

---

<b>9. CHAPTER 9: COMPLEX MULTIPROCESSOR ARCHITECTURES WITH HIERARCHY .....</b>	<b>9-2</b>
<b>9.1 THE APPLICATION DOMAIN: MULTI WINDOW DISPLAY FOR TV .....</b>	<b>9-2</b>
9.1.1 VIDEO SIGNALS .....	9-2
9.1.2 EXAMPLES OF FUNCTIONS .....	9-3
9.1.3 CHARACTERISTICS AND FORMULATION OF THE PROBLEM .....	9-4
<b>9.2 TOP LEVEL ARCHITECTURE.....</b>	<b>9-5</b>
<b>9.3 ARCHITECTURE FOR THE PERIODIC SUB SYSTEM .....</b>	<b>9-8</b>
9.3.1 RESOURCE SHARING AND DEADLOCK .....	9-10
9.3.2 PROCESSOR MODEL .....	9-13
9.3.3 COMMUNICATION NETWORK.....	9-14
<b>9.4 CONCLUSION.....</b>	<b>9-16</b>

## 9. Chapter 9: Complex Multiprocessor architectures with hierarchy

In the previous chapter we discussed a relatively simple system based on only one communication principle. Most systems however, are much more complex since they contain several layers of hierarchy that use various communication principles. However, this also leads to a greater variety of possible architectures. There is no such thing as a predominant architecture that can be applied in all circumstances. The choices that are made are heavily depending on the target application domain. But it goes further. There is even no such thing as a specific strategy, built up from procedures (= relatively simple logical steps that operate on a database) through the help of which architectures can be designed [Rechtin]. On the contrary, it is a process that is based on heuristics. And in which experience plays an important role. The aim of this chapter is to illustrate these heuristics using an example.

This chapter has been written from the point of view of the system architect. The role of the architect is intermediate. It is his task to develop an architecture that meets the demands of the client and at the same time can be made by the design team. The architect *cooperates with* the designers *for* the client. A first important step is to understand what the client really wants. This may seem trivial, but this is often not the case because the client is not able to formulate exactly what he wants or because he may have presumptions about the solution and anticipates on that in his formulation. That is why first the application domain has to be discussed, the characteristics of it are analyzed to come to a formulation of the problem next. Then an architecture is developed.

### 9.1 The application domain: multi window display for TV

For the application domain a choice has been made for a multi window display function, i.e. several windows as is common at a PC but now for video signals. This is a very challenging application, which requires a very high computation power. It is important to understand the characteristics of the application domain well at the start of the design. That is why we will first discuss video signals and then will give some examples of display functions.

#### 9.1.1 Video signals

There are video signals in many varieties. Usually the original components R, G, B are converted to a brightness signal Y (luminance) and two color difference signals U and V (chrominance). The band width of the Y signal amounts to 5 MHz, of the U and V components 1.3 MHz. Lines are scanned horizontally. This usually is done in an interlace way. In this the image has been built up from two fields, which contain the even or the uneven lines of the image respectively. Fields are scanned from the top downwards. There are not always the same numbers of Y, U and V samples. Often people work with a proportion of Y:U:V that is equal to 4:1:1, sometimes to 4:2:2 or 4:4:4. The standard sampling frequency amounts to 16 MHz.

### 9.1.2 Examples of functions

In the application a number of tasks with widely various constraints has to be executed. For a start there are a number of *control tasks*. Examples are user interaction (e.g. switching of channel), a modem function, conditional access, generation of menu-graphics, selection of a teletext page etc...A second group is formed by the signal processing tasks with *soft real-time demands*. An example is the display of a teletext image. This has to be done in a reasonable amount of time, but not immediately.

Furthermore, there are the real *video tasks with hard real-time demands*. Video output images have to be produced with a certain speed (e.g. every 10 msec for 100 Hz). Doing so a number of functions have to be executed. A first function is vertical and horizontal sample rate conversion (SRC). This is necessary for the scaling of images, for the zooming and for the change of aspect ratio (e.g. from 4:3 to 16:9). SRC could be done with the use of an interpolation in the time domain but this is not insufficiently exact. Hence, digital filters are used, e.g. 6 tap filters.

Furthermore, a number of operations are needed to improve the quality of the image, for instance noise reduction.

Another task is 100 Hz conversion. In this the image speed has to be doubled. A first approach is to simply repeat the previous image but this leads to a jerky effect in the case of moving objects. Therefore the movement that is present in the image has to be taken into account, which will lead to calculation intensive operations that moreover require a lot of memory band width.

An example of such an application is shown in Figure 9-1. This shows a flow graph in which the above mentioned tasks represent the basic operations. There are two video input streams and a teletext image.

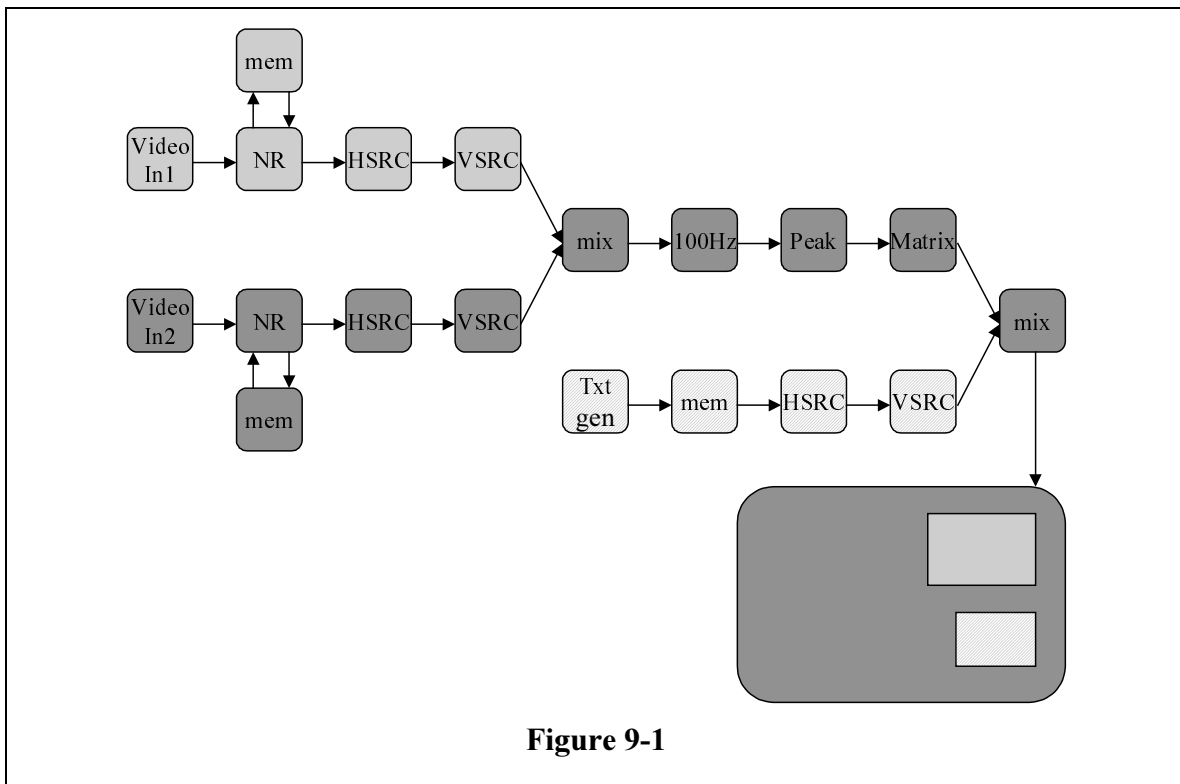


Figure 9-1

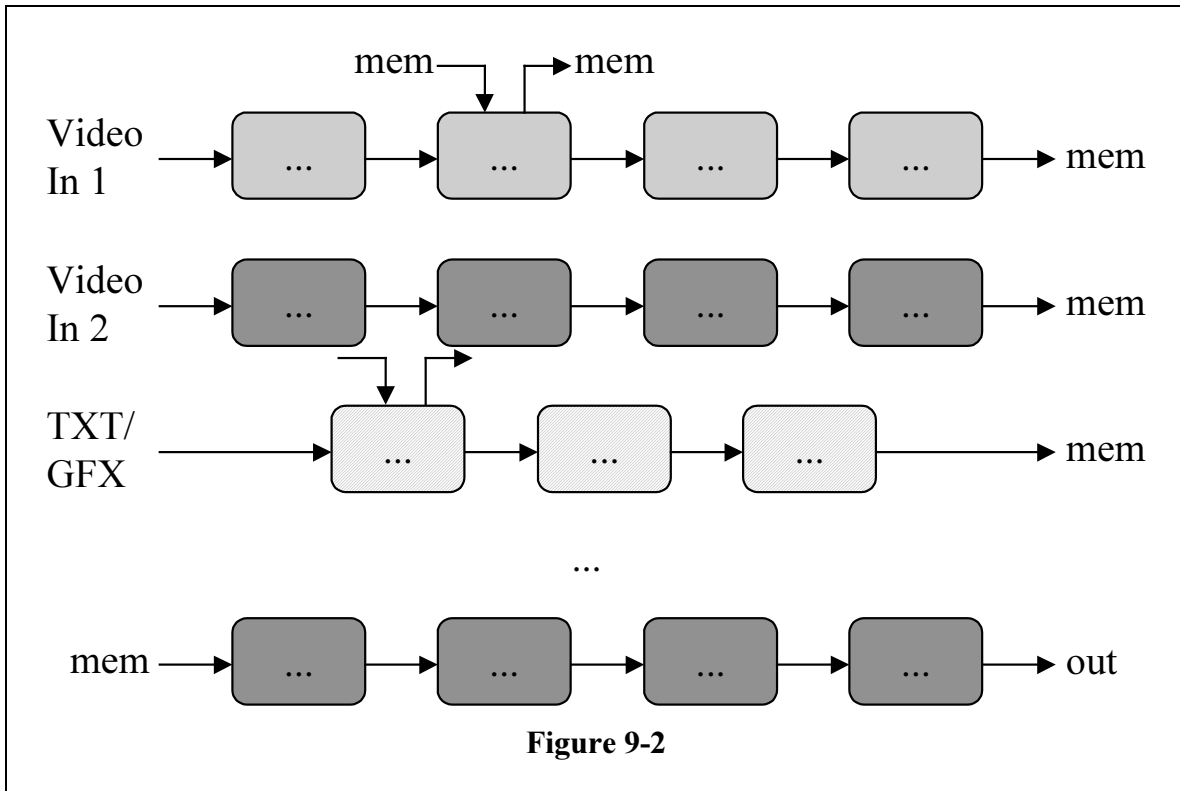
### 9.1.3 Characteristics and formulation of the problem

When we have charted all functions we can make an estimation of the necessary calculation power. This may be done on the base of block schemes of, for instance, the filters or on the base of a first rough specification. In fact we count the number of operations per input or output sample and multiply this with the signal rate. When we do so, we find some tens of Giga operations/sec. Even more important than the calculation power is the necessary band width for communication between the functions. Here we find some tens of Giga bits per second or some GB per second. From this we already are able to conclude that a solution with one single bus won't suffice.

But there are more characteristics. For it appears that though the content of the functions is well known, the order is not always the same. In total there are a hundred graphs and Figure 9-1 only shows one example. In some graphs one and the same functions occurs more than once. This leads to a trade-off with the quality. Take for instance a graph in which only one video stream has to be processed (so no picture in picture). In these circumstances all functions, such as noise reduction, sample rate conversion etc. only occur once and the client wants to switch to highest possible quality mode, e.g. 100 Hz instead of 50 Hz. When the opposite occurs, so like in Figure 9-1, one can switch to a lower quality but one has to process several streams though.

In fact this happens over a relatively limited area, namely 4 streams at a maximum. This leads to a somewhat more abstract definition of the design problem. To make this clear one has to realize that video streams occur in multiples of a standard frequency. In this example this is 16 MHz (sampling pixel rate) and multiples of 16, namely 32, 48 and 64. A factor 2 stems from the 50 to 100 Hz conversion, another factor two stems from high definition instead of standard definition.

Subsequently it has to be noted that some functions (such as temporary noise reduction) need access to the previous image. *So, when we assume that the present technology does not yet allow integration of complete image memories* we can cut the graph at these places or hence create an extra input and output. Other places are the mix-nodes where streams are put merged. In this way we come step for step to a more abstract formulation of the problem that is supported by Figure 9-2.



This figure shows a number of parallel sub graphs with the real data input or the external memory as inputs and the real data outputs (display) and the memory outputs as outputs. Hence, the nodes represent the above mentioned functions in which the following has to be met

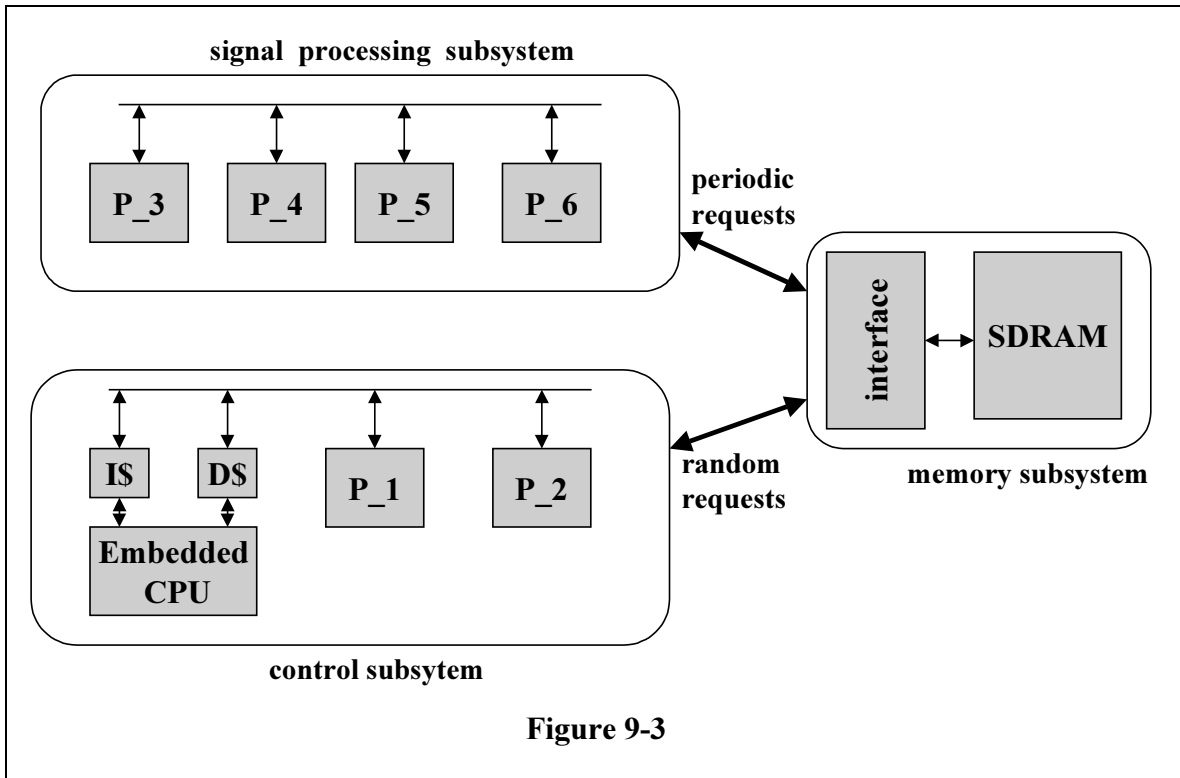
$$n_{16} * 4 + n_{32} * 2 + n_{48} * 3 + n_{64} * 4 \leq 4$$

in which  $n_{16}$  represents the number of 16 MHz streams,  $n_{32}$  the number of 32 MHz streams etc.. A last important note is that streams may be *asynchronous*. Two input signals stemming from two different senders have not been synchronized. This means for PIP pictures that there is always a main image at which is synchronized and to which the second image has to adapt.

For sake of completeness we repeat that the video streams have hard real-time demands. Hence, guaranties have to be given that under all circumstances enough calculation power is available. So this part of the application has to be designed in worst case mode.

## 9.2 Top level architecture

Because of the widely varying nature of the various tasks it is decided to create a separate sub system for the processing of video streams. This leads to an architecture as is shown in Figure 9-3. In this 3 sub systems are present.



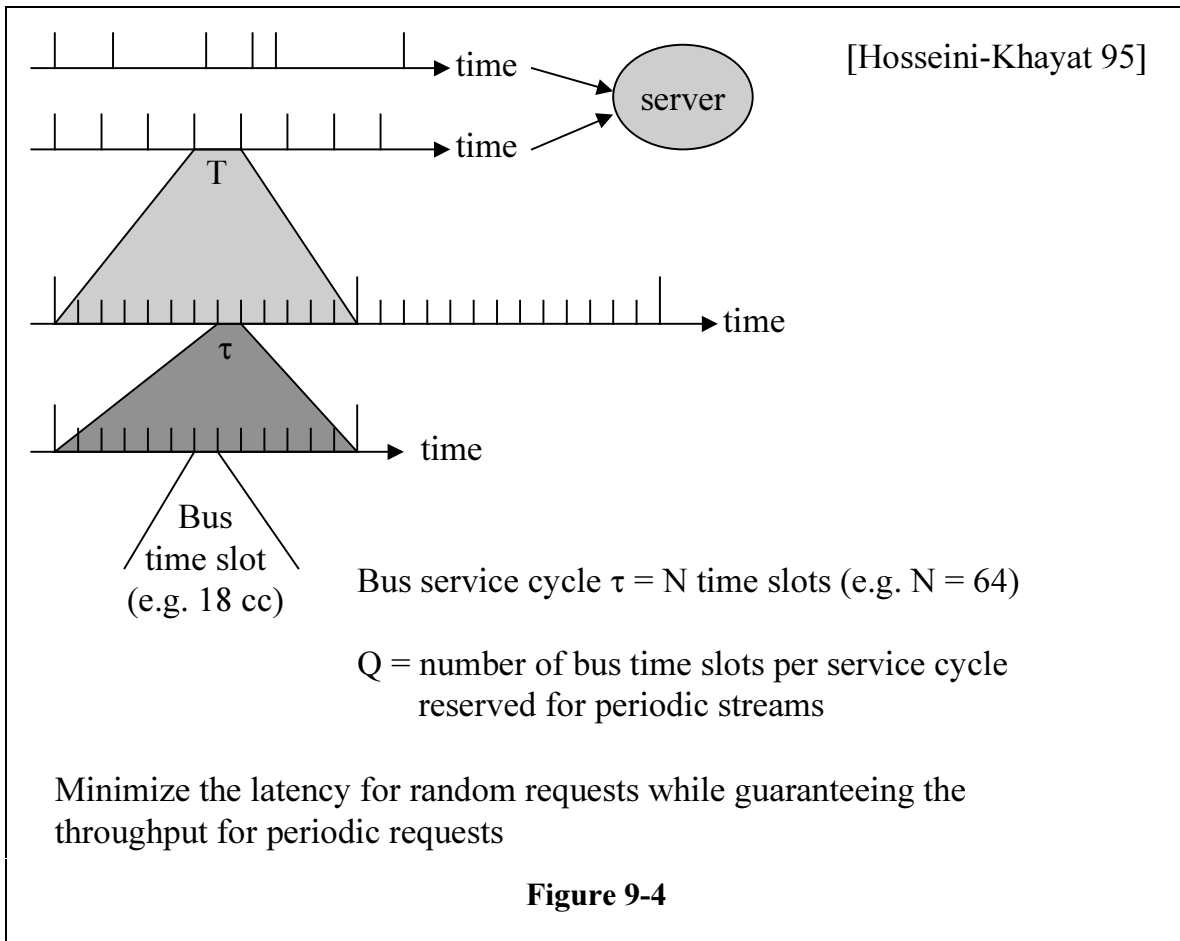
For a start there is a classical bus oriented sub system as has been discussed in the previous chapter unto which the control tasks are mapped. In it there is a programmable CPU, which is responsible for the top-level control. On the bus there may also hand other processors for specific tasks, e.g. a DSP that takes care of modem processing.

Next there is the stream oriented sub system in which all video processing takes place with hard real-time demands. Here also, there are several autonomic processors that are each optimized for a specific video function as has been discussed above. Furthermore, these processors have a number of separate communication tools within the sub system. They can exchange data without going via the external memory. This is necessary in view of the required band width of a few Gbytes per sec. Unto this sub system then the graphs of Figure 9-2 are mapped. The interface of this sub system is therefore specified as well by the inputs and outputs of these graphs.

Next, there is a third sub system, namely the memory system where the previous two subsystems come together. This consists of the external SDRAM and an internal arbiter that has to arbitrate between requests coming from the two earlier mentioned sub systems. It is important to note that the request may be of a totally dissimilar nature. The video sub system generates periodic requests. In this the throughput is important, not the latency. With sufficient buffering a variation in the latency can be taken care of while the throughput remains to be guaranteed. The requests coming from the CPU sub system have the exact reverse characteristics. For they are characterized by a random nature. For example, look at cache misses in the CPU. It cannot be predicted when these will occur.

On top of that the latency is the only thing that counts and the throughput is not relevant. The purpose of the arbiter is to reconcile the opposing characteristics.

This problem may be solved as follows [Hosseini-Khayat]. Consider two sources, one with random requests and one with periodic requests. Each request is stored in a random queue or a periodic queue respectively. Furthermore, there is a server that handles all requests. We call  $T$  the period of the periodic requests. The solution lies in splitting up this period into so-called *service cycles* ( $\tau$ ). Each service cycle in its turn consists of  $N$  *time slots*. Each time slot corresponds with an atomic access action on the memory as has been defined in the previous chapter. Each time slot therefore equals 16..18 clock cycles.



Every service cycle the server runs through the following algorithm. In this we use the following symbols:

- $N =$  number of time slots
- $Q =$  the number of time slots per service cycle needed for the periodic requests
- $n =$  remaining number of time slots (initially =  $N$ )
- $q =$  remaining number of time slots available for periodic requests (initially =  $Q$ )

The algorithm then is as follows.

1. When  $n > q$  random requests prevail. When there are no random requests periodic requests are handled.
2. When  $n \leq q$  the periodic requests prevail. When there are no periodic requests random requests are handled.
3. Decrement  $q$  when a periodic request is handled.
4. Decrement  $n$ . When  $n=0$ , then make  $n=N$  and  $q=Q$  and start again.

In this way the average delay of random requests, when periodic requests are present, determines the delay of random requests without the presence of periodic requests. In this it is assumed that no *overload* situations will occur in which the total number of requests becomes larger than  $N$ . In the Prophid project a choice has been made for  $N=64$ . The above mentioned can be easily extended when there are more periodic requests.

### 9.3 Architecture for the periodic sub system

Since the video functions are well known, but are to have the possibility to be knotted together in a flexible way, we come in a natural way to an architecture in which the functions are mapped onto a separate autonomous processors that can communicate with each other via a flexible reconfigurable network. The network is based on multiplexers from which the position is controlled from a program. The level of functions in this case is a natural level of hierarchy.

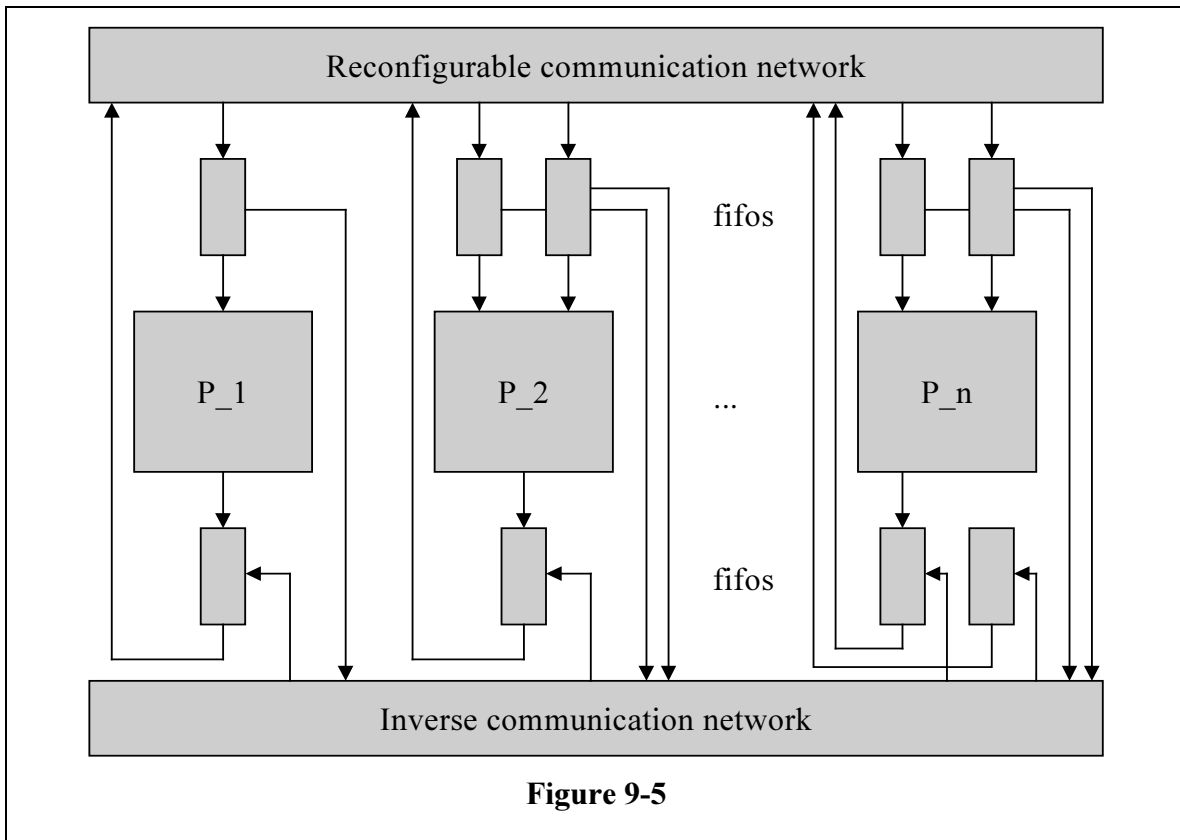


Figure 9-5

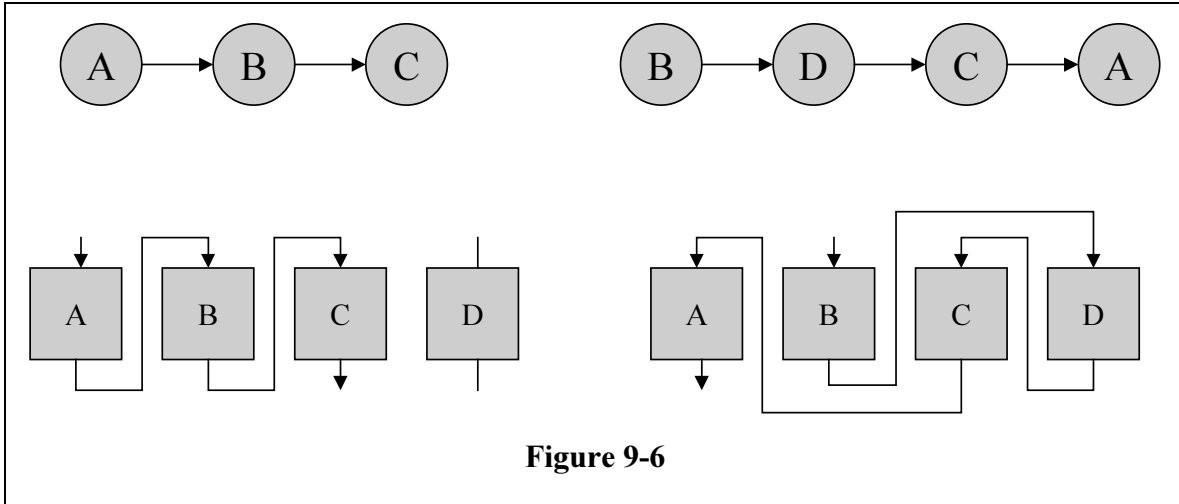
All processors are surrounded by *buffers* at the inputs and the outputs. The purpose of these buffers is to separate the processing from the communication. When these buffers would not be present, this would mean that the processors can only generate outputs at the moment that the transports can be taken care of immediately. When the communication network would not be available for a moment this would mean that the processor blocks. It is important to note that video operations are highly communication intensive. Most of the processors generate an output every stroke of the clock or every second stroke of the clock. (This is different with e.g. Audio processing where a relatively low amount of data can cause much processing.) We will return to the dimensioning of the buffer size and therefore the degree of disconnection later.

The next point is that we choose for *FIFO buffers*. The reason for this becomes clear when we look at the specification. The specification is an SFG in which the nodes are functions. Between the functions signals are transmitted. The internal signals represent measurable entities that, in principle, can be shown at a display. The only identification is the order of the samples in the stream. That is why the communication channels show a FIFO behavior.

Concerning this point there are two options: synchronous data-flow (SDF) or dynamic data-flow (DDF). In the case of SDF the complete schedule is determined at compile time. This means that a program is made at compile time that controls the communication network (as with a VLIW). A choice has been made for DDF, however, for the following reasons.

1. We want to use the field blanking in a useful way for soft real-time tasks. We would have to adapt the program constantly and have to switch between various modes (stream 1 in the blanking, yet stream 2 not etc...). Beside that, this leads to a fundamental problem with asynchronous input streams, since we cannot know at compile time when the blanking will occur.
2. Some functions such as a VLD produce a data dependent number of output tokens. In other words, they are in fact DDF functions.
3. DDF is often more simple because the local availability of the data determines the activity. With SDF with large pipelines there is often a long switching on phenomenon, which leads to a costly central controller.
4. The concept is easier to extend. (adding of processors)

This leads to a *blocking semantics* of the channels according to a DDF model. The processor is stopped when one of the output buffers is complete or when one of the input buffers is empty. That is why in Figure 9-5 an *inverse network* is added. In that way it becomes possible to execute various graphs.

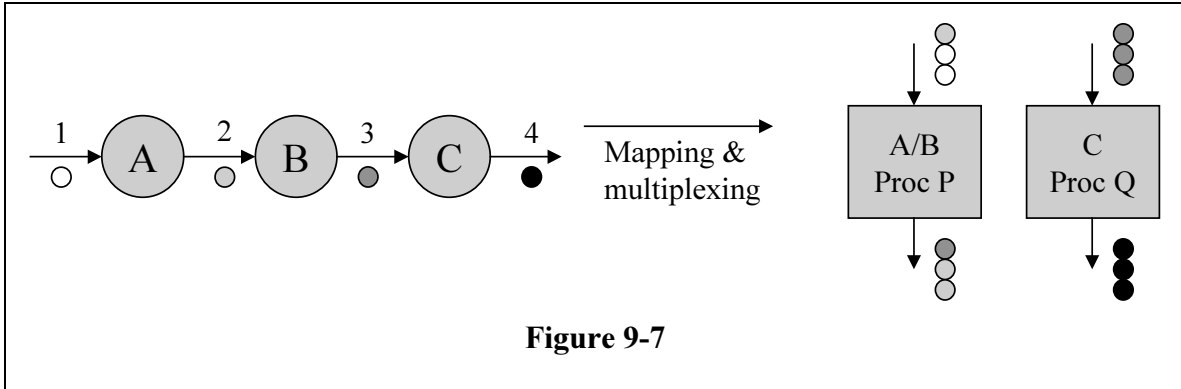


### 9.3.1 Resource sharing and deadlock

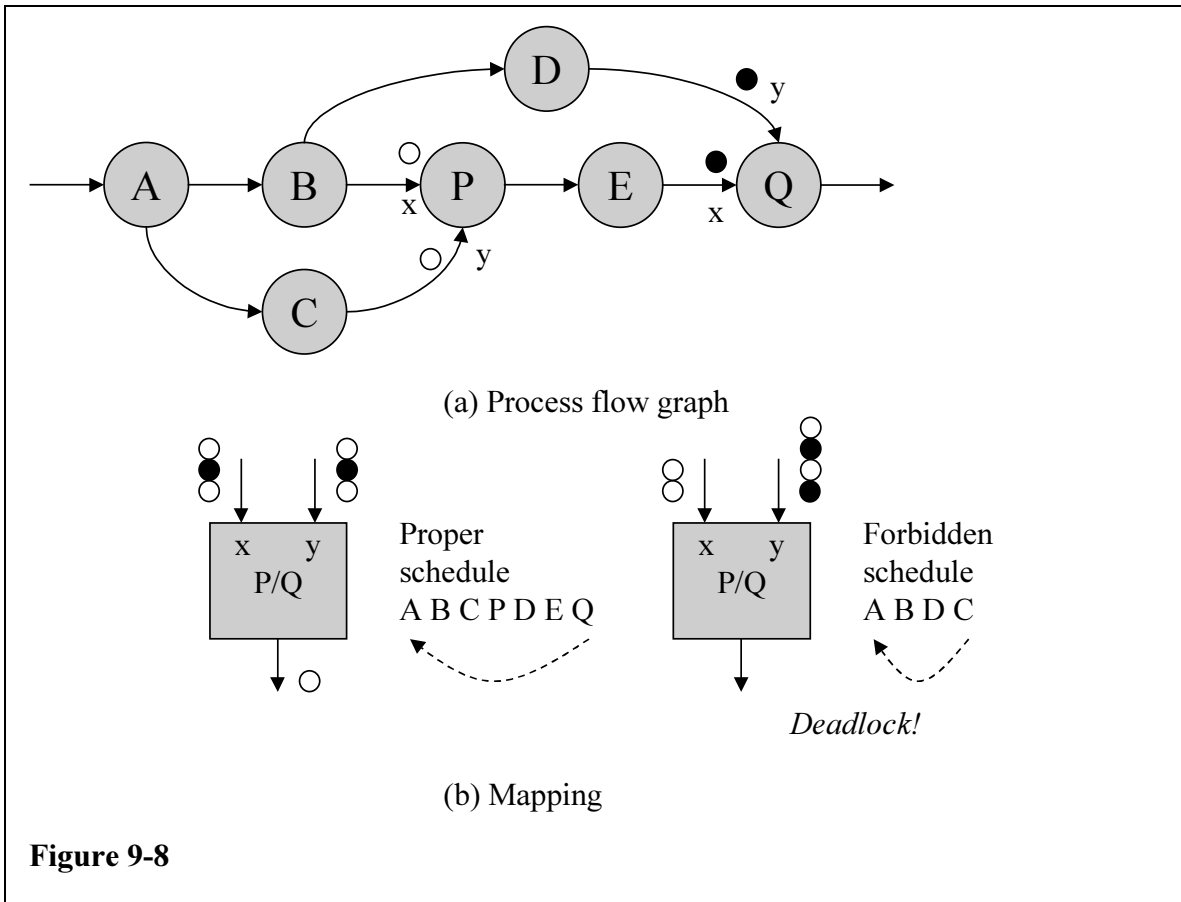
The above-mentioned architecture works fine with one important restriction however. Problems may occur with resource sharing. Resource sharing is necessary because otherwise we would each time need an equal amount of processors as the number of times that a specific function occurs in the graph. Or we have to execute all these function one after another in the time.

A distinction can be made between various forms of resource sharing. For a start, it is possible that a processor executes more than one task Furthermore, we can start to share the fifo buffers or the hardware in the interconnection networks.

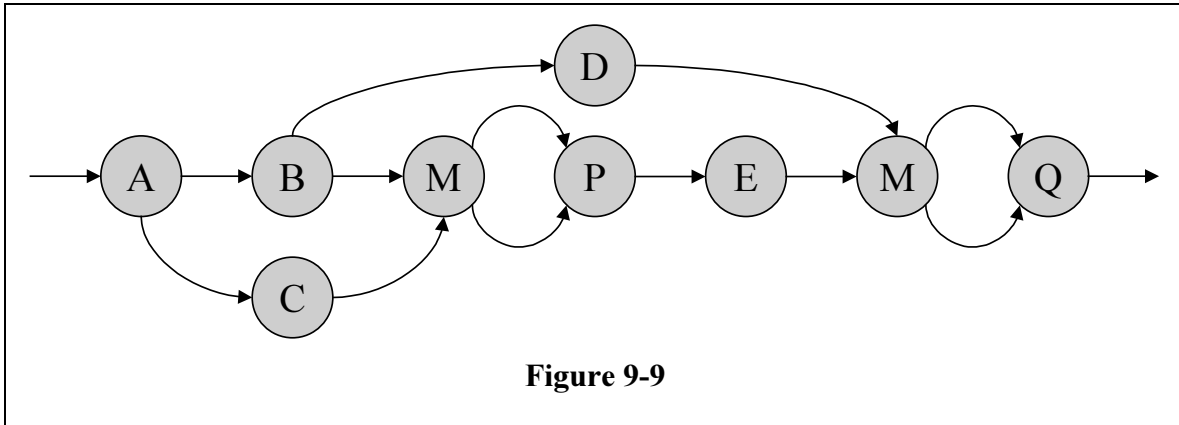
We start with sharing of processors. It is an obvious method to map all functions of the same type unto the same processor. In Figure 9-2 a function such as HSRC can occur more often in several graphs that have to be executed parallel. It is an obvious method to design only one HSRC processor. This processor will have to be sufficiently flexible to process a number of 1 to 4 streams. This will be discussed in detail later. At this moment it is important that the processor should be able to make a choice using the incoming data which function has to be executed. That is why each data package belonging to a task is preceded by an identification in the form of a header or a tag. In the figures this is indicated by the different coloring of the tokens. An example is shown in Figure 9-7. Here, 3 tasks are shown (A, B and C) that are mapped unto 2 processors (P and Q) from which processor P executes two tasks. Processor P produces two types of output tokens according to whether the data has to be sent to P again for the execution of process B or whether it should be sent to processor Q for the execution of process C. A unique identification of the data can be obtained by providing every edge in the specification with its own token.



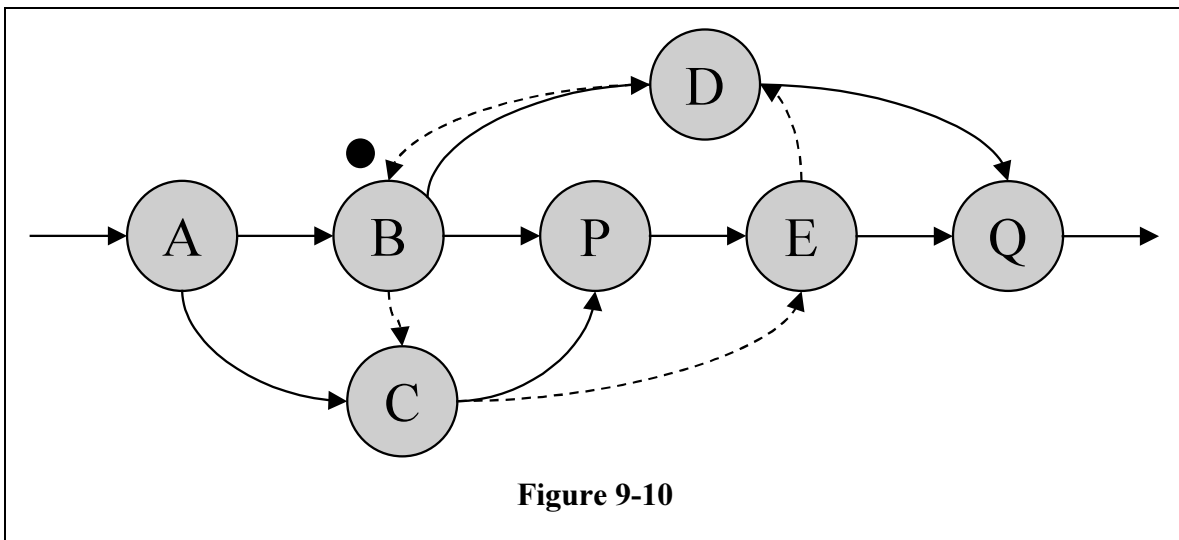
However some caution should be called for. It is possible that *deadlock* will occur. This means that there is data present in the system but yet no progression in the calculations can be made. For this purpose we have to make the graph more complex and also provide processors with several inputs. An example is the graph in Figure 9-8, which contain the processes P and Q. Both processors have two inputs and we assume that they are executed on the same processor. Processor P needs white tokens, processor Q needs black ones. The order in which the black and the white tokens arrive on the x and y inputs of the processor depends on the order in which the tasks are scheduled. Figure 9-8 shows a situation in which this accidentally goes well and one in which this goes wrong. We call this situation *deadlock*.



There are a number of methods to avoid deadlock. A first possibility is formed by algorithmic transformations. A second possibility is synchronization with the use of extra memory processes. The flow graph in the example can be transformed by adding extra memory processes (processes M as are shown in the next figure). The memory processes collect the tokens stemming from the processes B, C, D and E and send pairs of tokens with the same color to the received processes P, and Q respectively.

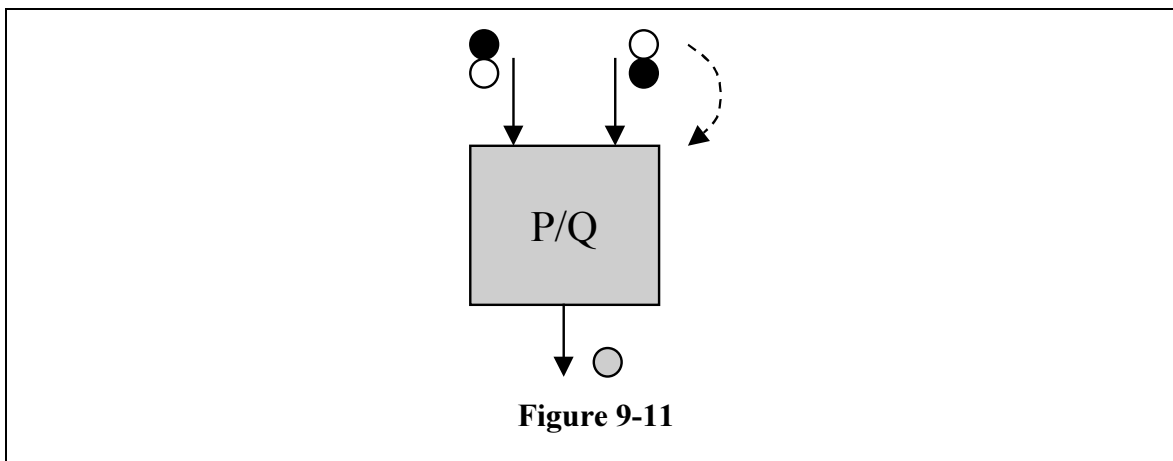


A third method is to add extra sequence arrows in the graph. The tokens on these arrows that are indicated in a dotted line do not represent data. A node can only fire when not only the data branches contain tokens but the sequence branches as well. In this way an extra ranging can be applied to the nodes. An example is shown in the next figure where extra ranging is applied between the nodes B, C, D and E. [also see Sriram, JVSP, March 97]



A fourth possibility consists in the so-called *out of order* consuming of input tokens. Instead of implementing the input channels as fifos they are implemented as random access memories. When unequal tokens appear at the inputs of the processor that executes P as well as Q the right tokens can be gathered again. This presumes that a random access

memory is present where a minimum space is always reserved for each process that is possibly executed on the processor. The number of processes that is executed parallel hence should be known beforehand (so on compile time). This is equivalent to forbidding the merging of the fifos in the multiprocessor model.



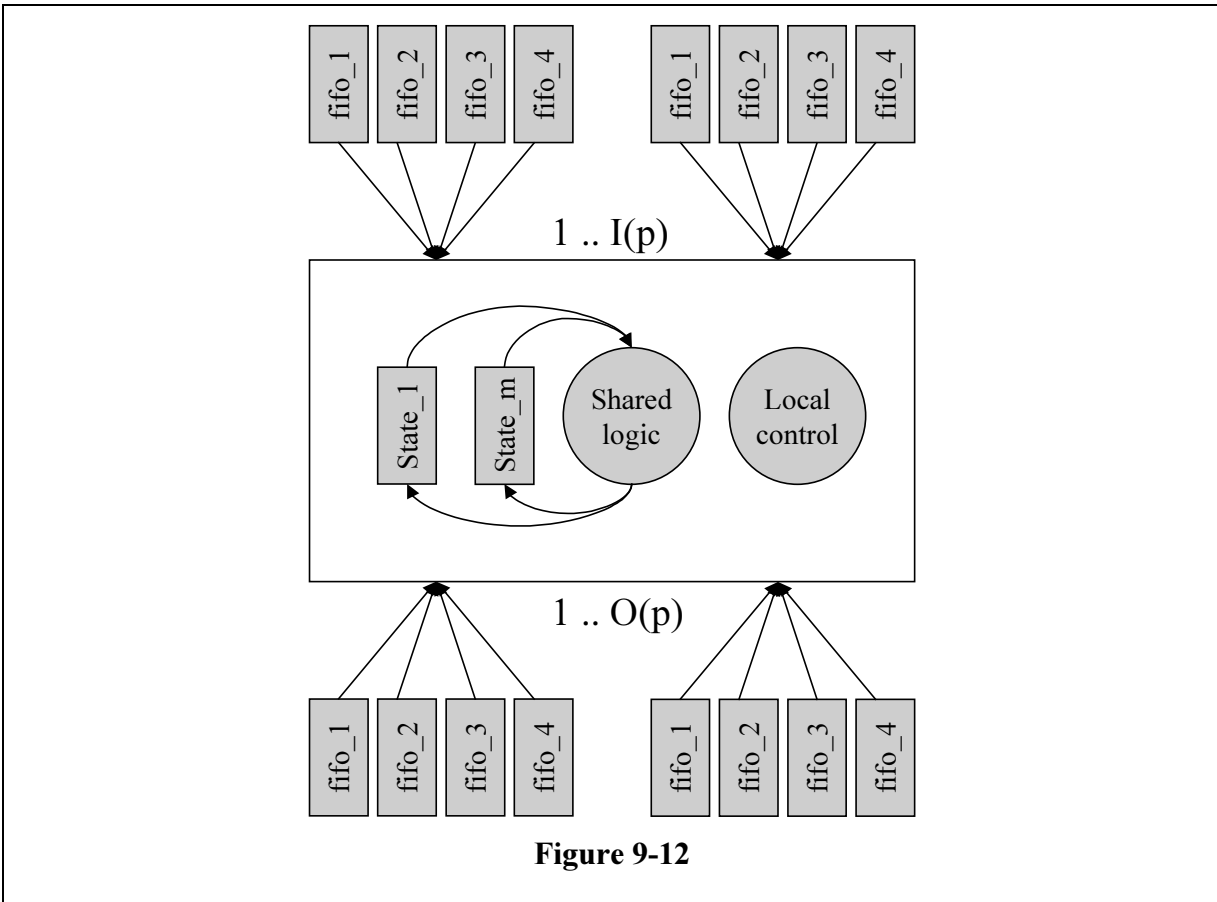
In the remaining part of this text we choose for the last solution. This means that between processors hardware is used in common and that we will also study in what way sharing can be applied in the communication networks. Fifos, on the contrary, are not used in common.

### 9.3.2 Processor model

We now study in which way processors can execute several tasks parallel. Given are one or more graphs in which functions appear that seem to be more or less similar. We want to map these functions unto one and the same processor (see the previously mentioned HSRC example in Figure 9-2). It is only natural to design only one HSRC processor. Since the data stream with the headers is divided into packages there are two possibilities. Either between functions connections are made at the moment that the packages are finished. Or a smaller grain is chosen and connections between functions can be made at a random moment. We call this *sub-package interleaving*. The advantage of the first method is that the state is often smaller (or even zero) when the package is processed completely. The disadvantage, however, is that the input data have to be buffered during a longer time.

This processor should be sufficiently flexible, though, to process a number of 1 to 4 streams. This leads to a model as is shown in the next figure. The processor has a number of input ports and output ports that follow from the function, but that are parameters in the model ( $I(p)$  and  $O(p)$ ). Each input port and output port is connected with 4 fifos at a maximum, one for every workspace. The number of workspaces once again is a parameter in the model ( $m(p)$ ). Furthermore we have just as much internal workspaces and a number of combinatory logic that is shared. Next there is a local controller. The idea is to separate the communication from the processing. For that purpose the local controller can decide autonomously which task makes progress and when. This is a piece

of run time scheduling in which the filling degrees of the input and output buffers are watched.



### 9.3.3 Communication network

The task of the network is to provide enough band width for the data streams between the output fifos and the input fifos of the processors. For this we make use of *circuit switching*, i.e. that we will create certain connections through which the data can be transported. A possible problem then is *blocking* which means that certain connections cannot be made because of earlier made connections. Consider, for instance, a network with A inputs ( $a_1, a_2, \dots, a_A$ ) and B outputs ( $b_1, b_2, \dots, b_B$ ) in which  $A > B$ . Here blocking may occur. See, for instance, the next figure which shows a 4x3 network where input line 4 is blocked. It is not possible to make a connection because there is no free output terminal available.

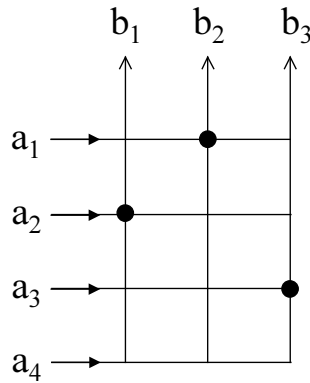


Figure 9-13

When  $A \leq B$  then the network is non-blocking. For instance when  $A=B$  then switching is always possible, that is to say, when no connection is needed with two times the same output terminal. The complexity of such a network is  $A \times B$  when we assume that the sets of input and output terminals are disjunctive. To limit these costs we move on to the so-called TST networks. This stands for *time-space-time* networks that consist of three parts. In the following figure we first explain the analogy between a T-switch and an S-switch. In an S-switch physical connections are used between inputs and outputs. In the example in the figure stream  $s_1$  is transported from input  $a_2$  to output  $b_3$  and stream  $s_2$  from input  $a_4$  to output  $b_2$ . In a T-switch the same functionality can be implemented but this time by switching a single physical connection four times in the time. For that purpose the input is provided with a multiplexer and buffering. At the output port we have a demultiplexer and also buffering. After 4 time slots the whole pattern repeats itself. So we may speak of a frame. The T-switch switches the order of the time slots within a frame. For example, in the figure stream  $s_1$  arrives in time slot  $a_2$  at the input and leaves the switch at the output on time slot  $b_3$ .

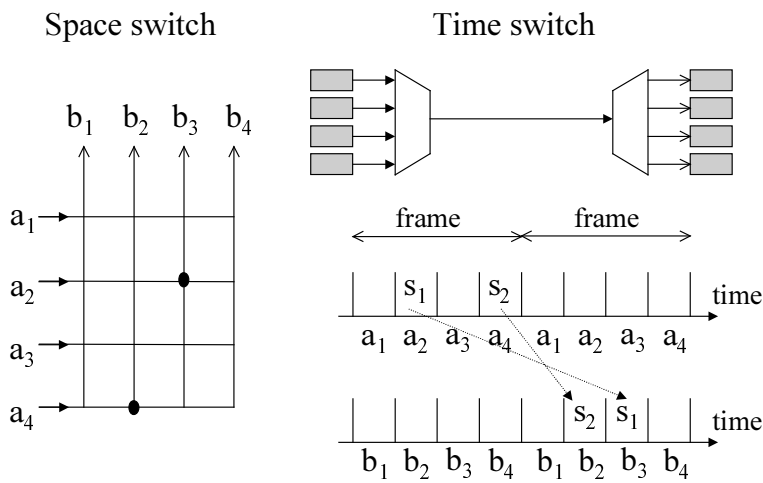
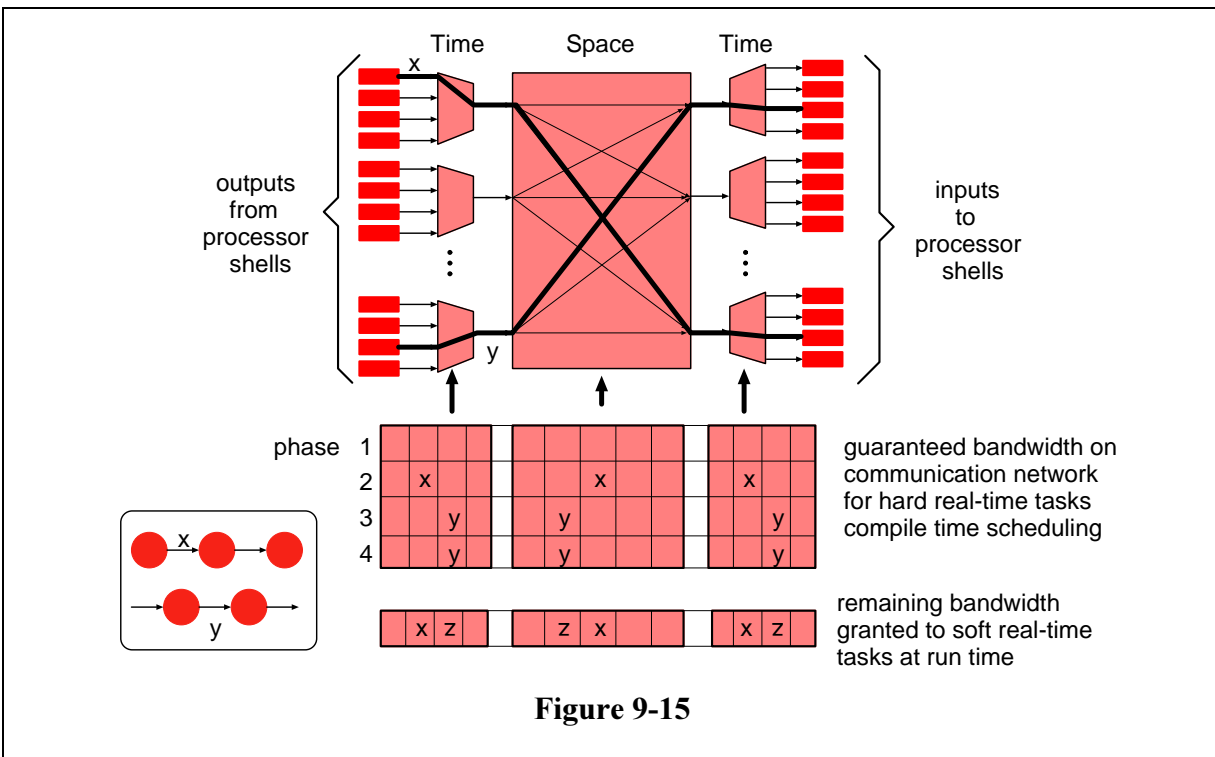


Figure 9-14

Eventually this leads to the TST network of the next figure. Two paths through the network are indicated. To make a specific connection a certain code has to be provided for the three different parts. The T-switch uses a multiplex factor of 4. A specific path then can be programmed in a table with four phases. The x-connection for instance, has been programmed in phase 2 by providing the right code in the compartments labeled x. By doing so a band width corresponding with one phase is guaranteed. The connection y has a double amount of band width because this connection has been programmed during two phases. In this way band width guarantees can be given for hard real-time streams. Additional yet another extra row has been provided for the soft real-time streams. When the hard real-time streams in certain phases do not use the band width this can be used for the soft real-time streams.



#### 9.4 Conclusion

When we put together a complete architecture we reach the following figure. So we have given the signal processing and the event processing their own sub system each. Within each sub system different communication principles are used. In this way each sub system can be optimized separately. Both have access to the external memory and the arbitrage mechanism has been discussed extensively.

The most important characteristic of this architecture is the *hierarchy*. This is present in many ways. When we look to the control that the user has, then there is the embedded cpu at the top level. On this the user can write programs, for instance to develop a user

interface of his own. But he may also develop application software and doing so create certain flow graphs. One level down we find the various processors in the event based sub system but also the control of the signal processing part. On the embedded cpu, for instance, a certain flow graph may be prepared that then can be loaded at the appropriate moment in the controller of the periodic sub system. This controller is further responsible for the communication between processors in the periodic sub system. The CPU, for instance, is no longer informed when certain communication takes place in the periodic system. Hence, there are clearly two levels of control: the cpu and the controller of the periodic sub system. In fact there is even a third level, namely the controller that decided for each processor which task is executed.

A similar hierarchy we find when we look at the memories. At top level we find the external SDRAM and the corresponding arbiter. One level down we find a buffer that is sufficiently big to store every stream that communicates with the periodic sub system long enough, even when in the meantime the CPU needs the memory. At the third level we then find the fifo buffers that encapsulate the processors.

It is this variety at the various hierarchical levels which is a characteristic of system level design. It is also an illustration of the wide variety of system level architectures that are possible. What has been discussed in this chapter, therefore, is merely an example.

