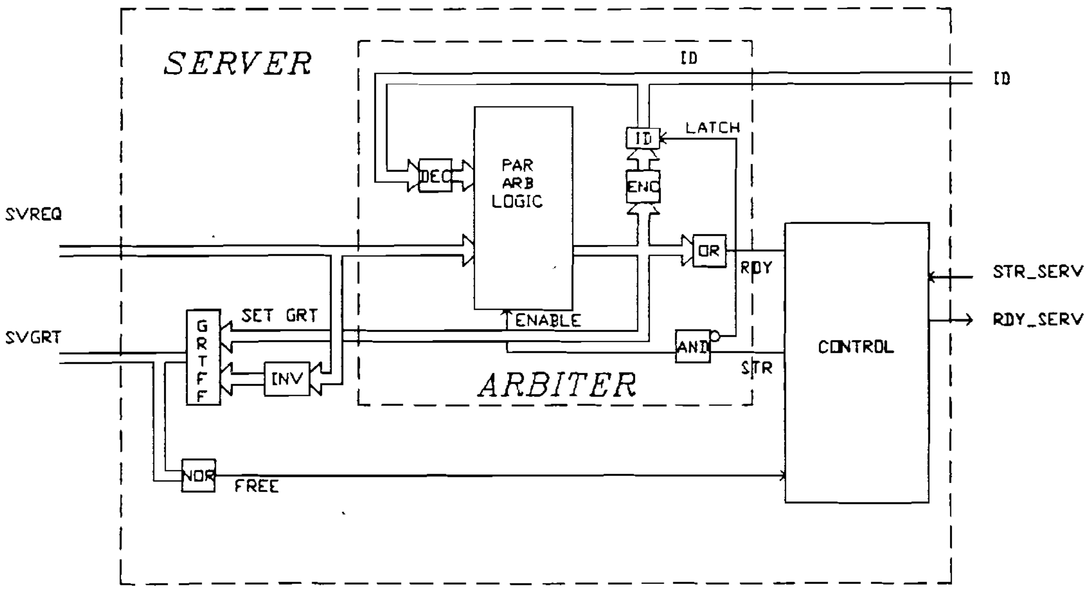


figure 116B: arbiter

2.1.2.2. NO RESOURCE MANAGEMENT

We remove the resource management and introduce a second arbiter in the server module multiplexing SVGRT and PTR. Multiplexing MNACK/RDY makes no sense. These lines are connected after the multiplexer.

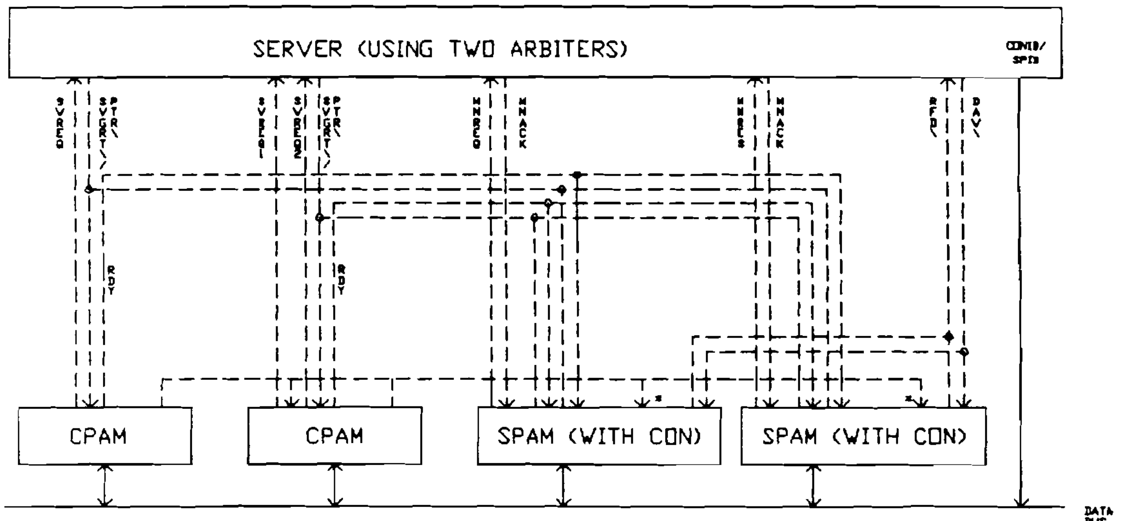


figure 117: AFD/AID without RMAM

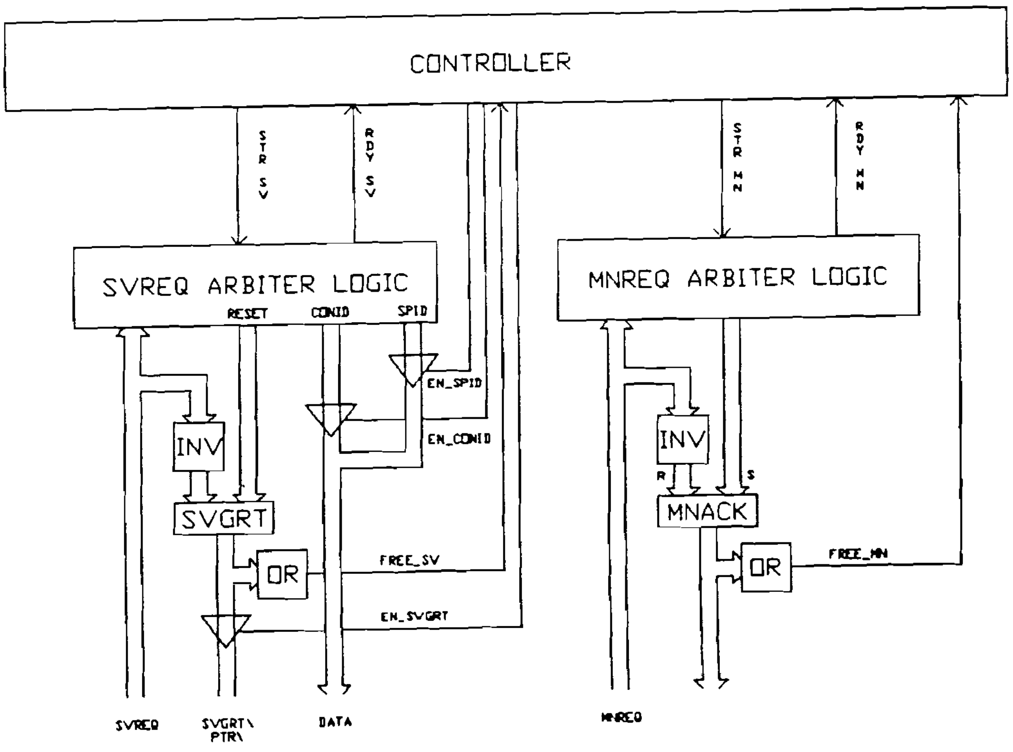


figure 118: two server implementation

SPAM MONITOR PSPEC:

```
while true do
    begin
        res__conid;
        mnreq(idspam); (* request new process to execute *)
        wmnack(idspam); (* spid and conid are ready *)
        get(spid);
        get(conid);
        remnnreq(idspam);
        case spid do
                idl     : proc idl;
                id2     : proc id2;
                ..   .....
                end;
    end;
od;
```

SERVER CONTROL PSPEC:

```
while true do
        begin
                str__mn:=true; (* enable mnreq logic *)
                while not (rdy__mn) do skip; (* wait until mnreq *)
                str__mn:=false;
                str__sv:=true; (* fetch a svreq *)
                while not(rdy__sv) do skip;
                str__sv:=false;
                (* rdy\ already active *)
                cobegin
                        en__spid:=true;
                        dav\:=true;
                coend
                while rfd\ do skip;
                cobegin
                        en__conid:=false;
                        dav\:=false;
                coend;
                while not(rfd\) do skip;
                cobegin
                        en__conid:=true;
                        dav\:=true;
                coend
                while rfd\ do skip;
                cobegin
                        en__spid:=false;
                        dav\:=false;
                coend;
                while not (free__sv) do skip;
                        (* wait until acknowledge of svgrt *)
                remmnreq(idspam);
        end
od;
```

The communication protocol is identical with the resource management timing chart. Only some lines have other names.(fig 119) This solution isn't really better. The number of lines doesn't differ. A few clockcycles are saved, because the same module that serves the monitor request executes the resource management.

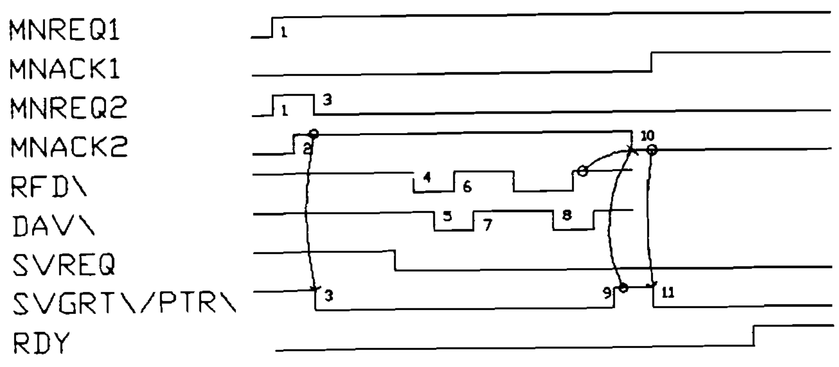


figure 119: timing hardwired call using two server arbiter and id on the bus.

2.2. COMMAND MODEL

2.2.1. one spam

2.2.1.1. THE PROTOCOL EXECUTION AND OPTIMIZATION

After requesting the **SPAM** and giving the parameters, **CPAM** executes its program until it needs the result and starts waiting for the result.

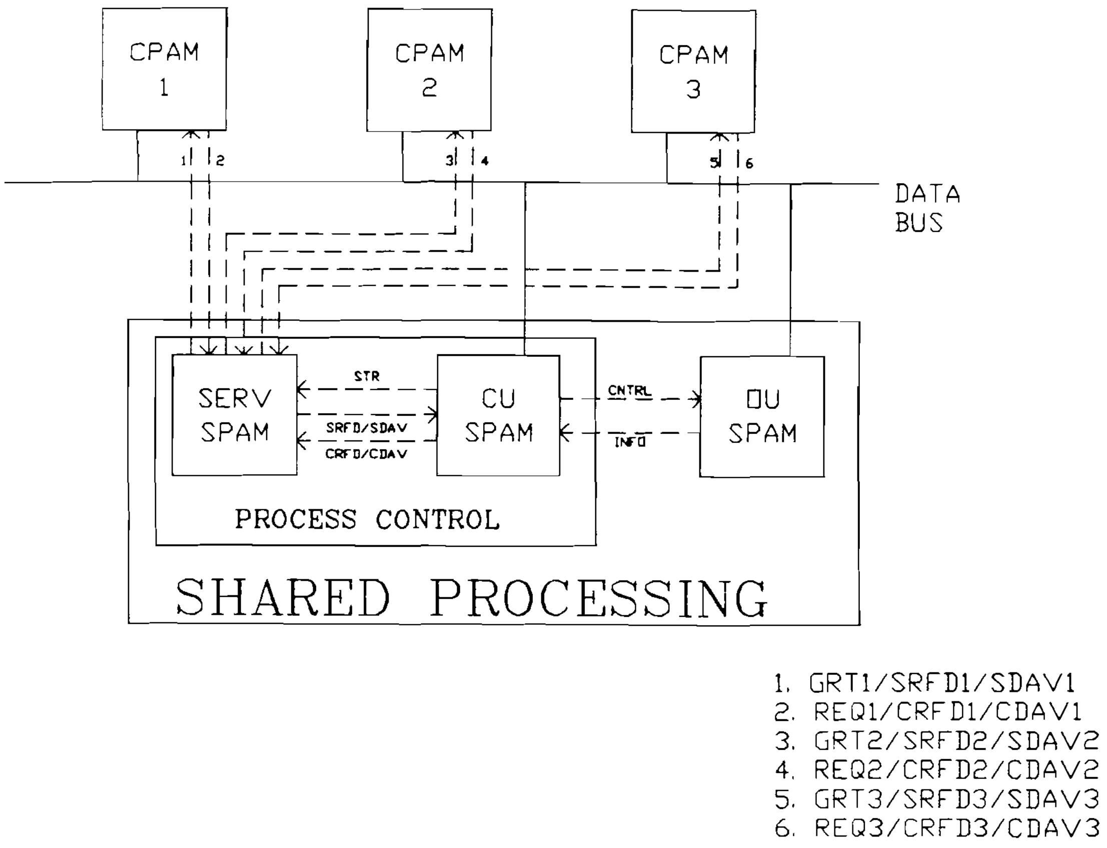


figure 120: cmd shared processing allocation

We use separate signals for the data transport by the caller (CDAV, CRFD) and the **SPAM** (SDAV, SRFD). (C=CALLER, S=SHARED). SRFD active means requesting the parameters and CRFD active requesting for the results. In figure 121 a timing and sequencing description of the communication using one caller (SVREQ, SVGRT) and one **SPAM** (MNREQ, MNACK) is given. In figure 122 you can find the connections between the modules.

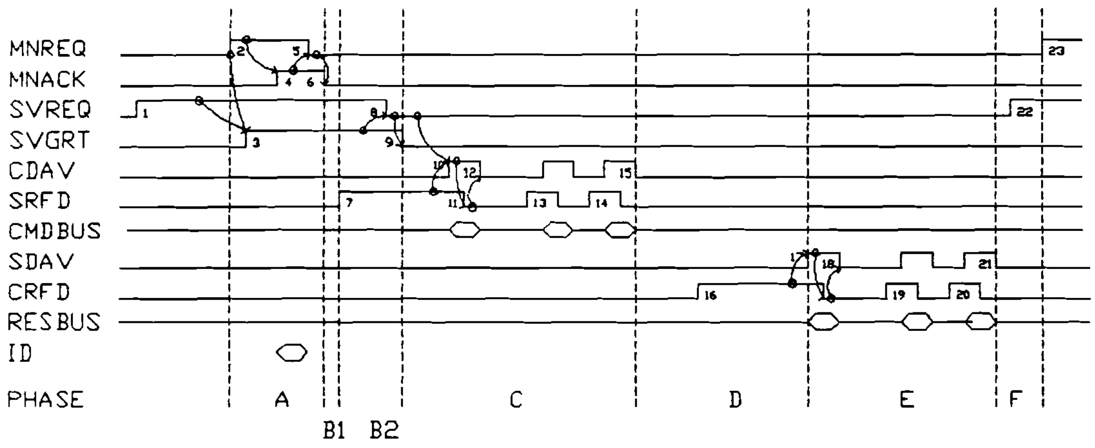


figure 121: basic sequencing

action:

1	: SVREQ of the CPAM, the SPAM is executing a program (no MNREQ exists)
2	: MNREQ; the SPAM asks for a new caller process
3	: SVREQ is granted
4	: monitor acknowledges that a caller is found.
5	: MNREQ removed; start-up of SPAM
6	: Server removes the MNACK
7	: SPAM is ready for the command word
8	: CPAM removes the request
9	: server removes grant
10	: CPAM puts CMD on the data bus
11	: SPAM copies data from bus
12	: CPAM removes the data
13	: SPAM asks for next parameter
14	: SPAM asks for another parameter
15	: CPAM removes the last parameter from the data bus
16	: CPAM signals that it can receive the first result (status)
17	: result ready; SPAM put status on the bus
18	: status removed from the bus
19	: CPAM asks for next result
20	: CPAM asks for another result byte
21	: last result removed from the bus
22	: CPAM generates a new request
23	: SPAM asks for a new CMD word (new caller)

The periods:

2-3	: one clockcycle; (arbiter response)
2-4	: response time could be one clockcycle;
4-5	: one clockcycle; monitor start the process preparation
5-6	: one clockcycle;
5-7	; depends on preparation actions
	================================
3-8	: depends on program in CPAM
8-9	: one clockcycle;
	================================
21-22	: depends on program
21-23	: depends on actions in SPAM

Sequencing:
(2,3,4,5,6,7) parallel with (2,3,8,9)
(7 and 8), 10,11,12,13,14,15,16,17,18,19,20,21
Action 7 and 8 should be executed before action 10.
(21,22) and (21,23) No relation exists between 22 and 23.

Phases:

A	: arbitration to find partner and to start handshake protocol
B1	: initialization of SPAM
B2	: SPAM ready while CPAM not ready (can be empty)
C	: parameter communication
D	: execution
E	: result communication
F	: clear action of SPAM

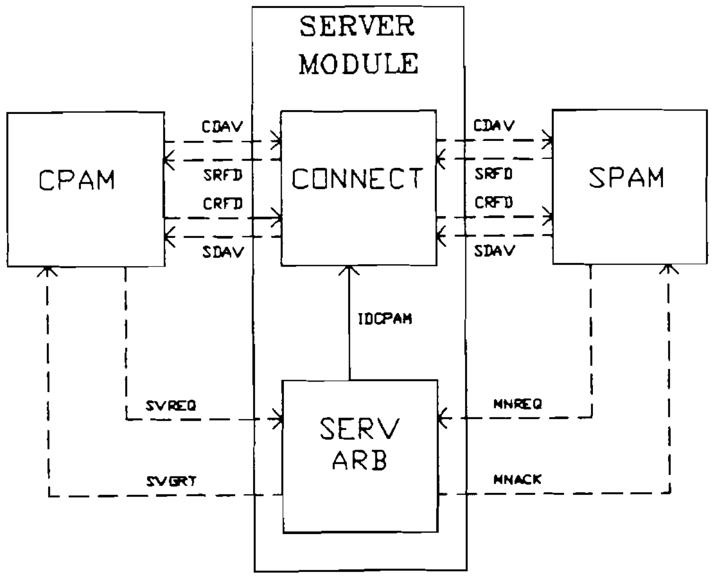


figure 122: connect signals

Normally a resource is locked for the duration of the action. So the caller would remove the service request after receiving the results. But we will try to multiplex several signals, which is impossible, when the request is still active during the process execution. *The difference between resource mutual exclusion and the shared processing is that the shared resource (*SPAM*) knows when the process/action is finished. So no necessity exits to maintain the request/lock.*

After the SVREQ and SVGRT handshake, the **CPAM** wants to pass the parameters to the **SPAM**, which should be ready to receive data. After receiving all parameters, the **SPAM** starts executing the process, while the **CPAM** executes its process until it needs the results. For the result communication we use a handshake protocol. The first active CRFD indicates that the **CPAM** is waiting for the result. It is unnecessary to introduce a separate primitive for the result communication, since the sequencing is known. The caller generates the first CRFD and tests the SDAV. When all results are sent to the caller, the **SPAM** starts a new process.

Sometimes a few phases of this protocol are redundant. After the grant the handshake REQ/GRT protocol can't give any information unless the grant could be missed. If the processing module is not able the react to the grant, because of the execution of a critical section, we need a complete handshake protocol.

The monitor requests a caller using MNREQ and waits for an acknowledge, which is redundant. The monitor needs only a new command. We change the handshake into a start signal signalling the server that a new CMD is necessary. The SPAM prepares itself for the new CMD and signals SRFD and waits until CDAV is received. The new descriptions are:

SPAM MONITOR PSPEC:

```
while true do
 begin
      str:=true; (* start pulse *)
      str:=false;
      init; (* some initialization *)
      get(par1);
      deccmd(procnumb,par1); (* decode cmd *)
      case procnumb do
            1      : proc 1;
            ..       ...
            ..       ...
            end;
 end;
od;
```

The connect unit breaks down the connection after the start signal, before a new SVREQ is generated. Otherwise this signal can be interpreted by the SPAM as a CDAV signalling the SPAM the selection of a partner.

2.2.1.2. THE MULTIPLEXING IMPLEMENTATION

The SVREQ and the SVGRT signals can be removed before the start of the parameter communication (C.1). The sequencing of events allows to implement only two lines connecting every CPAM to the server of the SPAM by two lines. We combine the SVGRT, the SRFD, the SDAV lines into one line (SMES) and the SVREQ, CRFD, CDAV into one line (CMES).

SVAM CTRL SPEC:

```
while true do
   begin
      while not str do skip;
      cobegin
         enable:=true
         mux:=0 (* disconnect *)
      coend;
      while not rdy do skip;(* wait until newcaller is found *)
      cobegin
         enable:=false;
         enable grt lines;
      coend;
      test grt lines (while not(free) do skip;)(* wait for caller to notice grant if needed*)
      mux:=[idcpam];
   end;
od;
```

The server module sets up the connection between the caller and the SPAM (figure 124). The server arbiter generates after a start signal the caller's IDCPAM, which is loaded in to a register after the handshake execution between the server and the caller. This register sets the multiplexer.

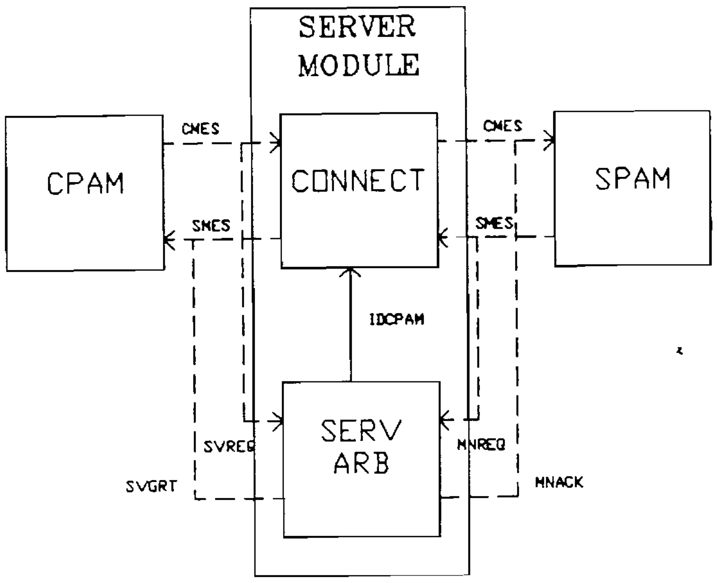


figure 123: multiplexed signals

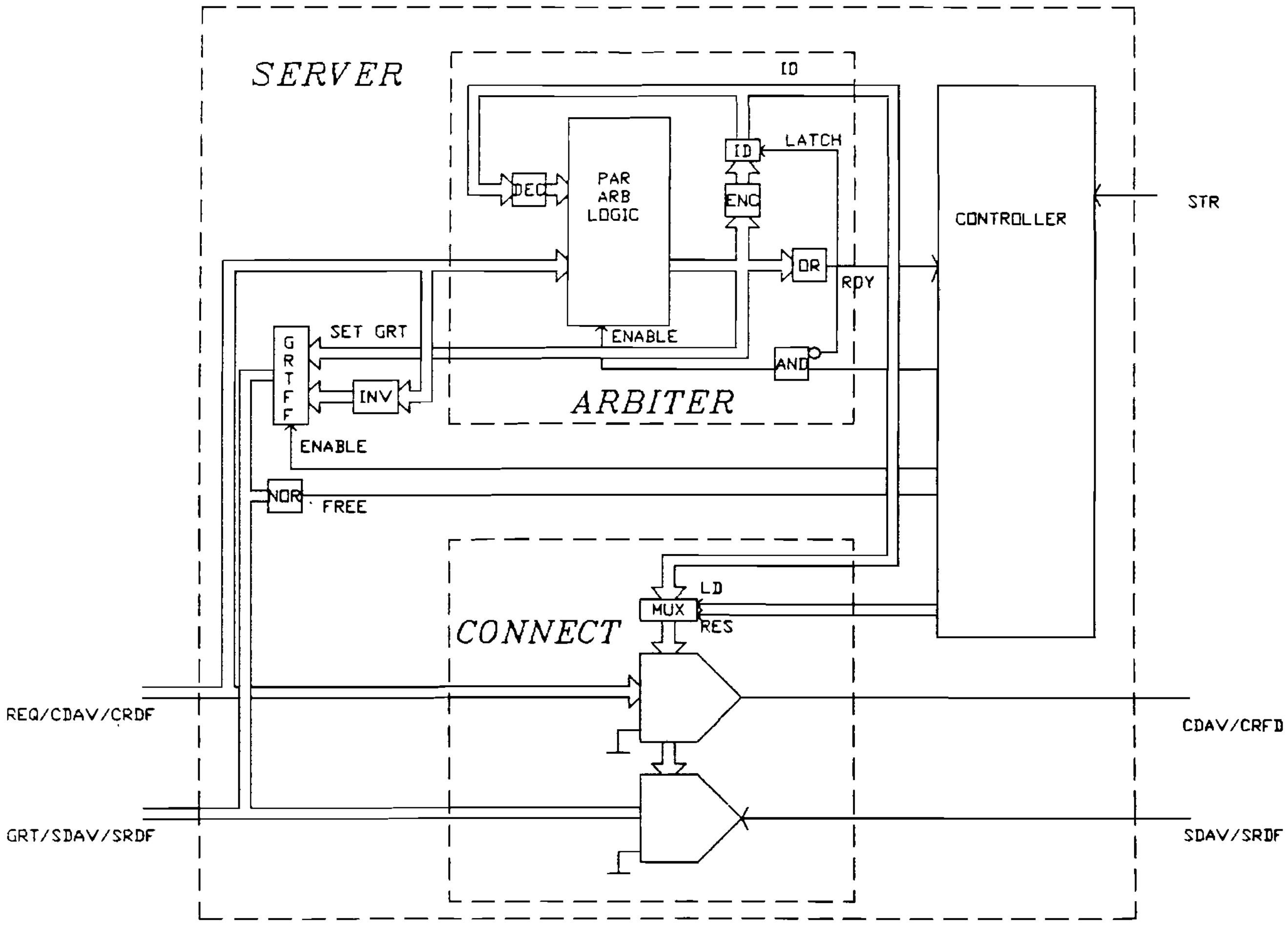


figure 124: server implementation

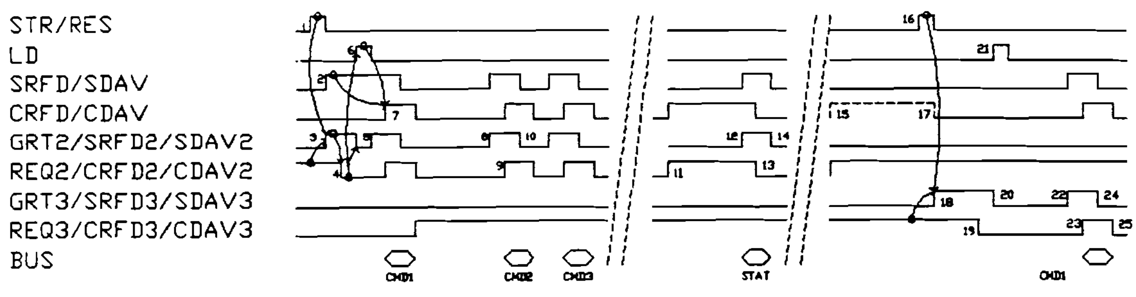


figure 125: cmd timing

	action:
1.	signal to find a new caller and to reset the old connection
2.	the SPAM is ready to receive data
3.	the grant is generated
4.	the SVREQ is removed
5.	the grant is removed
6/7.	the connection is set up
8.	data is put on the bus (parameter)
9.	data is clocked
10.	data is removed
11.	calling process is ready to receive results
12.	status info put on the bus
15.	new request of the calling process, before the connection is broken down. This signal is ignored
16.	disconnected
17.	because of the break down of the connection
18.	another caller is granted
19.	SVREQ removed
20.	granted removed
21.	connected
22.	shared processing module ready for data (the processing module has done some work before generating this signal).

Response time:

1-3	arbitration
1-2	process start-up; We don't know which action will occur first 2 or 3.
3-4	one clockcycle
4-5	one clockcycle
5-6	one clockcycle; this period can be longer, when the handshake between SVREQ and SVGRT includes four phases.
7-8	reaction calling process, the duration is application depended
11-12	this can be awhile.
15-17	signal given to SPAM. Because the SPAM doesn't expect a signal anymore until STR is raised, this signal is ignored.

It seems possible to reduce the handshake to SVREQ, LDMUX and use SRFD as SVREQ, but SVREQ will be seen as CDAV raising conflicts.

2.2.2. several spams

See also discussion in 2.1.2.

2.2.2.1. USING ARBITERS

We remove resource management and introduce an arbiter in the server. Our goal is to reduce the number of lines needed for the communication. Masking the SVREQ after the SVGRT gives an opportunity to multiplex this line, but the server module has to remove the mask requiring the server module to know which **SPAM** is serving which **CPAM**. It's easier to introduce a separate SVREQ line and to multiplex the SVGRT/SDAV/SRFD. Because of the event sequencing the **SPAM** can only generate a signal after giving a grant. Now we need only three lines for the **CPAM**.

Primitives starting an arbiter should not be multiplexed. (SVREQ and MNREQ). The acknowledge signals can be multiplexed, since the receiver is in a state where only one event can occur. We will use the MNACK as load signal for the connect module. The new configuration can be found in figure 126.

```
SERVER CONTROL PSPEC:
begin
    while not((or mnreq)) do skip; (* test if any mnreq; no request of spams to get partner*)
    strl:=true; (* the svreq module fetches a partner for the cpam *)
    while not (rdyl) do skip;
    cobegin
        strl:=false;
        str2:=true; (* fetch the spam partner *)
    coend;
    while not(rdy2) do skip;
end;
```

Informal description of
SERVER ARBITER 1:
- select SVREQ
- SVGRT
- WREMSVREQ
- REMSVGRT
- RDY1

SERVER ARBITER 2:
- select MNREQ
- MNACK
- WREMMNREQ
- REMMNACK (*load IDCPAM *)

See the server arbiter in appendix B.

The **CPAM**'s description isn't changed, but the new description of the **SPAM** monitor is as follows:

```
while true do
    begin
        mnreq(idcpam);
        wmnack(idcpam);
        remmnreq(idcpam);
        init;
        get(parl);
        deccmd(procnumb,parl); (* decode cmd *)
        case procnumb do
            1      : proc 1;
            ..       ...
            ..       ...
            end;
    end;
```

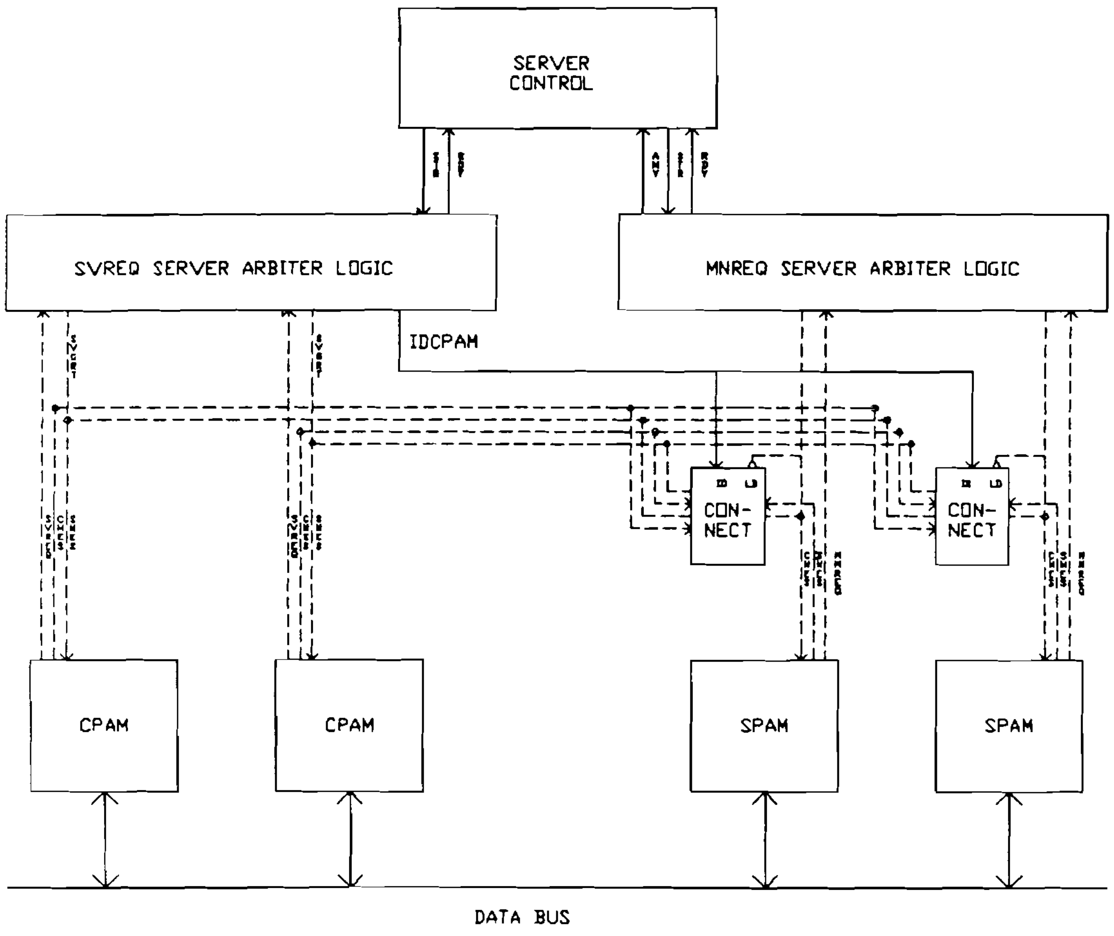


figure 126: AFD/AID no resource management

The IDCPAM can be transported on the bus requiring a data protocol. The function of MNREQ/MNACK changes altering the server control. After the SVREQ removal, the bus should be requested. But the implementation must select the correct MNREQ and MNACK to generate the signals requiring extra hardware, which is probably more expensive then to use two separate RFD and DAV (shared lines).

2.2.2.2. A PARALLEL SERVER MODULE

Suppose it takes awhile before the calling process module acknowledges the service grant. The response time of the calling process depends on its program. During this period the server module has to wait, while in the mean time another SPAM has generated a new MNREQ. We solve this problem by using an interrupt technique. The outline of the server module is:

- when the server module is idle wait for a MNREQ.
- find the IDCPAM of the SPAM.
- save this ID
- start searching for a partner CPAM
- find the IDCPAM
- generate SVGRT
- mask SVREQ of IDCPAM, so that this SVREQ won't be assigned to another SPAM avoiding two SPAMs handling the same SVREQ.

This program is interrupted, when the SVREQ is removed after receiving the SVGRT. The SPAM starts its execution. The server's request removal interrupts the processing. No interrupts can arrive, if no MNREQs are generated. The outline of the interrupt program is:

- find IDCPAM of the interrupt (several interrupts can arrive together)
- remove mask of the corresponding SVREQ
- fetch an IDCPAM of a SPAM
- load MUX[IDSPAM]:=IDINT

The SPAM can be given an IDCPAM, that is different from the CPAM that was given a grant after the MNREQ.

The server's module program outline is:

```
MAIN
begin
    enable int;
    while true do
       begin
          str_spam_arb_serv
          while not (rdy_spam_arb_serv) do skip;
          load_idspam_in_buf;
          str_svreq_arb_serv:;
          while not (rdy_svreq_arb_serv) do skip;
          set_idcpam_in_mask;
       end;
    od;
end;


INT_ROUTINE;
begin
    disable int;
    str_int_arb_serv;(*no wait on rdy required, because at least one signal is pending *)
    remove_idspam_from_buf;
    cobegin
          dec_idspam_in_sel_mux;
          load_idint_in_muxpnt;
    coend
    enable int;
end;
```

This solution shows what it costs to speed up the service of the SPAM. It is not clear, if the overall performance is better, since we introduced a certain amount of bookkeeping. But the costs in the hardware are enormous. A solution for masking the SVREQ can be found in the literature (interrupt's masking). We should be aware of the complexity of the connect module (two level multiplexing; the IDSPAM must select the correct MUX for the SPAM and for the IDCPAM loading).

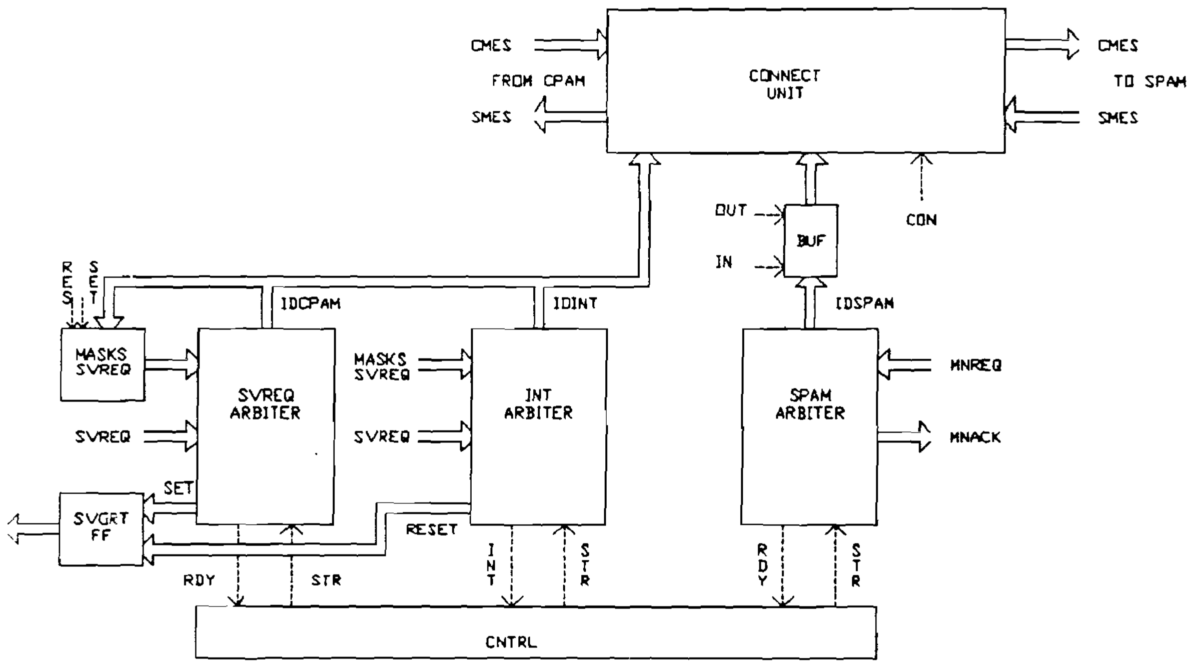


figure 127: parallel server

3. buffering

3.1. RESOURCE MANAGEMENT

3.1.1. process specification

We introduce the following identifiers:

IBUFCMDO	: for the communication between the BUFCMD and a SPAM
IBUFCMDI	: for the communication between the BUFCMD and a CPAM
IBUFRSU	: for the result ready signalling
IBUFRSUI	: for the communication between the BUFRSU and a SPAM
IDCPAM	: identifier for the caller

CALLER PSPEC:

```
cobegin
    begin
        parallel program;
    end
    begin
        waitres(ibufcmdi); (* request access to the bufcmd *)
        putarb(id);
        putarb(cmd1);
        putarb(cmd2);
        .....
        cobegin
            putarb(cmdn);
            relres(ibufcmdi); (* release buffer *)
        coend
        prog
        wsigrdy(ibufrsu); (*wait until the bufrsu signals that the result is ready*)
        get(res1); (* get status of bus *)
        decode(res1,ok); (* decode the status info *)
        if ok
            then
                begin
                    get(res2); (* get next result byte *)
                    get(res3);
                    .....
                end
            else proc errorhandling;
    end;
coend;
```

SHARED PROCESS PSPEC:

```
begin
    init;
    get(par2); (* fetch parameter *)
    get(par3);
    relres(ibufcmdo);(* all parameters fetched release the bufcmd,so that other can fetch
    parameters *)
    execute
    status(res1,ok);(* generate status info *)
    waitres(ibufrsui);
    update(id); (* the number cntpar is replaced by the number cntrsu *)
    putarb(id); (* send idspam of cpam to buffer *)
    putarb(res1); (* put status on the bus *)
    if ok then
            begin
                putarb(res2);
                .....
            end;
    relres(ibufrsui);
end;
```

SPAM MONITOR PSPEC:

```
while true do
   begin
        init; (* some initialization *)
        waitres(ibufcmdo);
        get(id); (* id saved to send back, when ready *)
        get(parl); (* command fetched *)
        deccmd(procnumb,parl); (* decode commands *)
        case procnumb do
                1       : proc 1;
                ..         ...
                ..         ...
                end;
   end;
```

We define two primitives for the data communication with the buffer (see appendix A):

PUTBUF(REG):

```
            WHILE WACK DO SKIP;
            WREQ:=TRUE; (* request the buffer to write the byte into the buffer *)
            WHILE NOT (WACK) DO SKIP;
            WREQ:=FALSE;
```

GETBUF(REG):

```
            WHILE RACK DO SKIP;
            RREQ:=TRUE; (* request the buffer to read a new byte *)
            WHILE NOT(RACK) DO SKIP;
            RREQ:=FALSE;
```

We start with the interface process for the BUFCMD.

THE BUFCMD MODULE

```
PROC_INTF_IN_COMMANDS; (* interface for the in process *)
begin
     while true do
          begin
             get(reg); (* fetch the data of the bus and put it in a register *);
             putbuf(reg);
          end;
     od;
end;
```

```
PROC_INTF_OUT_COMMANDS; (* interface for the out process *)
begin
     while true do
        begin
             getbuf(reg); (* fetch an element in the buffer *)
             putarb(reg); (* put the fetched element on the bus *)
        end;
     od;
end;
```

THE BUFRSU MODULE

```
PROC_INTF_IN_RSU;
begin
     while true do
          begin
             get(reg); (* fetch the data of the bus and put it in a register *);
             putbuf(reg);
          end;
     od;
end;
```

This is identical to INTF_IN_COMMAND. All processes are so simple, that they can be combined with the buffer read and write processes (see appendix A).

```
PROC_INTF_OUT_RSU;
while true do
  begin
    (* fetch an element in the buffer *)
    getbuf(reg);
    cobegin
      load__cnt; (* a part of the idspam is used to tell how many data bytes the result
                block contains *)
      set__sigrsu[idcpam]; (* set the signal result is ready for the correct cpam using
                the idcpam *)
    coend
    cobegin
      begin
          while not(rfdcp) do skip; (* used as acknowledge signal *)
          reset__sigrsu[idcpam]; (* this is no problem, because only one sigrsu is set *)
      end
      while cnt <> 0 do
          begin
            (* fetch an element in the buffer *)
            getbuf(reg);
            (* put the fetched element on the bus *)
            putarb(reg);
            cnt:=cnt-1;
          end;
        od;
    coend;
  end;
od;
```

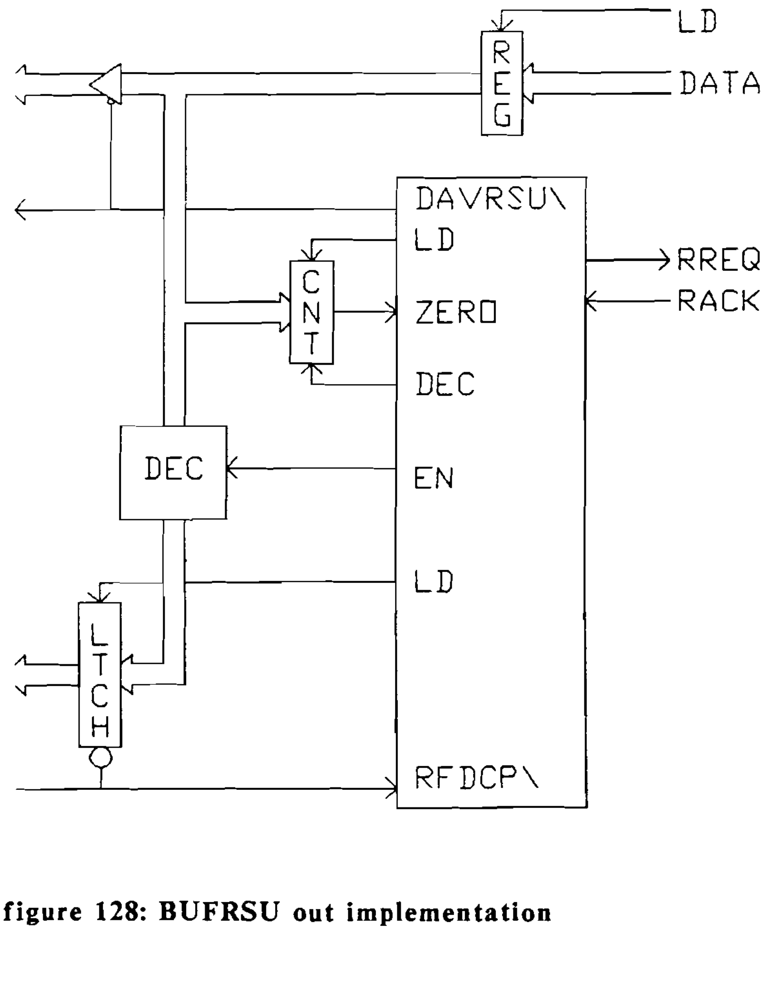
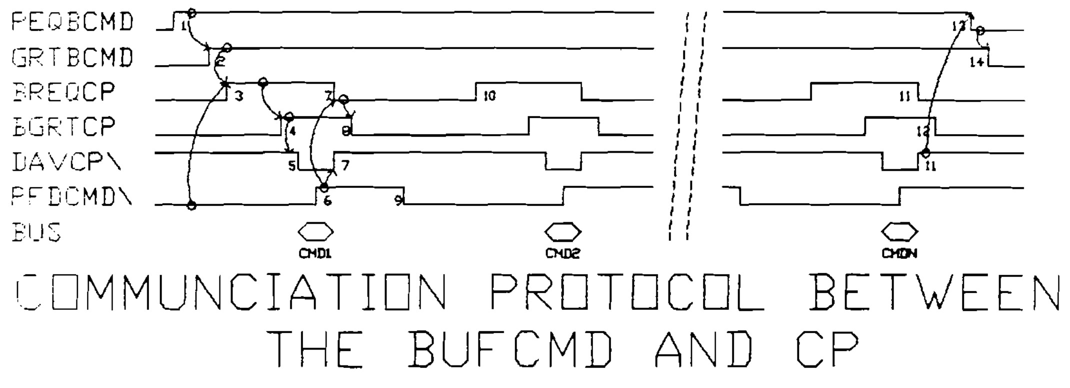


figure 128: BUFRSU out implementation

3.1.2. the communication protocols



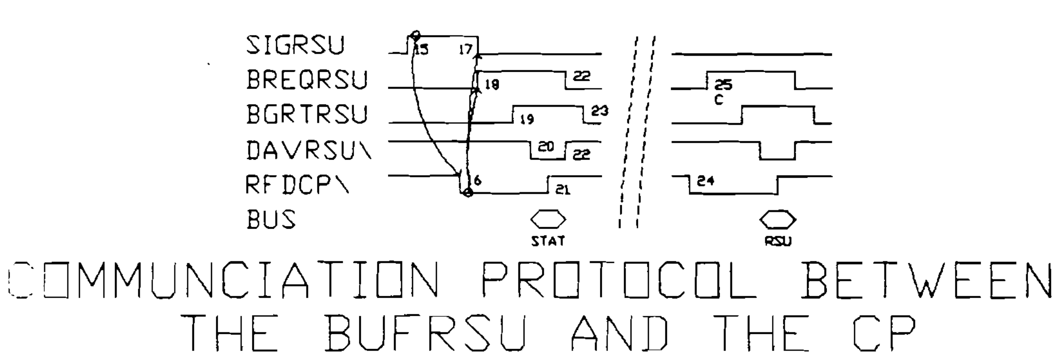


figure 129: the communication protocol with CPAM

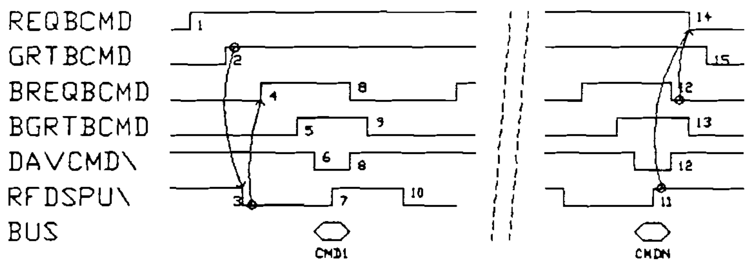
action:

1. request for the shared resource BUFCMD
2. the shared resource is granted to this CPAM
3. the CPAM requests the bus after testing if the BUFCMD can receive the data (RFDCMD). It has data ready otherwise it won't request the bus
4. bus granted
5. data put on the bus
6. data copied
7. data removed from the bus and the bus request removed
8. bus given to another requester
9. the BUFCMD is ready for a new data byte, because the buffer has received data
10. the CPAM has a new data byte and requests the bus
11. the last parameter is removed from the bus and the bus request is removed.
12. bus given to somebody else
13. request for the shared resource BUFCMD removed
14. shared resource given to somebody else
15. result is ready; signal to the CPAM
16. CPAM is ready for the data byte
17. signal that result is ready is removed
18. bus is requested
19. bus is granted
20. the BUFRSU puts the data on the bus
21. the CPAM copies the data
22. the bus request is removed and the data is removed from the bus
23. bus given to another requester
24. the CPAM can receive another byte
25. bus request

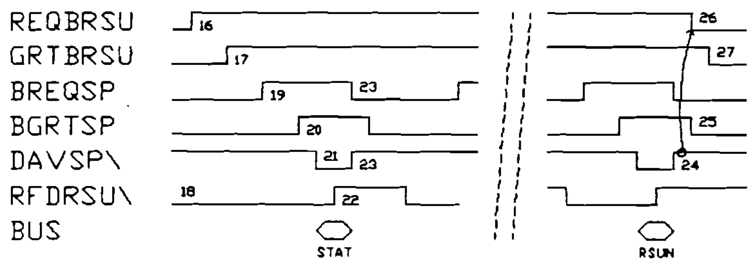
the response time:

1-2	depends on the number of requests. When only one request is depending one clockcycle
2-3	one clockcycle, the CPAM is waiting for this event, when the buffer can receive data
3-4	one clockcycle
4-5	one clockcycle
5-6	one clockcycle, when both module use the same clock
6-7	one clockcycle
7-8	one clockcycle
6-9	depends on the implementation of the buffer (WREQ and WACK) and if the buffer isn't full.
9-10	depends on the fact if the CPAM has the data ready and is ready to put it on the bus. If data is ready then one clockcycle.
11-13	could be zero
13-14	one clockcycle
15-16	depends on what the CPAM is doing. This is one clockcycle, when the CPAM is waiting for the result
16-17	one clockcycle
17-18	one clockcycle
21-24	depends on implementation of CPAM
24-25	depends on buffer. If data available (it can be that the SPAM hasn't put a next data byte in the buffer), this depends on the implementation of the buffer.

Remark: Notice the difference between the RFDCMD\ and RFDCP 16. This is essential for the protocol. 16 can only happen after 15, while RFDCMD\ can be active before the resource is granted.



COMMUNICATION PROTOCOL BETWEEN THE BUFCMD AND THE SP



COMMUNICATION PROTOCOL BETWEEN THE BUFRSU AND THE SP

figure 130 : communication protocol for SPAM action:

1.	request for the shared resource BUFCMD
2.	the shared resource is granted to this SPAM
3.	the SPAM is ready to receive a new data byte
4.	the BUFCMD requests the bus after testing if the SPAM can receive the data (RFDSPAM)
5.	bus granted
6.	data put on the bus
7.	data copied
8.	data removed from the bus and the bus request removed
9.	bus given to another requester
10.	the SPAM wants a new data byte
11.	the last parameter is copied
12.	data removed of the bus and bus request removed
13.	bus given to somebody else
14.	request for the shared resource BUFCMD removed
15.	shared resource given to somebody else
16.	result is ready and the BUFRSU is requested
17.	BUFRSU is granted
18.	the buffer can receive data
19.	bus requested
20.	bus granted
21.	data on bus
22.	that copied
23.	dav and bus request removed
24.	last result byte removed of bus
25.	bus given to another partner
26.	shared resource given free
27.	given to somebody else

the response time:

1-2	depends on the number of requests. When one request is depending one clockcycle
2-3	one clockcycle, the SPAM is waiting for this event
3-4	one clockcycle, if the buffer has data ready. Otherwise it will wait until a data byte is ready.
4-5	one clockcycle
5-6	one clockcycle
6-7	one clockcycle
7-8	one clockcycle
8-9	one clockcycle
7-10	depends on the implementation of the buffer (RREQ and RACK)
17-19	one clockcycle, if the SPAM is ready and the buffer can receive data.

3.2. WITHOUT RESOURCE MANAGEMENT

A scheme of the buffers can be found in figure 131 and 133. We describe the functions of the servers.

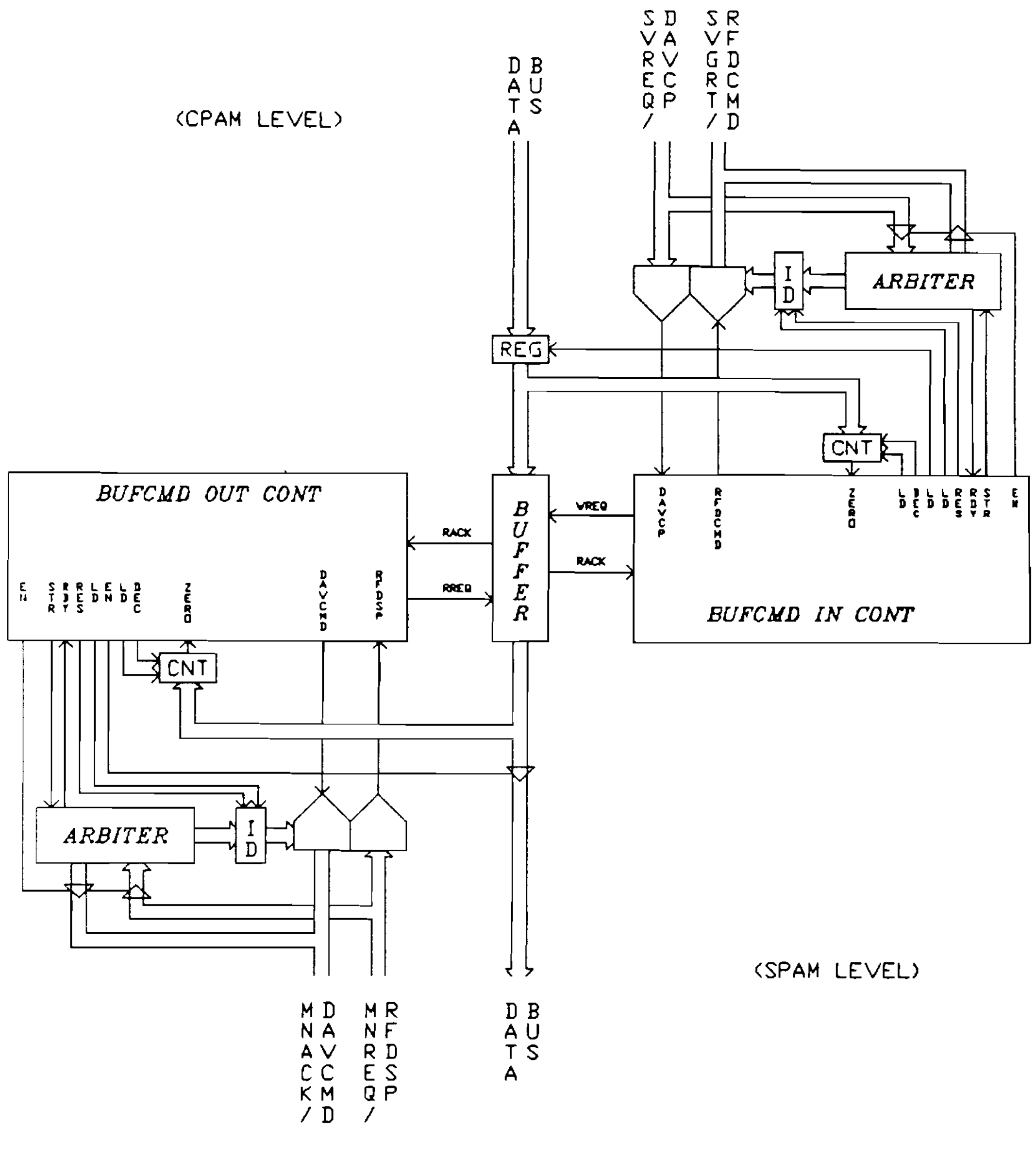
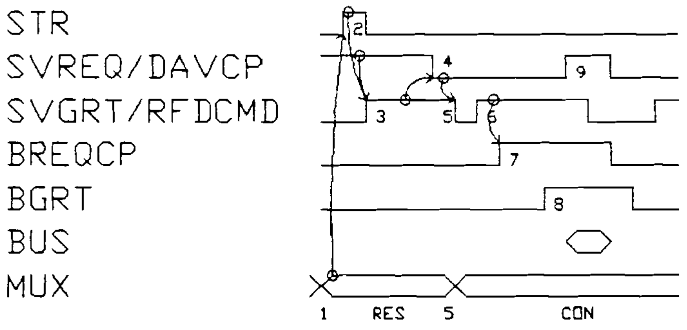


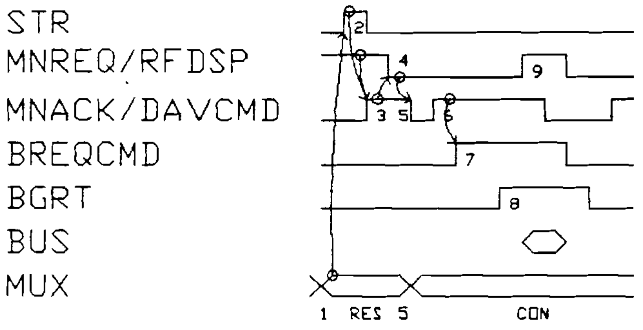
figure 131: bufcmd model

In the protocol (figure 132) a STR signal is introduced, to make the protocol easier to read. In the implementation this signal doesn't exist. We see that during the period 1-5 the handshake for the service request is executed. The duration depends on the parallel program the CPAM is executing, when a SVREQ is pending (period 3-4). Action 5 is a consequence of the server arbiter implementation, but is not functional. The buffer can always fetch the first byte, since the input register is empty, when the program reaches this point. Actions 6,7,8,9 make up the normal data communication handshake protocol.

```
The CMD_IN_SERV;
cobegin
  while true do (* grant logic *) od;
  while true do
    begin
        while not(rdy) do servarb(svreq[1..M],true,idcon,rdy);
        svgrt[idcon]:=true;
        wremsvreq(idcpam); (* while not(free) do skip, so no wrong interpretation is
        possible*)
        cobegin
              engrt:=false;
              mux:=id; (* set up link, now reset mnack *)
        coend
        get(id); (* the byte containing the id of the cpam and the number of bytes
                      that will be transmitted *)
        load__cnt; (* this statement could be executed at the moment the id is clocked,
                      but this requires a special get primitive *)
        putbuf; (*put received byte in buffer *)
        while cnt<>0 do
          begin
              get(reg);
              cobegin
                    putbuf(reg);
                    cnt:=cnt-1;
              coend
          end
          od;
        cobegin
              mux:=0; (* a new request can be served *)
              engrt:=true;(* enable arbiter response logic*)
        coend
    end;
  od;
coend;
```



PROTOCOL DATA COM FROM CP TO BUFCMD



PROTOCOL DATA COM FROM BUFCMD TO SP

figure 132: protocol data communication with bufcmd

```
THE CMD_OUT_SERV;
cobegin
   while true do (* grant logic *) od;
   while true do
       begin
              while not(rdy) do servarb(mnreq[1..M],true,idspam,rdy);
              mnack[idspam]:=true;
              wremmnreq(idspam);(* while not(free) do skip, so no wrong interpretation is
                                   possible*)
              cobegin
                     enack:=false;
                     mux:=idspam;(* set up link, now reset mnack*)
              coend
              getbuf(reg);(* this register is part of the buffer*)
              load__cnt;
              putarb(reg);(* give the id of the cpam to the spam*)
              while cnt<>0 do
                 begin
                     getbuf(reg);
                     cobegin
                            putarb(reg);
                            cnt:=cnt-1;
                     coend
                 end
                 od;
              cobegin
                     mux:=0;
                     enack:=true;
              coend
       end;
    od;
coend;
```

The executed protocol is identical with the protocol for the CPAM and BUFCMD (fig 132). Only the duration of periods will differ. The reset period (the handshake of monitor request and grant) will be shorter.

```
The RSU_IN_SERV;
cobegin
   while true do (* grant logic *) od;
   while true do
       begin
              while not(rdy) do servarb(reqrsu[1..M],true,idspam,rdy);
              grtrsu[idspam]:=true;
              wremreqrsu(idcpam);
              cobegin
                     engrt:=false;
                     mux:=idspam;(* set up link, now reset mnack *)
              coend
              get(id);
              load__cnt; (* could be executed in parallel *)
              putbuf(reg); (*put received byte in buffer *)
              while cnt<>0 do
                 begin
                     get(reg);
                     cobegin
                            putbuf(reg);
                            cnt:=cnt-1;
                     coend
                 end
                 od;
              cobegin
                     mux:=0;
                     engrt:=true;
              coend
       end;
    od;
coend;
```

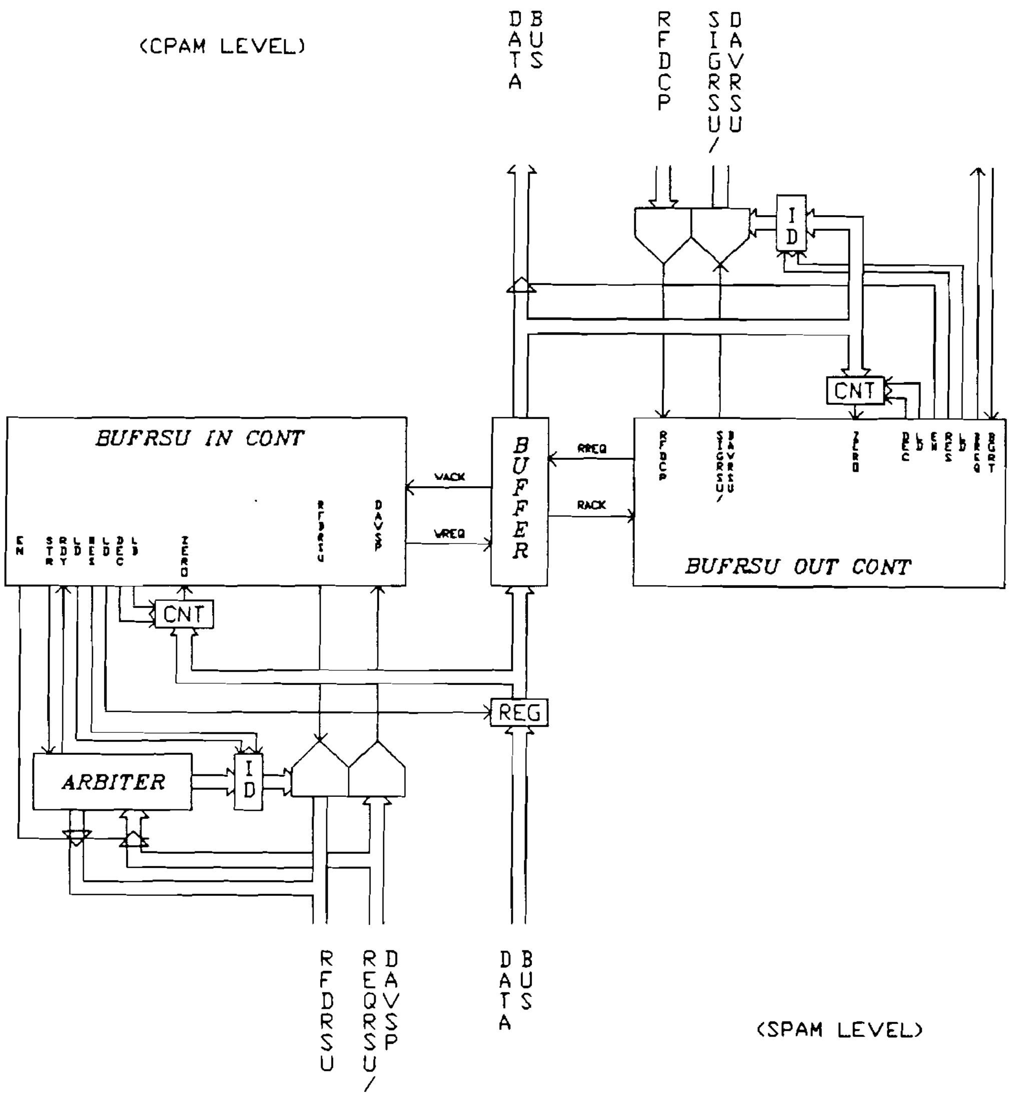


figure 133: bufrsu model

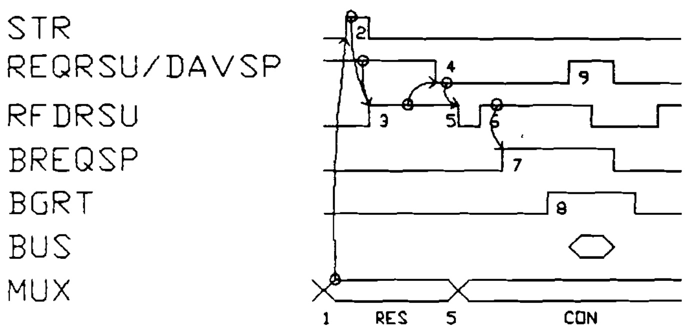


figure 134: protocol data communication between shared process and buffer

The protocol doesn't differ from the other ones. The period 1-5 will be short, because the **SPAM** is waiting for the moment when the buffer is granted.

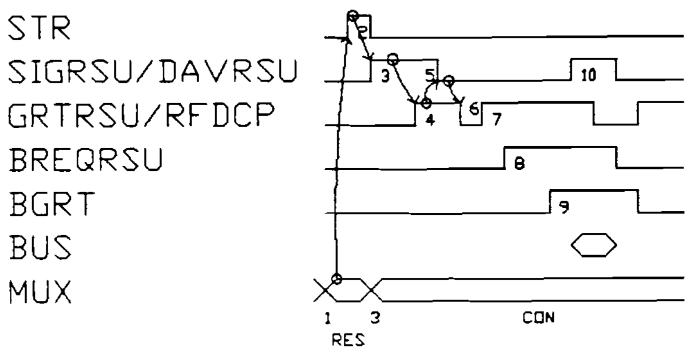


figure 135: protocol data communication between buffer and CPAM

This protocol really differs. The period 1-3 is very short, and could be removed, since no signal will be interpreted wrong. The period 4-5 depends on the program the **CPAM** is executing at the moment it gets the SIGRSU. When the SIGRSU is an interrupt for the **CPAM**, the reaction will be very quick.

```
THE RSU_OUT_SERV;
cobegin
   while true do (* grant logic *) od;
   while true do
       begin
           getbuf(reg);
           cobegin
               load_cnt;
               mux:=id;
           coend
           sigrsu[id]:=true; (* signal the cpam *)
           wackrsu(id); (* wait for the acknowledge, the cpam can be in a critical section
                          when it receives the sigrsu *)
           remsigrsu; (* finish handshake protocol *)
           wremackrsu(id);
           while cnt<>0 do
              begin
                  getbuf(reg);
                  cobegin
                         putarb(reg);
                         cnt:=cnt-1;
                  coend
              end
              od;
           mux:=0;
       end;
    od;
coend;
```

3.3. CONCLUSION

We will compare the two buffer solutions. The number of lines used in server buffer model is:

#CPAM (prot bufcmd) + #CPAM (prot bufrsu) + #CPAM (busreq) + #SPAM (prot bufcmd) + #SPAM (prot bufrsu) + #SPAM (busreq) + 2 busreq bufcmd + 2 busreq bufrsu =

6#CPAM + 6#SPAM + 4

between brackets stands an identification for lines
# CPAM is the number of CPAM
# SPAM is the number of SPAM
prot means protocol

The resource management model:

#CPAM (resource management bufcmd) + #CPAM (sigrsu) + #CPAM (busreq) + #SPAM (resource management bufcmd) +#SPAM(resource management bufrsu) + #SPAM (busreq) + 4 (data prot at CPAM level) + 4 (data prot at SPAM level) + 2 busreq bufcmd + 2 busreq bufrsu =

5#CPAM + 6 #SPAM + 16

It depends on the number of modules to find the minimum. Until twelve CPAMs are used, the buffer servers model uses less lines. We can only conclude that a clear and a simple model using resource management doesn't make much difference.

The logic needed is different. The arbiters are moved from the resource management to the buffer servers. The only difference is the number count in the buffer server model. Another disadvantage is the number of multiplexers.

The resource management model is conceptual easier, but doesn't give a great implementation profit. In the buffer servers model the handshakes are a bit unnatural. All primitives are more logical in the resource management model.

D. PROGRAM STRUCTURES FOR SHARED PROCESSING AFTER TASK ALLOCATION

Using the techniques introduced in chapter, we change the programs for shared processing and focus on the program structure and synchronization. The implementation of the techniques depens on the requirements model. The interrupt handling costs extra resources, so if the requirements model doesn't request it, we don't implement it.

1. hardwired calls

1.1. CALLER PROGRAMS

The interrupt handlers:

```
INT_SVGRT; (* the SVGRANT is the interrupt *)
begin
    remsvreq
end;
```

Purpose: avoid waiting until a spam is free.

```
INT_PAR; (* the REQPA is the interrupt *)
begin
    waitres(bus);
    reg1:=par1;
    reg2:=par2;
    reg3:=par3;
    .....
    cobegin
        reqn:=parn;
        relres(bus);
        rdypa;          (* signal parameters transmitted *);
    coend;
    sigtask(idtask2);
end;
```

Purpose: increase flexibility in cpam. The parameter request folows quickly after the SVGRT.

```
INT_RES; (* interrupt SIGRSU *)
begin
    waitres(bus);
    rrsu1:=rsu1;
    ......
    cobegin
        rrsum:=rsum;
        relres(bus);
        rdyrsy; (* signal result copied *)
    coend;
    sigtask(idtask3); (* make task waiting for the result runnable *)
end;
```

Purpose: avoid waiting for result
The implementation of the interrupts changes the tasks into:

The tasks:

```
TASK 1;
begin
      prog
      svreq(idcpam);(* request shared process *)
      save(proc__stat1);
end;

TASK 2;
begin
      remtask(idtask2);
      restore(proc__stat1);
      prog
      save(proc__stat2);
end;

TASK 3;
begin
      remtask(idtask3);
      restore(proc__stat2);
      prog
      (* further statements depend on task function *)
end;
```

Remark: Task 2 can be executed in task 1, if the resources where the parameters reside aren't used. The monitor doesn't change.

1.2. SPAM PROGRAMS

1.2.1. one spam

The structure of the called shared process (not including the monitor) doesn't change, because we don't allow the SPAM to execute another task, when it is running a requested process. slowing down other programs. The monitor doesn't change unless we can schedule processes, which aren't requested by other modules. Every executed requested process signals the server with a MNREQ that a new caller can be served. Because, the module can be running another task, we introduce an interrupt routine:

```
INT_SERV; (* interrupt MNACK *)
begin
      load__spid:
      remnnreq(idspam);
      case spid do
            id1    : sigtask(id1)
            id2    : sigtask(id2)
            ..   .....
            end;
end;
```

Purpose: increase flexibility, execute another task when no call is pending.

```
MONITOR SPAM;
while true do
     begin
          dih;
          dispatch(proid, shedinfo,taskinfo);
          eih;
          case procid do
               id1     : task id1
               .......
              end
     end
```

```
PROCESS SPID;
begin
  cobegin
     init; (* starts some initialization independent of the parameters *);
     begin
       reqpa;
       wrdypa;
       init2; (*initialization depending on the parameters *)
     end
    coend;
    processing
    cobegin
     exit1; (* statements independent of the result communication to leave the process*)
     begin
       sigrsu;
       wrdyrsy;
       exit2; (* exit statements, that can only be executed after the result is fetched. *)
     end;
  coend;
  mnreq(idspam); (* request new process to execute *)
end;
```

1.2.2. several spam

1.2.2.1. RESOURCE MANAGEMENT

The requested process resets the connection and requests a new caller. The grant for access of the server will activate an interrupt routine. We allow the mutual exclusion resource primitive to be interrupted using REQRESI and RESRESI. This is implemented in an interrupt routine to avoid busy-waiting, when no calls of a process is pending and the module can execute other tasks.

```
INT_SERV1; (* grant for server *)
begin
     mnreq(idspam); (* request a caller *)
end;

INT_SERV2: (* MNACK interrupt, it can take a while before it occurs,
             when no call is pending *)
begin
     remnnreq(idspam);
     load__conid;
     load__spid:
     relresi(sa);
     case spid do
          id1    : sigtask(id1);
          id2    : sigtask(id2);
          .. .....
          end;
end;
```

The monitor is identical with the one for one spam.
The requested process:

```
PROCESS SPID;
begin
   cobegin
      init;
      begin
         reqpa;
         wrdypa;
         init2;
      end
   coend;
   processing
   cobegin
      exitl;
      begin
         sigrsu;
         wrdyrsy;
         exit2;
      end;
   coend;
   res__conid; (* problems can arise if this connection isn't reset, when another spam
                  serves this conid, while no new svreq can be assigned to this spam.*)
   reqresi(sa); (* request server to get a new caller *)
end;
```

1.2.2.2. RESOURCE MANAGEMENT AND MULTIPLEXING
We define a new primitive GETNOI. This primitive gets data but can't be interrupted. We will use it also in interrupt routines, which can't be interrupted any way. But the name stresses the function.

```
GETNOI (DEST) :=
   BEGIN
     DIH;
     WHILE DAV\ DO SKIP;
     RFD\:=TRUE;
     WHILE NOT DAV\ DO SKIP;
     COBEGIN
          LD[DEST]:=TRUE; (* load destination register *)
          RFD\:=FALSE;
     COEND
     EIH;
  END;
```

The requested process id and the conid are transported on the databus.

```
INT_SERV2: (* MNACK  is the interrupt *)
begin
      remnnreq(idspam);
      getnoi(spid);
      getnoi(conid);
      relresi(sa);
      case spid do
            idl    : sigtask(idl);
            id2    : sigtask(id2);
            ..   .....
            end;
end;
```

Purpose: avoid waiting when no call is pending or the server is busy.

1.2.2.3. NO RESOURCE MANAGEMENT
The primitives RES_CONID and MNREQ(IDSPAM) are introduced at the end of the requested process. The MNACK is an interrupt, which actives:

```
INT_SERV;
begin
      getnoi(spid);
      getnoi(conid);
      remnnreq(idspam);
      case spid do
            id1    : sigtask(id1);
            id2    : sigtask(id2);
            ..     .....
      end;
end;
```

2. command direct connection

2.1. CALLER PROGRAMS

We will use PUTNOI, which is the put primitive executed with the interrupt handling disabled. We introduce a result signalling to implement an interrupt.

```
INT_GRT; (* the SVGRT is the interrupt *)
begin
      remsvreq(idspam);
      putnoi(par1);
      ....
      putnoi(parn);
      sigtask(idtask2);
end;

INT_RSU; (* interrupt is SIGRSU signal *)
begin
      ackrsu(idcpam); (* handshake protocol *)
      wremsigrsu(idcpam);
      remackrsu(idcpam);
      getnoi(res1);
      decode(res1,ok);
      if ok
         then
            begin
               getnoi(res2);
               ....
               getnoi(resm);
               sigtask(idtask2);
            end
         else sigtask(iderr);
end;
```

Because every grant is assigned to one interrupt line, the number of result bytes for the specific interrupt routine is known. If we wouldn't do this, we have to introduced tables to keep track of the destination of the data and which tasks has requested the process. The tables have to be used for task signalling.

```
TASK 1;
begin
        prog
        svreq(idcpam);(* request shared process idsc *)
        save(proc__stat1);
end;

TASK 2;
begin
        remtask(idtask2);
        restore(proc__stat1);
        prog
        save(proc__stat2);
end;

TASK 3;
begin
        remtask(idtask3);
        restore(proc__stat2);
        prog
        (* statements depend on task structure *)
end;
```

Remark: Task 2 can be executed in task 1, if the resources where the parameters reside aren't used.

2.2. SPAM PROGRAMS

2.2.1. one spam

```
CALLED TASK;
begin
        init
        execute
        status(res1,ok);(* generate status info *)
        sigrsu(idcpam);
        wackrsu(idcpam);
        wremack(idcpam);
        remsigrsu(idcpam);
        dih; (* to speed up the communication, the task may not be interrupted. Otherwise
                    putnoi is sufficient *)
        put(res1); (* put status on the bus *)
        if ok then
                begin
                    put(res2)
                    .....
                end;
        eih;
        mnreq(id);
end;

INT_ACK; (* the MNACK is the interrupt *)
begin
        remmnreq(idspam);
        getnoi(par1);
        deccmd(procnumb, #par); (*decode cmd *)
        while #par>0 do
                begin
                    getnoi(par[#par]);
                    #par:=#par-1;
                end
        case procnumb do
                1       : sigtask(id1);
                ..          ...
                ..          ...
                end;
end;
```

Purpose: avoid waiting when no call is pending.

2.2.2. several spams

The MNREQ line is RFD and the MNACK line is DAV

```
INT_SERV1; (* SVGRANT is the interrupt *)
begin
      rfd:=true;
      while not (dav) do skip;
      cobegin
            rfd:=false;
            relresi(sa);
            connect; (* load id *)
      coend
end;
```

Purpose: avoid waiting when no call is pending or server is busy.

The interrupt routine for fetching the command and the parameters was already described. The request process executes at the end a REQRESI(SA) and a DISCONNECT primitive.

3. buffers and resource management

3.1. CALLER PROGRAMS

The parameter phase can be implemented as an interrupt routine to avoid waiting or introduce more flexibility.

```
TASK 1:
begin
      waitres(ibufcmdi); (* request access to the bufcmd *)
      putarb(id);
      putarb(cmd1);
      putarb(cmd2);
      .....
      cobegin
            putarb(cmdn);
            relres(ibufcmdi); (* release buffer *)
      coend
      prog
end;

INT_RDY; (*interrupt is SIGRDY *)
begin
      getnoi(res1); (* get status of bus *)
      proc decode(res1,ok,#par); (*decode the status info *)
      if ok
         then
            begin
              while #par>0 do
                  begin
                        getnoi(par[#par]);
                        #par:=#par-1;
                  end
              od;
              sigtask(id...);
            end;
         else sigtask(iderr);
end;
```

3.2. SPAM PROGRAMS

The parameter communication is an interrupt routine to avoid busy-waiting, when no calls are pending. The spam program changes into:

```
INT_BUFFER_PAR; (* interrupt is access grant for buffer *)
begin
        getnoi(id); (*id saved to send back, when ready *)
        getnoi(par1); (* command fetched *)
        proc deccmd(procnumb, #par); (*decode cmd *)
        while #par>0 do
              begin
                    getnoi(par[#par]);
                    #par:=#par-1;
              end
        relresi(ibufcmdo);(* all parameters fetched release the bufcmd,so that other can
        fetch parameters *)
        case procnumb do
              1       : sigtask(id1);
              ..         ...
              ..         ...
              end;
end;
```

Purpose: avoid waiting when buffer is in use or no request is pending.

```
CALLED TASK
begin
        init
        execute
        status(res1,ok); (* generate status info *)
        waitres(ibufrsui);
        putarb(id); (* send id of cpam to buffer *)
        putarb(res1); (* put status on the bus *)
        if ok then
              begin
                    putarb(res2)
                    .....
              end;
        relres(ibufrsui);
        reqresi(ibufcmdo);
end;
```

It makes no sense to put the result communication in an interrupt routine, because we won't gain much flexibility.

```
MONITOR SPAM;
while true do
      begin
            dih;
            dispatch(proid, shedinfo,taskinfo);
            eih;
            case procid do
                  id1     : task id1
                  .......
              end
      end
```

4. buffer without resource management

See remarks buffer with resource management. The mutual exclusion primitives for the buffer are replaced by handshake signals.

4.1. CALLER PROGRAMS

```
TASK;
begin
        prog;
        svreq(idcpam);
        prog;
        save(proc__statl);
end;

TASK 2;
begin
        remtask(idtask2);
        restore(proc__statl);
        prog
        (* further statements depend on task function *)
end;

INT_GRT; (* the interrupt is the SVGRT *)
begin
        remsvreq(idcpam);
        wremsvgrt(idcpam); (* necessary to reset logic *)
        putarb(id); (* put the id and the number count on bus*)
        putarb(parl); (* put command on bus *)
        putarb(par2); (* put parameter on bus *)
                .......
        sigtask(id...)
end;
```

This last routine can be implemented in TASK 1, since we don't have to wait long for the access of the buffer. The routine just gives more flexibility.

```
INT_RES; (* the interrupt is the SIGRSU signal *)
begin
        ackrsu(idcpam); (* this necessary when a parallel program is executed, which can't
                be interrupted. And also to remove the sigrsu signal, which can be seen as a
                dav, so that the get primitive is never  executed *)
        wremsigrsu(idcpam);
        remackrsu(idcpam);
        getnoi(resl); (* get status of bus *)
        decode(resl,ok); (*decode the status info *)
        if ok
                then
                  begin
                        getnoi(res2); (* get next result byte *)
                        getnoi(res3);
                        ....
                        Sigtask(idtask2);
                  end
                else sigtask(iderr);
end;
```

4.2. SPAM PROGRAMS

```
INT_PAR; (* interrupt is the MNACK signal *)
begin
          remmnreq(idspam);
          getnoi(id);
          getnoi(parl);
          deccmd(procnumb,#par);
          while #par>0 do
                    begin
                              getnoi(par[#par]);
                              #par:=#par-1;
                    end
          case procnumb do
                    1         : sigtask(idl);
                    ..               ...
                    ..               ...
                    end;
end;

CALLED PROCESS;
begin
          init
          execute
          cobegin
                    status(resl,ok);(* generate status info *)
                    coded(id); (* generate id and number cnt *)
          coend
          wgrtrsu(idspam); (* request the bufrsu by raising reqrsu and wait for the grant
                                   rfdcp*)
          remreqrsu(idspam); (* remove request; implemented in hardware *)
          wremgrtrsu(idspam); (* this is not necessary, because signal will be raised again to
                                   indicate that the buffer is ready after the lines are set up *)
          putarb(idspam)
          putarb(resl); (* put status on the bus *)
          if ok then
                    begin
                              putarb(res2)
                              .....
                    end;
          mnreq(idspam);
end;
```

The communication with the result buffer is not implemented as an interrupt routine. This is not necessary, while the spam will be served quickly. Implementing an interrupt routine gives more flexibility, but requires more resources.

E. EXAMPLE TRANSCATION CENTER

We will discuss to illustrate the techniques introduced in chapter the control transformation transaction center. Usually all the processes will be allocated to the same processing module.

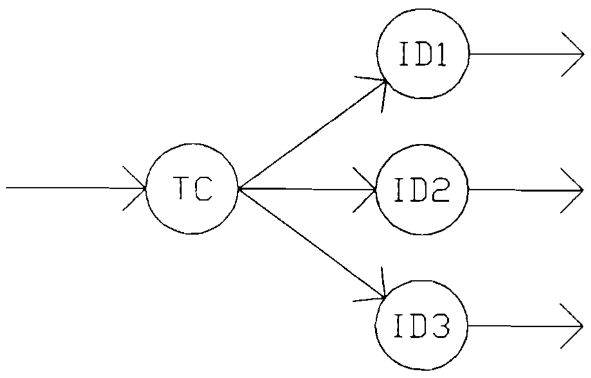


figure 136: transaction center

Functional description:

```
TRANSACTION CENTER;
begin
        testval:=func(input); (*depending on input and internal info a task will be selected*)
        case testval do
            1      : task1;
            .....
          end;
end;
```

we introduce synchronization:

```
PROC TC
begin
        remreqproc(idtc);
        wremgrtproc(idtc);
        testval:=func(input);
        case testval do
            1      : sigtask(id1);
            2      : sigtask(id2);
            3      : sigtask(id3);
          end;
end;
```

We choose to move the REQPROC primitive to the signalled task to avoid that an event will be missed. If we introduce a counting semaphore for the process scheduling we can move the REQPROC to the end of task TC.

```
TASK 1
begin
        remtask(id1);
        prog
        reqproc(idtc);
end;

TASK 2
begin
        remtask(id2);
        prog
        reqproc(idtc);
end;
```

Suppose that task 2 is allocated to another architecture module.

```
PROC TC
begin
        remreqproc(idtc);
        wremgrtproc(idtc);
        testval:=func(input);
        case testval do
                1       : sigtask(id1);
                2       : sigproc(id2);
                3       : sigtask(id3);
          end;
end;
```

```
TASK 2
begin
        remreqproc(idtask2);
        wremgrtproc(idtask2);
        prog
        cobegin
                reqproc(idtc);
                reqproc(idtask2);
        coend;
end;
```

If the scheduling condition is simple (only signalling), we can signal the event directly to the architecture module using R/A synchronization.

In the last example we will start two processes on one condition.

```
PROC TC
begin
        remreqproc(idtc);
        wremgrtproc(idtc);
        testval:=func(input);
        case testval do
                1       : sigtask(id1);
                2       : cobegin sigproc(id2a);sigproc(id2b); coend
                3       : sigtask(id3);
          end;
end;
```

```
TASK 2A                                     TASK 2B
begin                                       begin
        remreqproc(id2a);                           remreqproc(id2b);
        wremgrtproc(id2a);                          wremgrtproc(id2b);
        prog                                        prog
        cobegin                                     cobegin
                reqproc(idtc);                              reqproc(idtc);
                reqproc(idtask2a);                          reqproc(idtask2b);
        coend;                                      coend;
end;                                        end;
```

The schedule module has to check that both processes are executed before it can start process TC again. More flexibility is introduced using a counter semaphore for the scheduling.

REFERENCES

[Ancea,1986] Ancea, F
(1986)
The architecture of microprocessors
Addison-Wesley

[Andre,1985] Andre, Herman D and Verjus J.P.
(1985)
Synchronization of parallel programs
North Oxford Academic

[Brinch Hansen,1975] Brinch Hansen P.
(1975)
The programming language concurrent pascal
IEEE Trans. Software Eng. 1,2 juni 1975
pp 199-207

[Brinch Hansen,1977] Brinch Hansen P.
(1977)
The architecture of concurrent programs
Prentice Hall

[Davio,1983] Davio M, Deschamps J.P. and Thayse A. (1983)
Digital systems with algorithm implementation
Wiley & Sons

[DeMarco,1978] DeMarco, Tom
(1978)
Structured analysis and system specification
Yourdon Press

[Dykstra,1982] Dykstra E.W.
(1982)
A personal summary of the Gries-Owicki theory
Selected writings on computing: a personal perspective
Springer N.Y.

[Dykstra] A tutorial on the split binary semaphore
internal paper TUE EWD 703

[Dykstra] The superfluity of the general semaphore
internal paper TUE EWD 734

[Gouda,1984] Gouda G. Mohammed and Yu, Yoa-Tin
Synthesis of communicating finite state machines with guaranteed progress
IEEE trans on comm C32 N7 jul 1984

[Hatley,1987] Hatley, Derek J. and Pirbhai, Imtiaz A.
(1987)
Strategies for real-time system specification
Dorset House Publishing

[Haynes,1981]	Haynes, Alan B Stored State Asynchronous Sequential Circuits IEEE trans on comp C30 N8 aug 1981
[Hoare,1976]	Hoare, C.A.R. (1976) Monitors: an operating system structuring concept Comm ACM 17 549-557, Oct 1974
[Hoare,1978]	Hoare, C.A.R. (1978) Communicating Sequential Processes IEEE Comm of ACM V21, aug 1978 pp 666-677
[Janson,1985]	Janson, Phillipe A. (1985) Operating systems structures and mechanism Academic press pp 1-86
[Joseph,1984]	Joseph, Mathai, Prasnad V.R. and Natarajan N. (1984) A multiprocessor operating system Prentice Hall
[Kylstra,1984]	Kylstra F.L. Proces computer systemen TUE
[Mead,1980]	Mead C. and Conway L. (1980) Introduction to VLSI systems Addison Wesley chapter 7
[Raynal,1986]	Raynal, Michel (1986) Algorithms for mutual exclusion North Oxford Academic
[Stevens,1987]	Digital System Group TUE Structured VLSI Design Course Eindhoven, University of Technology
[Ward,1985]	Ward Paul T. and Mellor, Stephen J. (1985) Structured development for real-time systems volume 1: introduction and tools volume 2: essential modeling techniques volume 3: implementation modeling techniques Yourdon Press

[Yourdon, 1978]	Yourdon Ed, Constantine L (1978) Structured design Yourdon Press