TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

The control of a digital copier based on the Intel 80960SA

Manrique Lizarraga, Jose de Jesus

Award date:
1995

Link to publication

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

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

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

EB 579

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Section of Digital Information Systems

Master's Thesis:

The control of a digital copier based on the Intel 80960SA risc processor.

José de Jesús Manrique Lizárraga.

Coach	: Ing. Cees van Tilborg (Océ)
	Ir. Hans Wierink (Océ)
Supervisor	: Prof. Ir. M.P.J. Stevens
Period	: August 1994 - February 1995

"le coeur a des raisons que la raison ne connaît pas"
Pascal

The Faculty of Electrical Engineering of Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses.

Abstract.

CISC (Complex Instruction Set Computer) processors have been used in embedded applications since the introduction of the microprocessor, mainly because of its flexibility. Low cost solutions can be achieved when the peripherals are integrated into the microprocessor chip to form a microcontroller. During the last decade another kind of processors has been developed to respond to the increasing demand of performance, from those applications of intensive data manipulation like a workstation: The RISC (Reduced Instruction Set Computer) processor.

There are different implementations of RISC, every producer has its own architecture. The common features are the use of cache memory for those pieces of code and data that are often used by the processor, and the substitution of microcode by hardwired techniques. These characteristics have proven to boost the performance of the conventional processor, but imply a certain degree of uncertainty or non deterministic behaviour which might affect negatively the operation of real time applications.

Certain applications of the embedded sector, that also require high performance for intensive data processing, have taken advantage of the RISC developments, an example is a laser printer. But in real time systems, which are interrupt driven, the context switching can make the effectiveness of the RISC approach less notorious.

Even though, the combination real time and RISC is not an obvious one, those RISC processors that are used in the industrial sector offer other benefits like long term availability and often more performance when compared to the existing CISC processors and microcontrollers.

This work deals with the implementation of the control of a digital copier based on the Intel 80960SA RISC processor, a member of the i960 family of embedded processors. This processor offers the benefits of RISC , but its internal facilities have been simplified (no MMU, no FPU) as well as the external data bus (16 bits multiplexed address/data bus), which makes the processor interesting for low cost applications.

The control of a digital copier is a multitask real time system. Because of its digital character, more functionality is available than with the analog counterpart. Furthermore, the copier can behave like a printer or like a scanner. This makes possible the integration of the copier (or portions of it) in other environments. To cope with all this, there is an increasing performance demand of the control unit of the machine.

Several hardware architectures are considered in order to find a soluition that shortens the time to market. An implementation for the short term start of production is given as well as a development plan for the future versions.

One important engineering trade-off is the integration of the control panel into the control unit. By doing this, the price of the system is maintained low, but this will increase the required processor power, because the control unit will be in charge of generating and manipulating bitmaps for a graphic Liquid Crystal Display.

Because RISC may involve risks in real time applications, an analysis of the extent of these problems is given. An important conclusion is that the contribution of the RISC architecture to the latency time (for asynchronous context switchings) is in average slightly higher than the time required by the scheduler to perform its administration tasks in this application.

Table of Contents.

1 Introduction 4

2 Control of a digital copier 5

3 Hardware architectures of a digital copier 9

4 Implementation of the engine control, phase 1 14

4.1 Choice of the Main Processor 14

4.2 The 80960SA processor 16

4.2.1 Processor highlights 16

4.2.2 Execution environment 17

4.2.3 System Data Structures 19

4.2.4 Call/Return mechanism 20

4.2.5 Interrupts 23

4.2.6 Signaling interrupts 25

4.2.7 Internal Structure of the 80960SA processor 26

4.3 The Interprocessor Communication 29

4.4 The Eprom Interface 33

4.5 The RAM interface 35

4.6 The hardware system architecture 36

5 Implications of using a RISC processor for the control of a digital copier 38

5.1 Context Switching 38

5.1.1 Asynchronous context switching 38

5.1.2 Synchronous context switching 47

5.1.3 Effects of the CPU to the context switching 49

5.2 Non deterministic behaviour 53

5.3 Debug facilities 53

5.4 Amount of software code 54

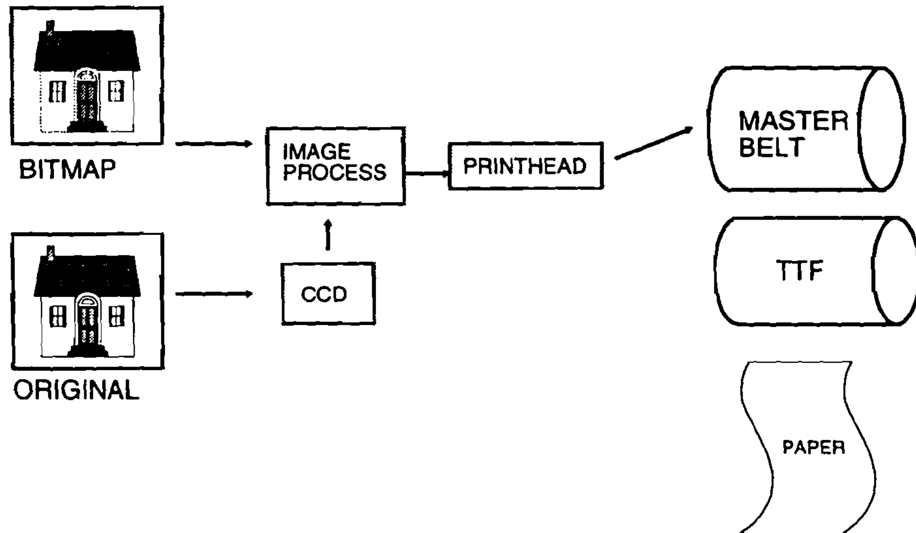
6 Conclusions 55

7 Literature Index 59

APPENDIX A1 Communication Memory Interface (GAL Listings) A1-1

APPENDIX A2 EPROM interface (GAL listings) A2-1

1 Introduction.



Analog copiers that are controlled by digital means have been produced by Oce during the past 15 years. The digitalization has gained even more territory influencing also the copier process of the machine. This has created a number of possibilities that has transformed the copier from a "black box" document reproducer into a system that can be integrated in other environments. A digital copier can for instance operate like a printer and the digital image processing functions can service alien devices like an external scanner. This results in an increasing performance demand to the control unit of the copier.

The complex image processing items are available to the user through a control panel with a graphic Liquid Crystal Display. The control unit will be in charge of generating and manipulating the necessary bitmaps

To cope with this functionalities as well as the multitasking real time aspects of the engine control, a high performance hardware architecture is needed.

This work deals with the analysis of different hardware architectures under the following constraints:

- Strategic need to exchange, to remove and to add modules, in order to reduce the time to market.
- Overall price of the system should be as low as possible.
- Reserve processing power for future applications.
- Flexibility items in the design to support the development of functions.
- A solution should be ready for production by the summer of 1995.

Furthermore the implications of using a RISC processor for this real time multitasking system should be identified.

2 Control of a digital copier.

A brief description of the activities of the control of a digital copier are given in this chapter. Some items have already been used for years by the analog copiers, but the digital character of the system gives them an extra dimension: the digital copy process involves an implicit storage of the scanned information, this makes the retrieval of the original information independent of the actual copy process.The following figure depicts the different functions in a digital copier:

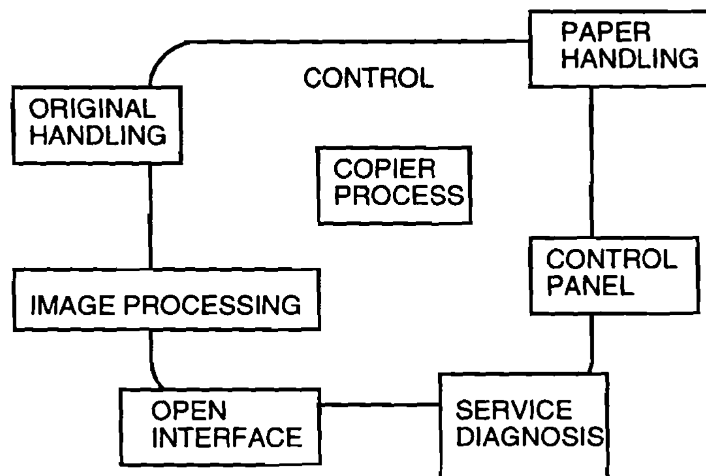


Figure 2.1: Digital copier, control functions.

- Copier process.

This function involves a number of actions that constitute the oce technology of analog copiers: A light sensitive master is uniformly moved and passed through a series of stations, at the first one, electrostatic charge is deposited on the master, then the printhead creates an electrostatic image of the document by illuminating and in consequence reducing the impedance of those positions where the charge will flow to ground. The charge image is developed by the accumulation of toner particles at the places where the charge is still present. The toner is transferred to a silicon band, taking advantage of the higher adhesion coefficient of the band to the toner particles than the cohesion coefficient of the particles. Then the particles are pressed and fused into the recipient paper under high temperature.

This is a real time control system that needs a master position reference clock and a real time clock, to perform a number of simultaneous controls. The productivity of the copier (amount of copies per minute: cpm) and the precision of the registration depend on the performance of the processor and its interrupt latency. The last parameter includes the context switching and the scheduling time of the last functional task. For a productivity of 60 cpm and 0.1 mm registration error, the maximum interrupt error may be:

t= .1 mm x 1 sec / 210 mm = 476 microsec.

- Original handling.

The original information is retrieved in two phases: during the first phase, the scanner moves to its start position and delivers rough information to the image processing to determine the background compensation. In case that the Automatic Document Feeder (ADF) is used, both original and scanner move to the start position (the original is transported to the same position it would have if no ADF is used). During the

second phase the actual scanning process is started: the CCD (Charge Coupling Device) is uniformly moved from the start position.

The ADF function consists of the accurate control of motors to separate, transport and eventually to turn a duplex document. The precision of the registration is closely related to the stability and precision of the velocity profile of the scanner motor. The speed of the scanner motor determines also the zoom factor gain in one direction. For reductions the zoom factor is determined by the image processing for both directions.

Because of the fact that the scanning process is independent of the print process, the velocity of the motors does not need to be synchronized to the master motion. Furthermore, the information is stored in a set memory in order to be able to reproduce the original several times (Electronic Recycle Document Feeder): scan one, print many (this improves the reliability of the system).

- Paper Handling.

This function is related to the storage, separation, transport, precise feeding into the (toner) transfer unit and output of the paper. All along the paper trajectory, there are a number of sensors to signal a possible paper-jam.

- Control Panel.

The functionality of the copier is made available to the user by means of the control panel. The last generation of analog copiers of Oce makes use of dedicated keyboards and standard graphic LCD's. Graphic information is used to warn about possible positions of paper-jams and to generate fancy menus for the different operating modes.

- Service Diagnosis.

This is actually a function that covers the whole machine. It applies hardware feedbacks and software routines to verify the different functions and to monitor the sensors. The installation parameters are also introduced through this function.

- Image Processing.

This function characterizes the digital behaviour of the copier. It consists mainly of the following operations:

Scanner compensation.

Photo quality reproduction algorithms.

Zoom.

Background histogramming.

Conversion of 400 dpi scanned information with gray values into a 600 dpi bitmap.

Rotation (90°, or 180°).

Overlay functions.

Compression.

Storage into a set memory (dram).

Decompression.

Deliver of processed information to the printhead.

The implementation of these operations require sequential manipulation of 34 Mpixels (A4 document), this can only be achieved by the use of dedicated components:

ASICS. A certain degree of flexibility is attained by parameters that are updated by the image processing control.

The image processing control is in charge of the control of the asics and synchronization with the copy and original processes. The retrieval of document information (scan process) is validated by a real time signal generated by the original information control, and the print process is enabled by a real time signal generated by the copy control.

- Open interface

A document can also be a bitmap delivered by an alien scanner or a computer. The copier behaves then as a printer. In case the information comes from a computer, a SCSI interface is used.

The functions of a digital copier can also be roughly divided into two categories: Data Processing (Image Processing control, and open interface) and Engine control (the rest of the functions).

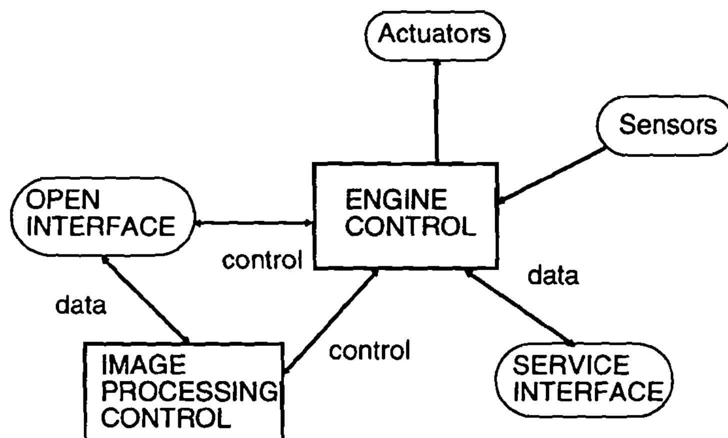


Figure 2.2: Digital copier, rough division of the control.

The image processing with its dedicated functionality is a stable unit with limited degree of flexibility. Future needs will require a technology migration towards a new generation of DSP's. The Image processing control will keep its present implementation for the years to come and will be used as a black box by the different versions of the basic copier. This functional block will have its own control unit.

The functions related with the analog part of the copier have been controlled during the last decade by CISC processors of the fourth generation: Motorola 68000 or similar.

The engine controller depends in a great extent on the configuration of the machine. There is the need to add, to remove or to exchange modules. The basic machine will have for instance three paper reservoirs and no sorter. Another spin-off product will

require a sorter and a fancy finisher. Other project will require the same ADF/ Scannner but different copy control and paper handling.

One of the subjects of this report is to find the most suitable engine control architecture to satisfy the different configurations of the machine, so that the time to market can be shorten.

By the time this work was started, fundamental development was taken place on the ADF/Scanner and the behaviour of the master during the production process was analysed in order to improve its yield. One important constrain was the fact that a hardware solution should be ready for production by the summer of 1995

3 Hardware architectures of a digital copier.

The strategic need to exchange, to add and to remove modules of the machine to shorten the time to market of spin-off projects asks explicitly for a field network. In such a network, the different functions are implemented as individual modules that receive operating instructions from a central control unit. The communication medium is a bus or a ring that gives the possibility to expand or remove functions without considerable deterioration of the quality of the signals.

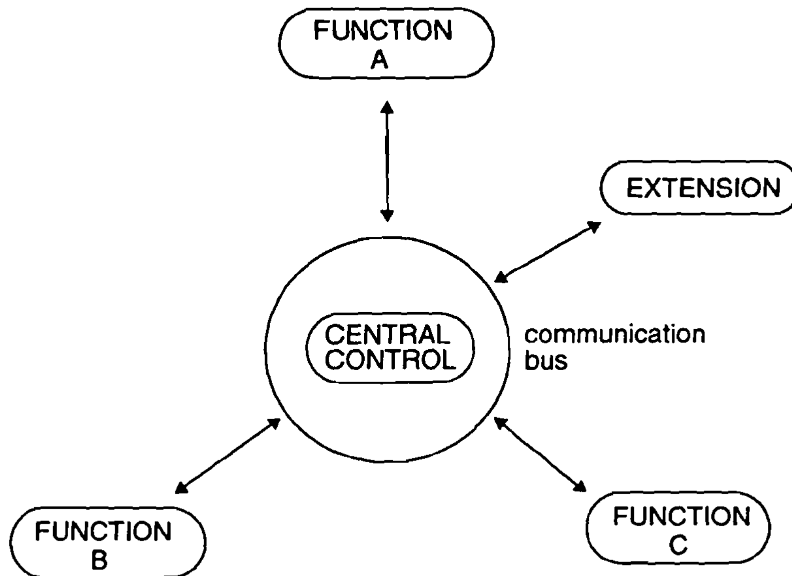


Figure 3.1: Modular approach for the hardware architecture.

The digital nature of the machine with features that may be used by the outside world demands whether spare processing power or the possibility to add it when necessary.

A modular approach implies a very flexible solution, and when a serial bus is used, the reliability of the system is increased due to the low amount of wring. However the communication overhead poses limitations to the real time behaviour of the system. Another factor that determines the hardware architecture is the overall price of the system. By choosing a different module per function, a great deal of resource sharing is lost and the price increases. Activities involved in one function can not easily be distributed among the intelligent elements of the network and idle items of the modules are not ready to be used by the others. Furthermore, because of the fact that a design should be flexible enough to support the development of the functions (see chapter 2) a serial bus might introduce limitations. The trade-offs to be taken into account can be summarized as follows:

- Strategic need to exchange and add modules.
- Overall price of the system.
- Reserve processing power for future applications.
- Flexibility items in the design for development support.
- A hardware solution should be ready for production by the summer of 1995.

The need of spare processing power can be satisfied by choosing a processor with more than enough power to control certain functions of the copier. Spare processing power means that the infrastructure of the processor should also be available or expandible in order to cope with the required functionality. The advantages of having spare processing power instead of adding it when necessary, are that the costs of interfacing the processor are spent one time and the communication overhead with the eventual extra processor is avoided. Besides the system remains simple which reduces the chance of failures. The powerful processor will get the role of main or central processor, and its functionality will be referred to in the following paragraphs as the control-core functionality.

The following figure depicts the possible combinations to form the basic functionality.

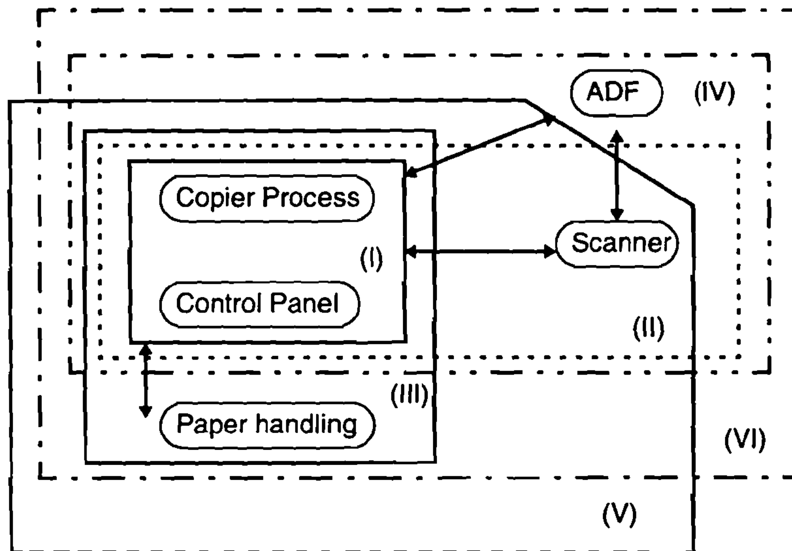


Figure 3.2: Basic functionality, possible combinations.

There are three kinds of modules:

a) Fixed modules.

These are present in all the different configurations without alterations (copier process, control panel).

b) Exchangeable modules.

Which can be removed or added to the system (ADF/Scanner).

c) Configurable modules.

That are built upon submodules whose amount depend on the product (paper handling: amount of paper reservoirs).

It is obvious that the control-core will be at least in charge of the fixed modules: the copier process and the control panel.The interfacing to the control-core should be as simple as possible in order to keep the communication overhead low, and guarantee the real time behaviour of the machine. This can be implemented by four intelligent modules: The control-core, the ADF, the Scanner and the paper handling. The com-

plexity of the paper handling is low and its actual control functionality can be added to the control-core without significant performance deterioration, the modular behaviour can still be kept by using remote i/o submodules for this configurable function. This leads to combination III.

Modules D and C have a high influence on the real time behaviour of the machine, both need the accurate control of motors, a close synchronization with each other and the Image Processing control, and fast detection of errors to prevent damage of documents and mechanic elements. The use of microcontrollers (embedded controllers) for these functions is the most practical choice. The fact that the scanner only controls one motor, the processor capacity and the integrated i/o peripherals of the eventual scanner microcontroller may be available to the control-core (combination V). This introduces a great amount of resource sharing that allows the main processor to reserve even more performance for future applications and reduces the overall price of the system. A fast communication medium for both processors is required in order to optimize the resource sharing and comply with the real time requirements of the system.

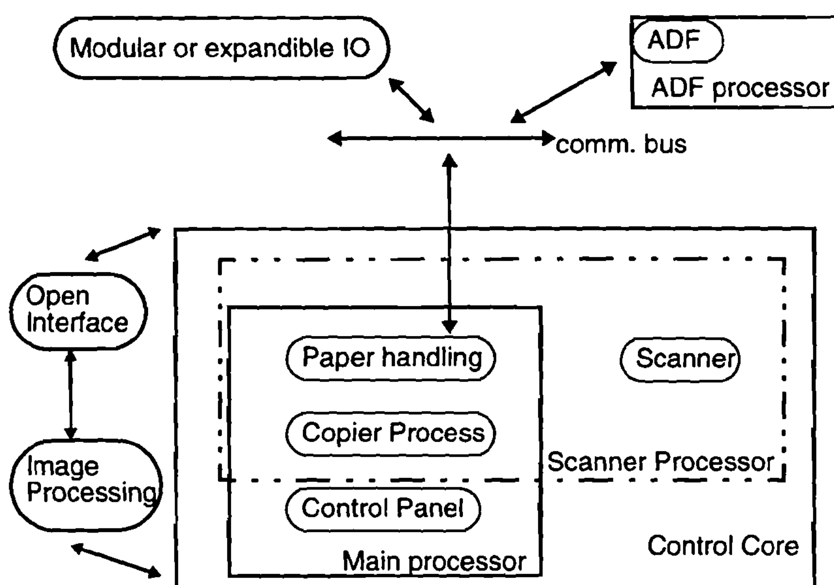


Figure 3.3: Modular solution with a certain amount of resource sharing.

The combination of the original retrieval functions (ADF and Scanner) might be an interesting alternative. Both functions could be implemented by one processor, but the lost of modularity does not render the advantages of the solution depicted above.

The introduction of a bus between the control-core, the ADF and the paper handling submodules, makes automatically possible the expansion of the system.

A problem needs still to be solved: how can the development of the functions be supported by means of a hardware implementation ready for production? The functions that might require expansions or modifications are spread among three different processors. The kind of modifications can be as simple as the addition of control lines or more complex electronics that need to be accessed by one of the functions at high speed. Flexibility items at the different processor buses might be an alternative, but it would increase the cost and the complexity of the system. A more adequate solution is the addition of one expansion I/O bus to the processor with the highest performance, and let also the ADF processor share the high speed communication

medium of the control-control, this way all the functions would access directly or indirectly (through the main processor) the needed hardware functionality. The physical modular approach is lost but the start of production would not be in danger. Furthermore the real time risks of a serial bus are avoided. In order to regain modularity at a later stadium, the following development phases are proposed:

- Phase 1.

A Flexible solution with three processors that communicate with each other by means of a high speed communication medium. The main processor will have an i/o expansion channel for extra hardware functionality for any function.

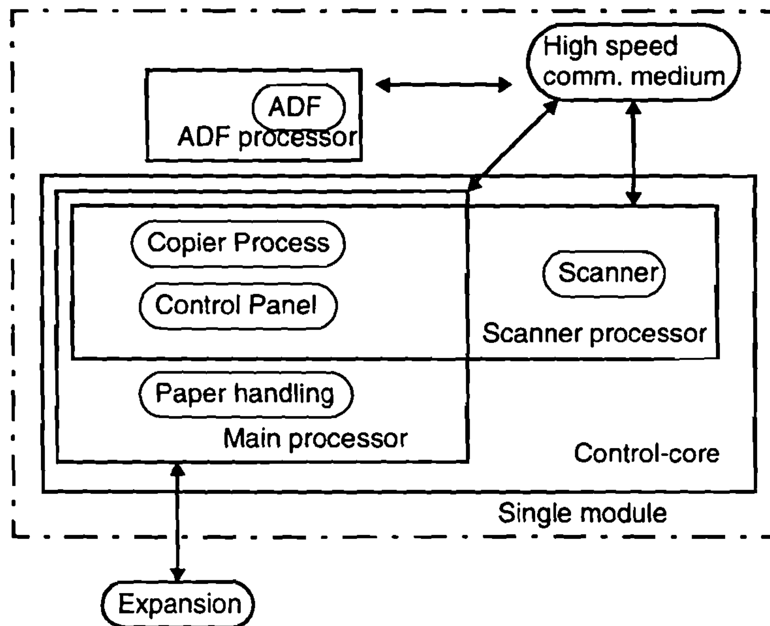


Figure 3.4: Flexible solution.

- Phase 2.

A value engineering step to decide whether or not to separate the ADF processor, and investigate the real time implications of a serial bus.

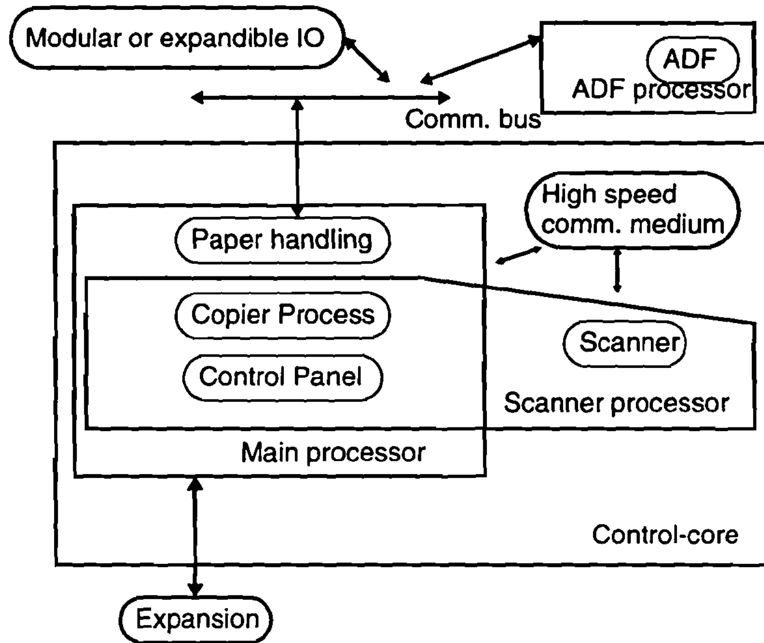


Figure 3.5: Modular solution.

This report will deal with the further realization of phase 1, and in particular with the performance issues of the main processor and its real time behaviour.

4 Implementation of the engine control, phase 1.

This chapter deals with the implementation of a flexible solution for the engine control of the digital copier. As stated in the previous chapter, the control-core will be based on a high performance processor, which is tight coupled to the scanner embedded controller. The ADF controller will be so far also coupled to the internal high speed communication medium of the control-core so that it can take advantage of the extension i/o bus of the main processor.

We are going to concentrate our attention to the performance items of the main processor: the clock frequency, the eprom interface, the high speed communication medium and the context switching during interrupts. A brief discussion of the considerations that led to the choice of the Intel 80960 SA-processor follows first:

4.1 Choice of the Main Processor.

The factors that contribute to the choice of the 80960 SA were:

- Compatibility with the ADF/Scanner-processors.

The 80196KC microcontroller was chosen for the ADF/Scanner functions, a member of the MCS-96 family. This embedded controller will operate at 16 MHz and zero wait states for eprom accesses. It contains a number of analog functions (ADC's and PWM's) which can be used for the implementation of the temperature and motor controls of the machine. Besides, it has normal and high speed i/o lines (HSI/O) that operate in combination with the 2 internal 16 bits timers for accurate i/o control. A serial channel is available. Furthermore a Peripheral Transaction Server (PTS) provides DMA-like response to an interrupt with few CPU-overhead. Single and block transfers are supported, as well as special modes to service the AD-converter and the HSIO.

By choosing Intel microcontrollers and the fact that a high speed communication medium is needed, a compatible main processor is required to guarantee the simplicity and the transparency of the hardware: the data should be interpreted the same way by all the participants of the communication medium. To illustrate this, consider the following 16 bits aligned structure:

```
struct {
short a;            /* 16 bits a = AB */
char c;             /* 8 bits c = C */
short d;            /* d = DE */
char f,g;           /* f = F, g = G */
long h;             /* h= HIJK
};                  A through K are bytes*/
```

The data is stored into memory by a little endian machine (Intel processors) in a different way than a big endian processor (e.g. Motorola):

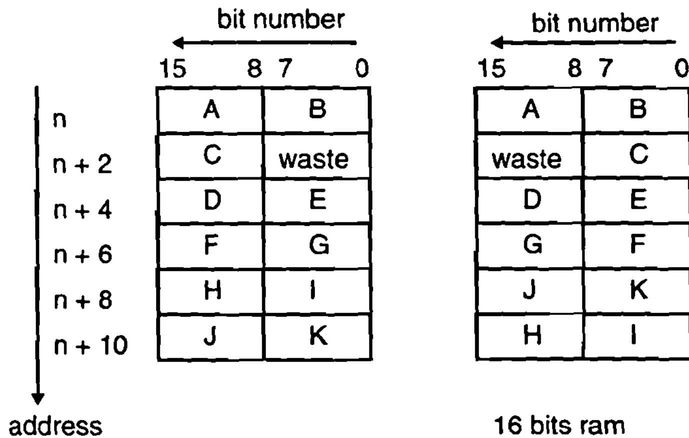


Figure 4.1: Little endian vs. big endian; memory allocation.

Byte swapping takes place at addresses n+2 and n+6 and word swapping at n+8 and n+10. A detection and eventually a correction of this situation need that the sequential accesses of one data type occur within the same machine cycle. The Motorola processors that comply with this requirement are the MC68020 and its successors, because they feature a dynamic data bus sizing. In order to keep low the complexity and cost of the system a little endian processor is chosen.

- Availability.

The general purpose market has been dominated by Intel with its 80x86 line. Intel has been able to confirm its position by introducing every couple of years more powerful processors maintaining their instruction set compatibility. The kind of applications of the personal computer area demands increasing of performance, which limits the life cycle of the processors (about 3 years). Embedded controllers on the other hand, are intended for a complete different market segment, where price is mostly the steering factor. Extra performance does not always results in attractive items for the end user (e.g. washing machine). The life time of this kind of processor is often longer: about 7 years. An indication that the embedded controller market is a different area is illustrated by the fact that 60% of the applications is still implemented by 4-bits processors, 30% by 8-bits processors, 9% by 16 bits processors and 0.8% by 32 bits processor (according to the Intel representantion in the Netherlands). Taking this into consideration it is recommended to make use of an embedded processor for this kind of application.

- Performance.

Taking into account that a MC68000 has already proven to control an analog copier with similar requirements, a processor with a comparable performance to that of the MC68020 is needed, which has more than two times processor power. This way, capacity would be available for future applications. The nature of this application: control and a certain degree of bit manipulation for the graphic display, makes a MMU and a floating point processor unnecessary. Possible embedded candidates are the AMD29K, the i960 family or one 80386-based microcontroller. The 80376 was about to be discontinued by Intel because of the planned introduction of the i386EX microcontroller. A choice had to be made between a low end processor of the I960 family of one of the AMD29K. A benchmark report [1] indicates that the 80960SA performs almost two times better than the AMD29200 using the Standford benchmark and

about 30% better when the Dhrystone is used. This figures give only an indication of the possible performance of the processors, because they do not reflect the demands of the actual application. These benchmarks do not include operating system calls and i/o control. However they suggest that the 80960SA might be a better choice.

- Cost, support and development equipment.

Even though these are engineering aspects, these factors are also mentioned to give a complete picture of the items that are taken into account during the development of a system in the industry. The cost of the 80960SA at 16 MHz is about USD 20.00 when quantities of 5 000 pieces/year are purchased. The cost of the AMD29200 is slightly higher. There are cross language systems for both processors and in circuit emulators are available.

Because of the fact that the 80960SA is an Intel processor, which guarantees compatibility with the other processors, and that the price and the performance are better, this Intel embedded processor has been chosen. A clock frequency of 16 MHz has been selected in order to accommodate standard memory components with low amount of wait states. The inherent characteristics of this risk processor and its implications to the real time behaviour of the system need to be analysed.

4.2 The 80960SA processor.

A brief description of the 80960 processor follows. We are going to concentrate our attention on the features of the processor that might affect the real time behaviour of the system: call/return mechanism and interrupt mechanism. The interprocessor communication, and the memory interfaces are considered later on, these items are tightly related to the performance of the processor.

4.2.1 Processor highlights.

- Load and store model: most of the operations are performed in registers rather than in memory. The architecture provides an amount of general purpose registers: For each procedure, 16 global- and 16 local-registers are available. The global-registers maintain their contents across procedure boundaries, whereas the processor allocates a new set of local registers each time a new procedure is called. There are four local register-sets on chip. The processor can perform burst transfers of 1, 2, 4, 8, 12, or 16 bytes of information between memory and registers during one machine cycle.

- On-chip caching of code and data: the size of the instruction cache is 512 bytes. Data caching is provided by means of the general purpose registers.

- Overlapped Instruction execution: This is accomplished through register score-boarding, which permits instruction execution to continue while data is being fetched from memory. When a load instruction is executed, the processor sets one or more scoreboard bits to indicate the target registers to be loaded. While the target registers are being loaded, the processor is allowed to execute other instructions that do not use these registers. The processor uses the scoreboard bits to ensure that target registers are not used until the loads are complete.

- Single-clock instructions: The processor is able to execute commonly used instructions such as move, add, substract, logical operations compare and branch in a minimum number of clock cycles. All the instructions are 32 bits or 64 bits long and aligned on 32-bit boundaries, this feature allows instructions to be decoded in one

clock cycle. It also eliminates the need for an instruction-alignment stage in the pipeline. About 50 instructions are one-clock-cycle-executable.

- Procedure Call Mechanism: Each time a call instruction is issued, the processor automatically saves the current set of local registers and allocates a new set of local registers for the called procedure. Likewise, on a return from a procedure, the current set of local registers is deallocated and the local registers for the procedure being returned to are restored. On a procedure call, the program thus never has to explicitly save and restore local variables that are stored in local registers.

- 4-gigabyte address space.
- On chip trace facilities to ease debugging.

4.2.2 Execution environment.

The execution environment of the processor consists of a set of general purpose registers, an arithmetic control register, Instruction Pointer (IP), Processor Controls register, Trace Control register and an address space of 4 gigabyte. The process controls register shows the current execution state of the processor. The trace facilities of the processor are controlled through the trace controls register.

- Register Model.

The 80960SA provides two types of data registers: global and local. The 16 global registers constitute a set of general purpose 32-bits registers, the contents of which are preserved across procedure boundaries. The 16 32-bits local registers are provided to hold parameters specific to a procedure. For each procedure that is called, the processor allocates a separate set of 16 local registers; there are four available sets. The general purpose global registers g0 through g14 are general purpose; g15 is reserved for the current frame pointer (FP), that contains the address of the first byte in the current stack frame. The local registers r3 through r15 are general purpose registers; register r0 contains the previous frame pointer (PFP), r1 contains the stack pointer (SP), and r2 contains the Return Instruction Pointer (RIP; for branch and link instructions g14 contains the RIP). The following figure shows the register organization. A description of the pointers is given in the next paragraphs.

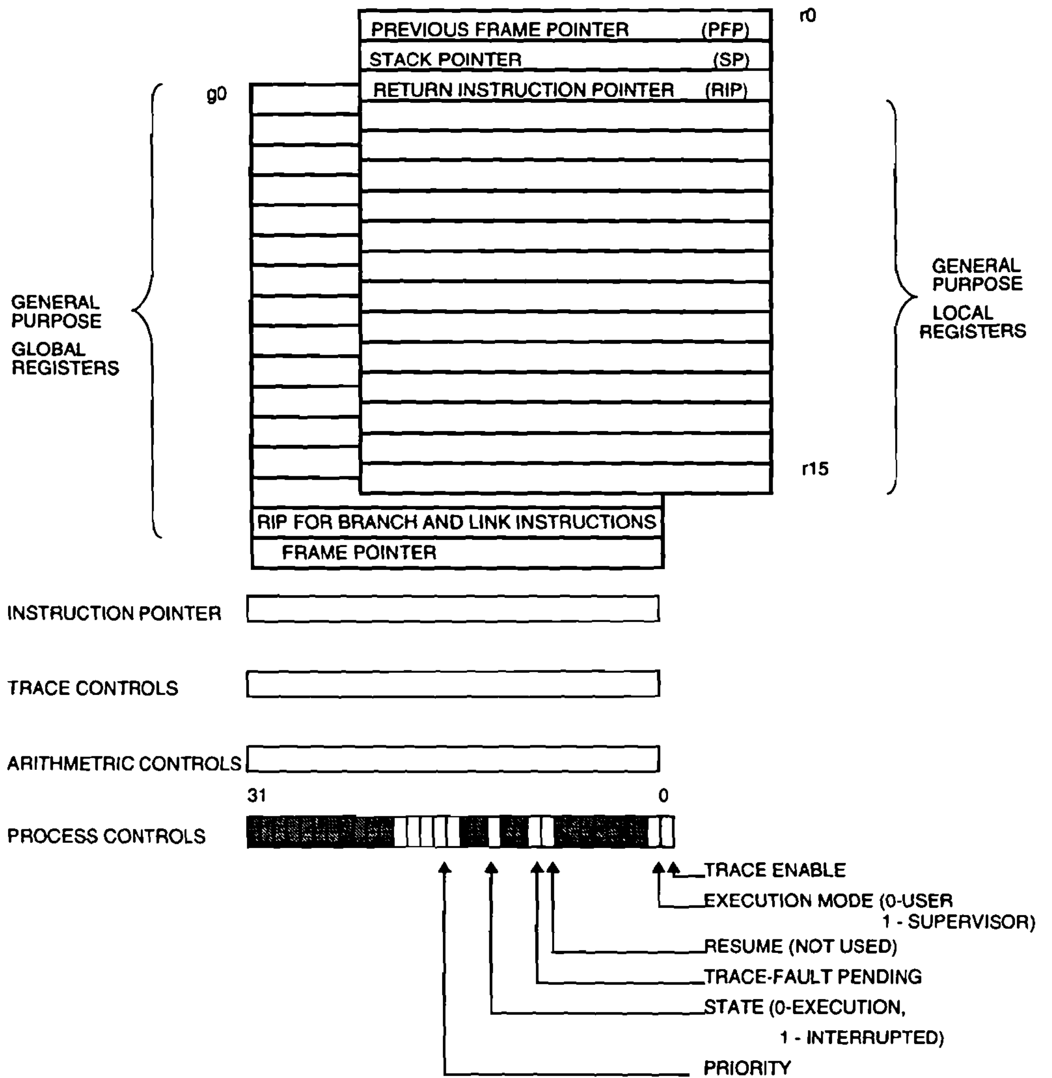


Figure 4.2: 80960SA, Register model.

4.2.3 System Data Structures.

The data structure offers a means to configure the processor to operate in a specific way

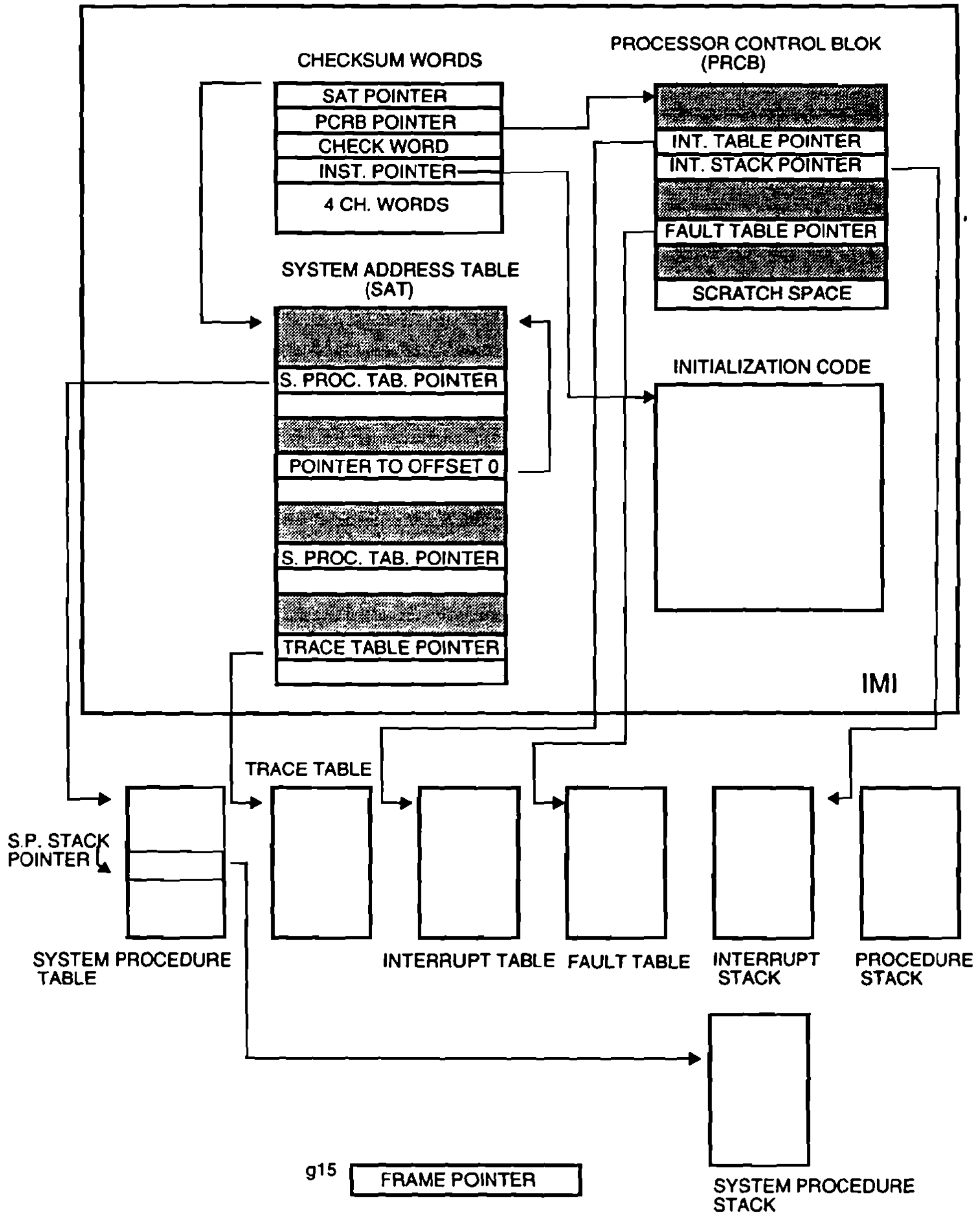


Figure 4.3: System data structures.

The IMI (Initial Memory Image) contains the minimum data structures required for the processor to initialize itself and begin executing code. It contains pointers to the other data structures. It is a common practice to copy a portion of the IMI into RAM during initialization. This can be necessary because of the dynamic character of the pointers as well as the processor scratch space. If the interrupt posting mechanism is used, the interrupt table must also be copied into RAM for the processor to operate properly because it contains the interrupt pending fields, which the processor must be able to write to. After the copy process, the processor is reinitialized with the new addresses

of the System Address Table and the Processor Control Block. (This reinitialization is performed through an IAC, Intra-Agent Communication, message. The primary function of the IAC mechanism is to provide an alternative to the external interrupt facilities to communicate with the processor, without the need of context switching. The processor handles an IAC in much the same way as it handles an instruction. All IACs have the highest priority level. The IAC messages are used for setting breakpoint registers, purging the instruction cache, or sending software initiated interrupts).

Upon power up the processor is in the interrupted state. It is a common practice to take the processor out of the interrupted state through the execution of a call statement, then the frame is fixed up to cause a return from interrupt. Before calling the application software, the frame pointer and the stack pointer should be initialized with the corresponding addresses at the user stack.

4.2.4 Call/Return mechanism.

The processor has two structures to support this mechanism: the local registers (on the processor chip) and the procedure stack (in memory).The stack consists of contiguous frames, one frame for each active procedure. For each procedure, the processor allocates a set of local registers and a frame on the procedure stack. If additional space for local variables is required, it can be allocated in the stack frame.

When a procedure call is made, the processor automatically saves (if necessary) the contents of the local registers on the stack frame for the calling procedure and sets up a new set of local registers and a new frame for the called procedure. When the number of nested procedure calls exceeds the number of registers sets (= 4), the processor automatically stores the contents of the oldest set of local registers on the stack to free up a set of local registers for the most recently called procedure.

The processor aligns each stack frame on a 64-byte boundary. Each frame provides besides a space for the local registers an optional area for additional variables; when the processor creates a new frame on a procedure call, it will if necessary, add a padding area to the stack so that the new frame starts on a 64 byte boundary.

Global register g15, and local registers r0-r2 contain information to link procedures together and to link the local registers to the procedure stack. The frame pointer (g15) is the address of the first byte of the current stack frame; the stack pointer (r1) is the address of the next available byte of the stack frame; the previous frame pointer (r0) is the address of the first byte of the previous stack frame; the return instruction pointer (r2) is the address of the instruction that the processor is to execute after returning from a procedure call. The following figure depicts the use of the pointers:

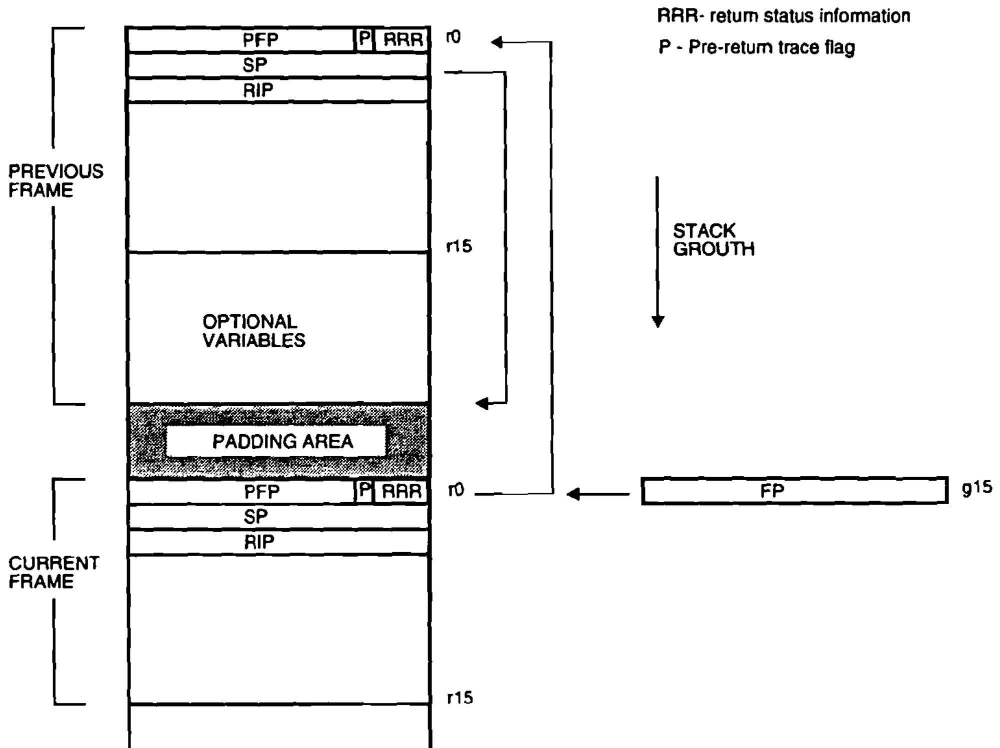


Figure 4.4: Procedure stack structure.

Occasionally, it is necessary to have the contents of all local registers sets match the contents of the register save areas in their associated stack pointer. The processor provides the *flushreg* instruction to allow voluntary flushing of the local registers.

There are three different kinds of procedure calls: local call, system call and branch & link.

- Local Call.

During a local call, the processor performs the following operations:

1. Stores the RIP in r2 (of the calling procedure).

2. Allocates a new set of local registers for the called procedure.

3. Allocates a new frame on the procedure stack (RIP is invalid).

4. Changes the instruction pointer to point to the first instruction in the called procedure.

5. Stores the FP for the calling procedure in new local register r0 (PFP).

6. Stores the FP of the new frame in global register g15.

7. Allocates a save area for the new local registers in the new stack frame.

8. Stores the SP in the new local register r1.

On a return the processor performs the following operations:

1. Sets the FP in global register g15 to the value of the PFP in current local register r0.

2. Deallocates the current local registers for the procedure that initiated the return and switches to the local registers assigned for the procedure being returned to.

3. Deallocates the stack frame for the procedure that initiated the return.

4. Sets the IP to the value of the RIP in new local register r2.

- System Call.

The system call is similar to the local call, except that the processor gets the IP for the called procedure from a data structure called the system procedure table

.

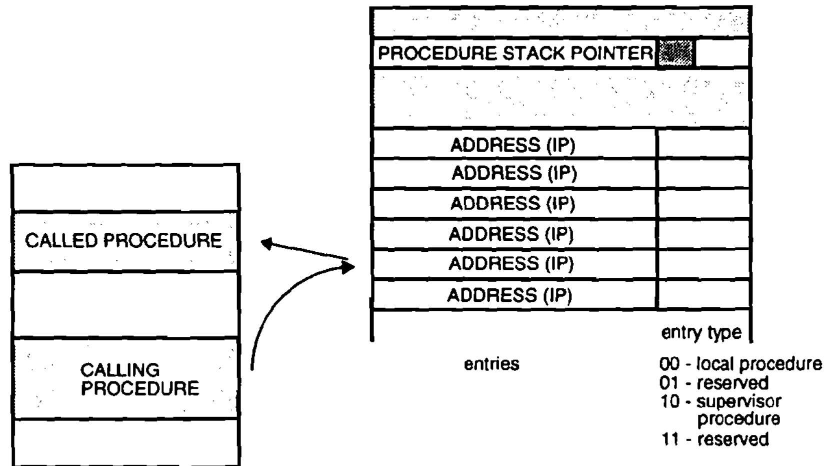


Figure 4.5: System call mechanism.

The System Procedure table contains a list of Instruction Pointers (IPs) which can be accessed through the system call mechanism. IP's for fault handlers may be stored. (The fault table contains a pointer to a specific entry of the System Procedure Table).

If the entry type specifies a supervisor call the processor switches to Supervisor execution mode and the Supervisor Stack becomes active. The processor gets a pointer to this stack from entry 12 of the System Procedure Table. When the supervisor mode is invoked, the field RRR of the local register r0 of the calling user procedure is set to 01X to indicate that mode and stack switch has taken place.

The processor provides a mode and stack switching mechanism called the user-supervisor protection model. This protection model allows a system to be designed in which kernel code and data reside in the same address space as user code and data, but access to the kernel procedures (called supervisor procedures) is only allowed through a system call.

The processor remains in the supervisor mode until a return is performed from the procedure that caused the original mode switch. When using the user-supervisor mode mechanism, the processor maintains two separate stacks in the address space, one for the procedures executed in the user mode (local procedures) and another for procedures executed in the supervisor mode (supervisor or system procedures). Both stacks are identical, except that the processor obtains the stack pointer for the supervisor stack from the system procedure table.

In the supervisor mode the Process Controls register can be modified through the modpc instruction. An application may be a modification of the processor priority.

- Branch and Link.

A branch and link instruction save the address of the next instruction (RIP-return instruction pointer) in a specified location, then branches to a target instruction. A return instruction specifies the RIP. The state of the local registers and the stack remains unchanged.

For the bal (branch and link) instruction the RIP is stored in g14; for the balx (branch and link extended), the RIP is specified with one of the instruction operands.

This kind of procedure call does not cause context switching. It is commonly used for the so called leave-procedures.

4.2.5 Interrupts.

The 80960SA supports a vector interrupt mechanism. An interrupt vector is 8 bits in length and has a predefined priority (= vector/8). At each priority there are 8 possible vectors. An incoming interrupt is serviced if the current priority is lower than the requested interrupt. A priority 31 is always serviced immediately (if the processor priority < 31; priority level 0 is not supported, because the memory locations that would be required for the interrupt vectors, are use by the interrupt table header).

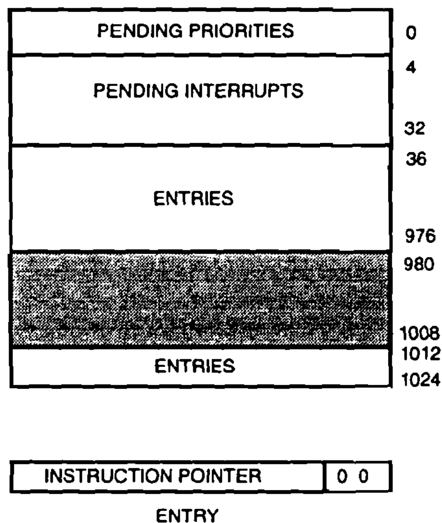


Figure 4.6: Interrupt Table.

A vector number specifies an entry to the Interrupt Table. The first 36 bits of the interrupt table are used to record pending interrupts. This section is divided into two fields: pending interrupt priorities and pending interrupts. When a pending interrupt is logged, its corresponding interrupt pending bit and interrupt pending priority are set.

The processor execution mode is set to supervisor while an interrupt is being handled. When an interrupt service routine is called, the states of the process controls

and arithmetic controls for the interrupted program are saved. The interrupt handler shares the global registers with the user/system procedures.

Except from the interrupt record, the interrupt stack has the same structure as the local procedure stack.

The method that the processor uses to service an interrupt depends on the state the processor is when it receives the interrupt: execute or interrupted. In the first case the processor switches to the interrupt stack. In both cases an interrupt record is also stored on the top of the stack prior to the new frame that is created for use in servicing the new interrupt.

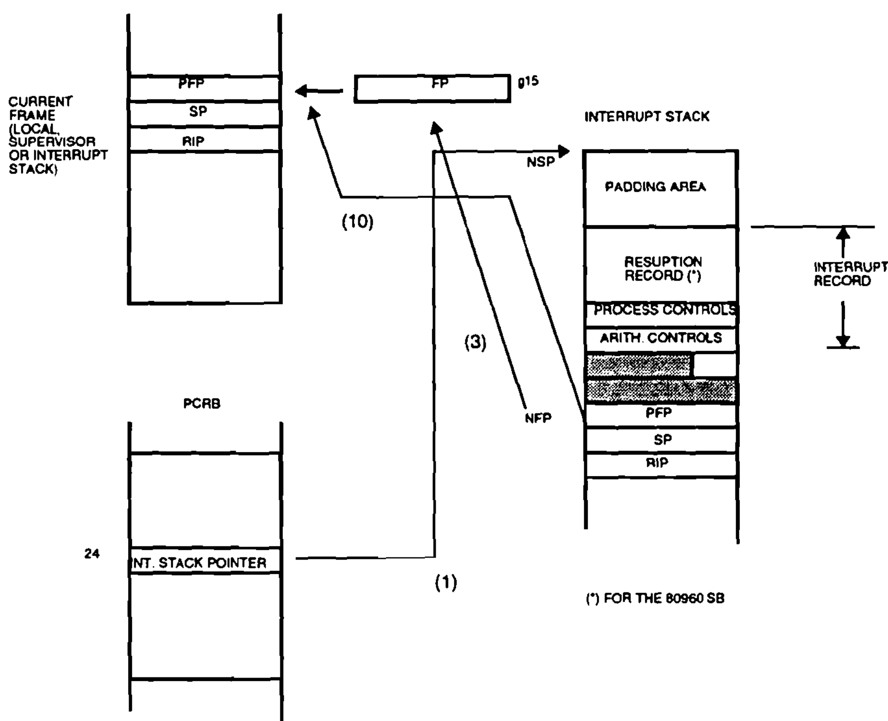


Figure 4.7: Framing manipulation during interrupts.

The following events take place:

1. If necessary, the processor switches to the interrupt stack. The interrupt stack pointer becomes the new stack pointer (NSP) for the processor.

2. The processor saves the current state of process controls and arithmetic controls in an interrupt record on the interrupt stack.

3. The processor allocates a new frame on the interrupt stack and loads the new frame pointer in register g15.

4. If necessary, the processor switches to interrupted state.

5. The processor, sets the state flag in its internal process controls to interrupted, its execution mode to supervisor, and its priority to the priority of the interrupt.

6. The processor sets the frame return status field (associated with the PFP in register 0) to 111 (interrupt return).

7. The processor performs an implicit call-extended operation. The address for the procedure call is that specified in the interrupt table by the corresponding interrupt vector.

10. The frame is linked to the previous frame through the address of the previous frame being written into r0 of the new created frame.

Once the processor has completed the interrupt procedure, it performs the following:

1. The interrupt record is restored.

2. The processor deallocates the current stack frame and if necessary switches to the user or system stack.

3. The processor checks the interrupt table for pending interrupts that are higher than the priority of the program being returned to. If a higher interrupt pending interrupt is found, it is handled as if the interrupt occurred at this point.

4.2.6 Signaling interrupts.

The 80960SA can be interrupted by: signals on its interrupt pins, signals on the interrupt pins of an external interrupt controller, an IAC message from a program or a pending interrupt. The processor has four interrupt pins, two of which can be configured for handshaking with an interrupt controller such as the 8559A programmable interrupt controller.

In this application more than 4 interrupt lines will be needed, the following figure shows the how these lines are going to be used:

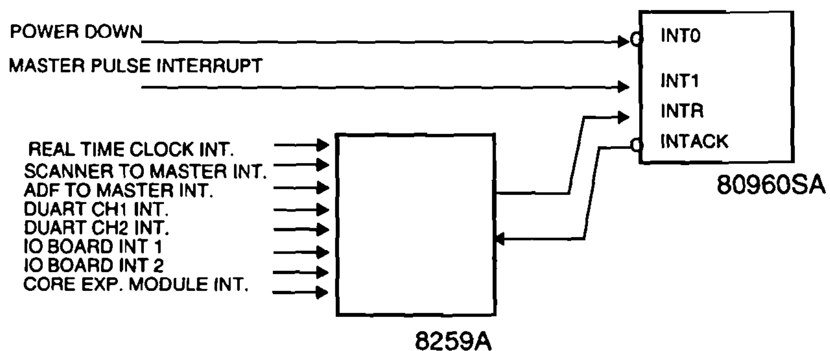


Figure 4.8: Interrupt signals.

The interrupt control register is memory mapped at address 0ff000004. Only the processor can access this register through two instruction (synld, synmov), which prevents the register from being modified by another external agent. The interrupt vector for INT0 is specified at the first byte of the interrupt register, that for INT1 should be specified in the next byte.

Important to know now is that both the real time clock and master pulse interrupts are used by the executive. The first one provides the time reference, the other one the position reference with respect to the master belt. The frequency of the master pulse

is 300 Hz and that of the real time clock is 200 Hz. The resolution of the events is determined by these figures. Some precise actions need to be activated through hardware timers.

4.2.7 Internal Structure of the 80960SA processor.

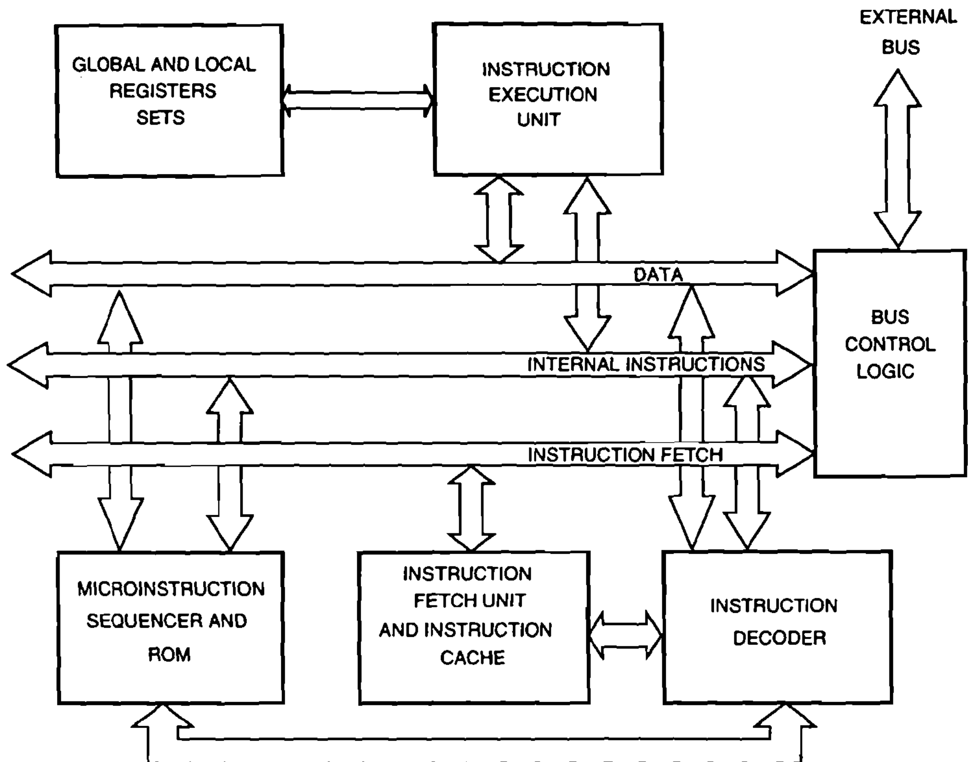


Figure 5.9: 80960SA, block diagram.

The i960 architecture defines several mechanisms for increasing performance through the use of pipelining and parallel execution of instructions. The 80960SA processor is composed by five major functional units. These units are described below:

- Bus Control Logic (BCL).

The Bus Control logic provides the interface between the processor and the external world. It accepts requests from other units. It attempts to maximize bus access efficiency through buffering ad burst accesses.

It provides a queuing mechanism that can buffer up to three outstanding requests at any given time. This allows other internal units to continue operation without waiting for bus requests to be completed. As a result, the execution of most memory reference instructions require little delay in the instruction execution pipeline.

The BCL generates burst cycles on the external bus, which allow 1-, 2-, 4-, 8-, 12-, or 16-bytes of read or written in a single operation. Instructions can be fetch in 16-byte bursts.

- Instruction Fetch Unit (IFU) and Instruction Cache.

The IFU acts as an intelligent buffer for the Instruction Decoder. The IFU contains a 512 byte, direct mapped instruction cache. While the other units in the processor are executing instructions, the IFU looks ahead in the flow of instructions stored in the

instruction cache, if a cache miss is detected, the IFU issues a prefetch to the BCL. Upon receiving the requested instruction, the IFU updates the instruction cache. The fetch and load might take place before the ID requires the instruction cache. The major exception to this rule happens on branch conditions.

The ID informs the IFU of any branch operations that are about to take place; the IFU checks for a cache hit, if the instruction is not present, the IFU begins fetching instructions for the new control path. To further minimize delays in the instruction pipeline, the ID sends a special signal to the IFU whenever instructions are required immediately. The IFU then passes the fetched instruction directly, rather than writing them back to the cache and reading them back again (*instruction-cache bypassing*).

The Instruction Pointer (IP) register in the processor and the IFU maintain several instruction pointers. These pointers point to instructions at various stages of the fetch-decode-execute pipeline.

- Instruction Decoder (ID).

The ID decodes the instructions it receives from the IFU and routes them to the appropriate execution units

The ID decodes the so called simple instructions (shift; integer add and substract; ordinal add and substract), and passes them to the Instruction Execution Unit (IEU), where they are executed, usually in a single clock period.

The ID executes branch instructions directly. If the branches are unconditional, no interaction with the other execution units is required. On conditional branches, the ID uses a condition code *scoreboard* (various mechanisms within the processor can be marked as in use). When the ID prepares to execute a conditional branch, it checks the condition code scoreboard, if it is clear, the ID signals the IFU immediately if a change in program is about to happen.

The complex instructions (those that are executed using one or more microcode instructions: *flushreg* (flush local registers), *mark and fmark*, are decoded by the ID and forwarded to the Micro-Instruction Sequencer (MIS). The MIS sends the equivalent microcode to the IEU.

Load and store instructions are sent directly to the BCL, the ID is responsible for converting the address information encoded in load, store, branch and call instructions into effective memory addresses. These instructions are executed by the ID or the BCL; this preserves the pipeline.

- Micro-Instruction Sequencer (MIS) and ROM.

When the ID receives a complex instruction, the MIS supplies the microcode to the execution unit for that instruction, which can be the IEU or BCL. The MIS also supplies microcode for the power-up and self-test sequences, which are performed during processor initialization.

- Instruction Execution Unit (IEU).

It contains the ALU and the mechanism for register and condition code scoreboarding. It also manages the 16 global registers and the 4 sets of 16 local registers.

The IEU handles the reading and writing of global and local registers. It also handles the allocation of local register sets on procedure calls. The EIU allocates a new set of local registers on each procedure call. If all register sets become allocated, the IEU automatically flushes the oldest frame to the stack on the next procedure call. The IEU retrieves automatically any local register frame from the stack when required by a return operation.

The *register scoreboard* provides scoreboarding for the local and global registers: when one or more registers are being used in an operation, they are marked in use. The register scoreboard allows the processor to continue executing subsequent in-

structions, as long as those instructions do not require the contents of the score-boarded registers.

One feature of the IEU that enhances instruction-execution performance is *register bypassing.* This is a mechanism that allows an instruction that would ordinarily require source operands to be placed in registers to be executed without accessing one or both of the source registers. This can be achieved in two situations: When the IEU executes an instruction with two source operands and one or both of the operands are literals (5 bits). Register bypassing will also occur when the second of two source operands is the result of the previous instruction. The net result of register bypassing is the saving of one clock cycle. Most instructions that the IEU executes can be executed in a single cycle when register bypassing occurs.

4.3 The Interprocessor Communication.

A communication medium for the three processors, which operates at a high transmission rate should be defined. An important condition is the fact that the performance of the processors should not suffer a considerable deterioration. A balanced communication medium is therefore one which is isolated from the buses of the different processors and is fairly granted to any of them. A communication ram, whose bus is independent to the local buses satisfies this requirement. The communication memory arbiter needs to comply with another requirement:

- Data present in memory should be unambiguously interpreted by the processors at any time. In order words, the validity of the messages or data types that involve more than one memory access should be protected.

One way to solve this problem is by the use of semaphores. This mechanism involves read modify write accesses to control and set a flag. The 80196KC processor do not support this kind of operations, and in consequence a hardware solution would be necessary.

The best way to satisfy this requirement is that the processors agree upon the maximum size of the data types and implement an arbitration mechanism that grants the bus until the maximum amount of bytes has been transferred. In order to calculate the effect of the maximum data size to the transfer rate and the stall time of the processors, the following arbitration mechanism is assumed:

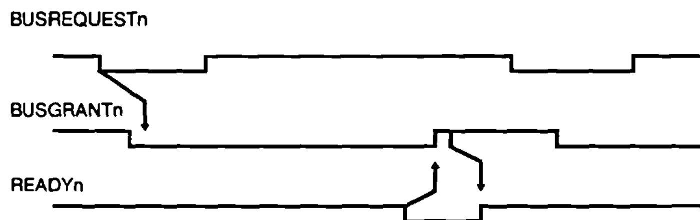


Figure 4.10: Arbitration Mechanism.

1) When a processor wants to access the communication memory, it will request it by activating BUSREQUESTn.

2. If no other device is performing an access, the communication bus is granted (BUSGRANTn active).

3. The processor will relinquish the bus and activate READYn. This will allow another device to perform an access.

The data bus width of the 80960SA is 16 bits. This processor does not have a dynamic bus sizing mechanism, which implies that the communication memory should also be 16 bits. The 80196KC can operate with a data bus of 8- or 16-bits. The 80960SA can access 8 consecutive memory locations within the same machine cycle in the so called burst mode, while the 80196KC can only access one location per machine cycle.

The machine cycle time of the main processor as a function of the amount of wait states and the number of consecutive accesses is [13]:

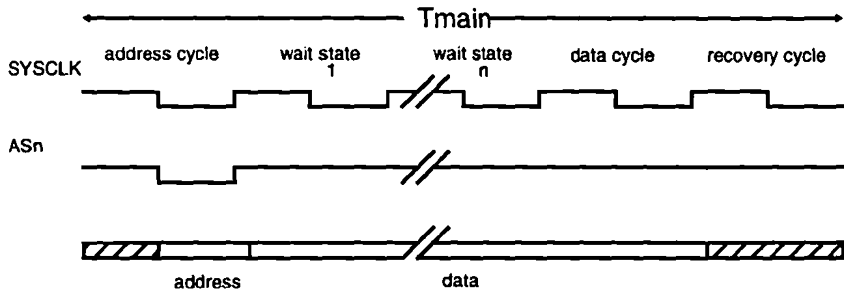


Figure 4.11: 80960SA, machine cycle.

Tmain(M) = (NxM + M + 2) x Tclk

N = amount of wait states to accommodate standard rams = 2 (if Tclk = 62.5 ns).

M = amount of consecutive accesses during one machine cycle

The machine cycle time of the 80196KC as a function of the amount of wait states is [14]:

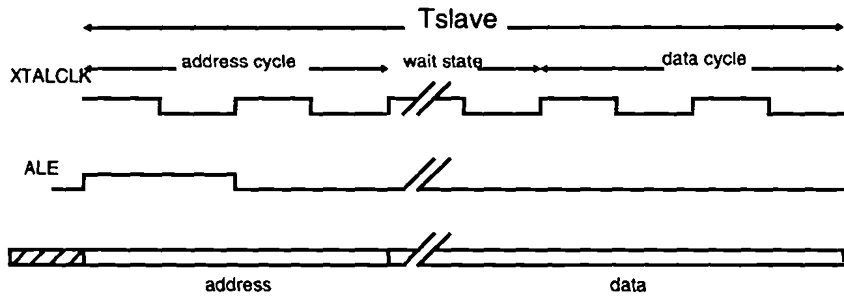


Figure 4.12: 80196, machine cycle.

tslave = (2xN + 4) x Tclk

N = amount of wait states to accommodate standard rams = 1 (if Tclk = 62.5 ns).

Assuming that the arbiter clock has the same frequency as the reference clock of the processors, the arbitration and synchronization introduce the following amount of wait states:

Synchronization of request signal	- 1 cycle
Arbiter change of state	- 1 cycle
Synchronization of grant signal	- 1 cycle (80960SA))
	- 2 cycles (80196KC)
	(Tclk = 2 x Txtalclk)
Synchronization of ready signal	- 1 cycle

Toverhead_main = 4 Tclk

Toverhead_slave = 5 Tclk

The stall time as a function of the amount of allowed accesses is:

Tstall_main(M) = (2xToverhead_slave + 2x Tslave) x M;

Tstall_slave(M) = (Toverhead_slave + Tslave) x M + Tmain(M)

+ Toverhead_main;

The access time as a function of the amount of allowed accesses is:

Tacc_main(M) = Tstall_main(M) + Tmain(M);

Tacc_slave(M)= Tstall_slave(M) + MxTslave;

The effect of the amount of allowed states is summarized in the following table:

Table 1: Effect of amount of accesses

M	MxTslave	Tmain(M)	Tstall_ slave(M)	Tstall_ main(M)	Tacc_ slave(M)	Tacc_ main(M)
	expressed in amount of clock periods #Tclk				#Tclk - Mbyte/sec (fclk = 16 MHz)	
1	6	5	16	22	22-1.45	27-1.18
2	12	8	30	44	42-1.52	52-1.23
4	24	14	58	88	82-1.56	102-1.25
8	48	26	114	176	162-1.58	202-1.26

The transfer rate (access time) remains almost constant and the stall time increases linearly, its effect per byte is slightly less notorious for the slave processors at high values of M. Increasing M does not contribute in a big extent to the transfer rate. By choosing M = 2 almost all the data types may be supported (max. 4 bytes). Furthermore by choosing M low, the complexity and price of the arbiter will be kept low.

A problem remains to be solved: the fairness of the arbitration. None of the participants of the communication memory should get priority over the others, in order to prevent high stagnation of one of the low priority processors. In a VME solution for instance, fair bus requesters are implemented, that do not request the bus after an own transaction has taken place and other requests are pending. A VME configuration has a daisy chain grant mechanism, that even if a single priority is used, the boards located far from the arbiter have an intrinsic low priority. This justifies the fairness to be implemented on the requester. As far as this application is concern, all the processors will generate its own request signal, this way, the arbiter can decide whether or not to grant the bus to a device. An arbiter that polls the requests of the processors has the properties we are looking for.

The arbiter should have the following characteristics:

- The arbiter should allow the slave processors to execute two consecutive accesses.
- The main processor may acquire the bus only for one machine cycle.
- Because of the fact that the amount of software code of the slave processors will be low (less than 64 KByte) compared to that of the main processor (more than 3 MByte if all the different languages for the messages of the user interface are supported), the software code of the Scanner and ADF controllers may be stored in the main processor eproms. During initialization, the code could be down loaded. This approach eases the replacement of the eprom modules and makes down load of different kinds of software possible for service diagnosis and testing. This operation can take place while the controllers are in the reset situation, because the buses float or have a high impedance. A dma access during the normal operation of the processors

will not be implemented because it does not have practical applications and in case of a dma access is requested while the slave processor tries to communicate via the share memory, the dma request would have to be postponed until the slave processor access has taken place. This would increase the complexity of the hardware.

The architecture of the shared memory looks as follows:

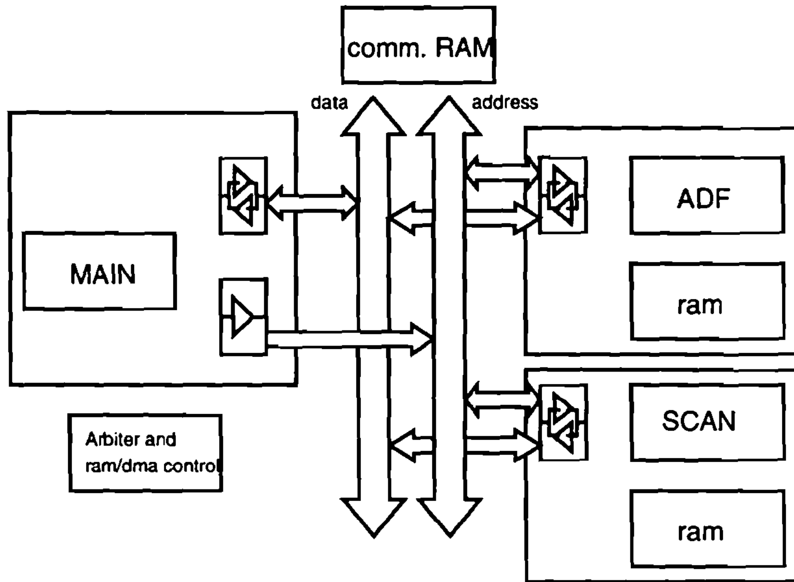


Figure 4.12: The communication memory interface.

The state diagram of the arbiter is depicted below:

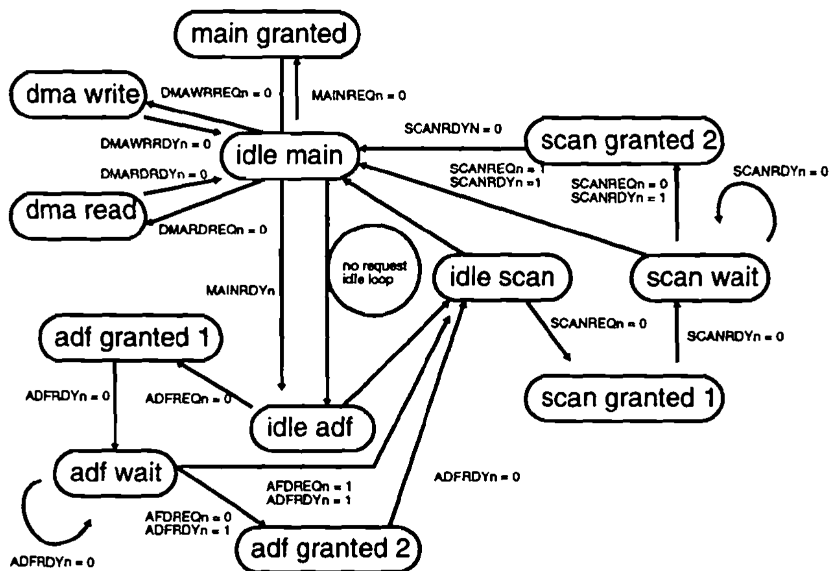


Figure 4.13: The communication memory arbiter (state diagram).

The implementation of the arbiter and communication memory/dma controller has been realized by means of Generic Array Logic (GAL) components, the hardware description listings are shown in appendix A1.

4.4 The Eprom Interface.

The 80960SA provides a 512-byte cache for instructions. When the processor fetches an instruction or group of instructions from memory, they are stored in this cache before being fed into the instruction execution pipeline. In order to load the cache at a high rate, the processor can access 8 consecutive memory locations within one machine cycle (burst mode). If we want to take advantage of this feature of the processor, a mechanism should be found to be able to perform high speed accesses even in combination with standard commercial eproms. The 80960SA operating at 16 MHz has the following timing for a burst cycle of two accesses:

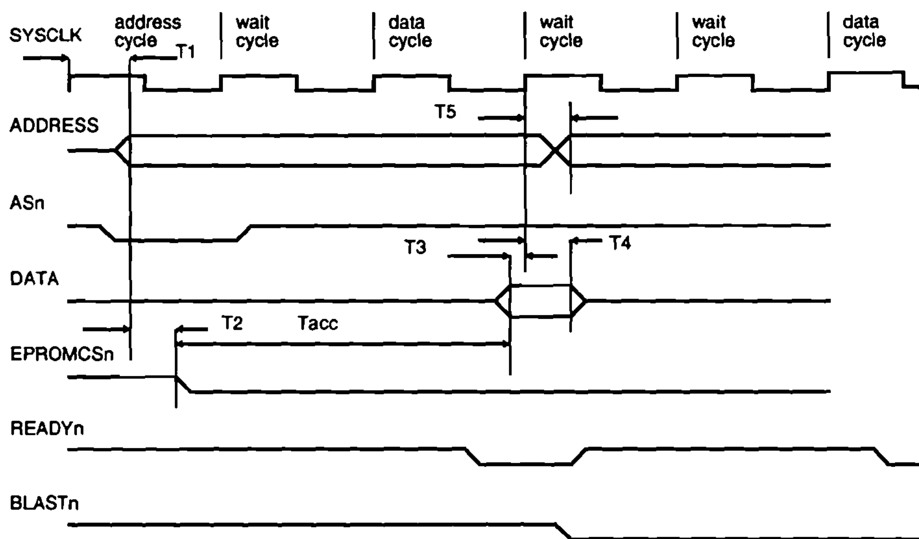


Figure 4.14: Eprom access.

During the first cycle, the access is determined by the last generated signal among EPROMCSn and the less significant address lines that are latched from the multiplexed data/address bus. If the chip select is generated by means of a 15 ns GAL, and a commercial 120 ns eprom is used, the amount of wait states during the first cycle is calculated below (the timing parameters of the processor can be found in ref. [13]):

T1 = T5 = 25 ns.

T2 = 15 ns.

T3 = 10 ns.

Tacc = 120 ns.

Tclk = 62.5 ns.

Wait_states1 = (T1 + T2 + T3 + Tacc)/Tclk -2

= 170ns/62.5ns - 2= 0.72 wait states

1 wait states is necessary.

The amount of wait states during the following accesses is determined by the low address lines that specify the memory location during the burst cycle [A1,A3]:

Wait_states2 = (T5 + Tacc + T3)/Tclk -1 = 1.48 wait states.

2 wait states are necessary.

The amount of wait states can be reduced by defining an eprom array of 32 bits and making use of two interleaved banks. While the processor is accessing one bank, the access of the other bank is started by an eprom_control interface.

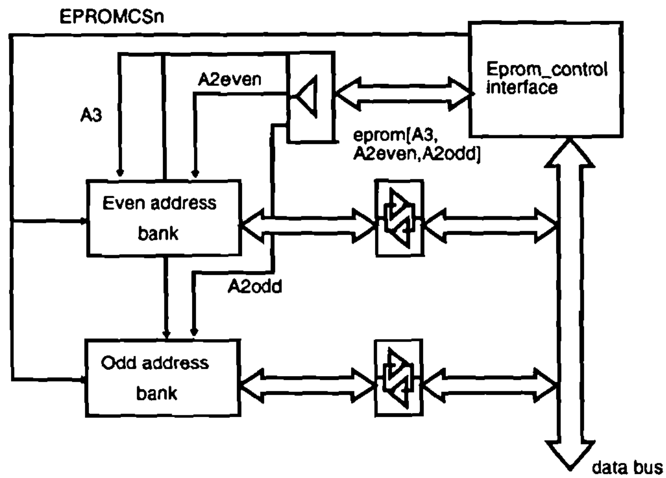


Figure 4.15: Interleaved eprom interface.

The eprom_controller interface generates the two least significant address lines and enables the buffers of the proper bank at the correct moment. The following figure illustrates this:

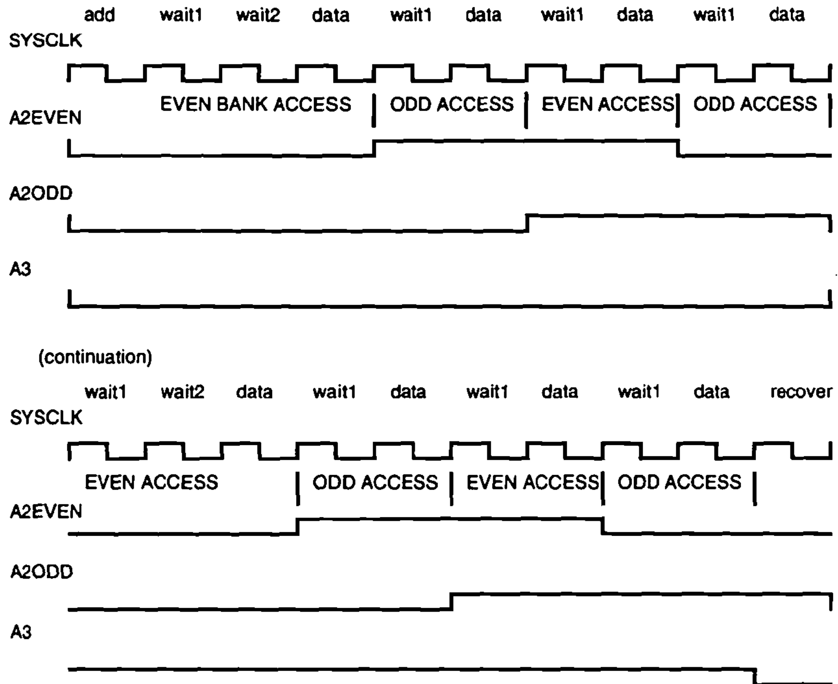


Figure 4.16: Interleaved eprom access.

The state machines were implemented also in GAL's, see appendix A2.

Note that transceivers were used in order to be able to make use of Flash Eproms in the future. This has influenced the amount of wait states, along with the fact that the state machine that produces the RDYn signal to the processor is also used for the other memory components. The average amount of states is 1.25 wait_states/access. Without this configuration, the amount of wait states would have been 1.875 wait_states/access.

4.5 The RAM interface.

Standard static ram devices are used in this application. This avoids the use of refresh circuitries that are required by dynamic rams. Furthermore, the low power requirements for battery back up are met. The amount of wait states is 2, this is necessary to compensate the propagation delay introduced by the protection circuit (which prevents accesses to take place during power down). The accesses to ram are controlled by means of the same state machine of the rom interface.

An interleaved configuration makes the hardware very complex and expensive, due to the selection of the proper byte during write cycles.

This choice might have strong implications on the behaviour of the processor during context switching. The amount of clock cycles as a function of the amount of accesses (M) during one machine cycle is given by:

$$Tmain(M) = (3xM + 2) \times Tclk$$

4.6 The hardware system architecture.

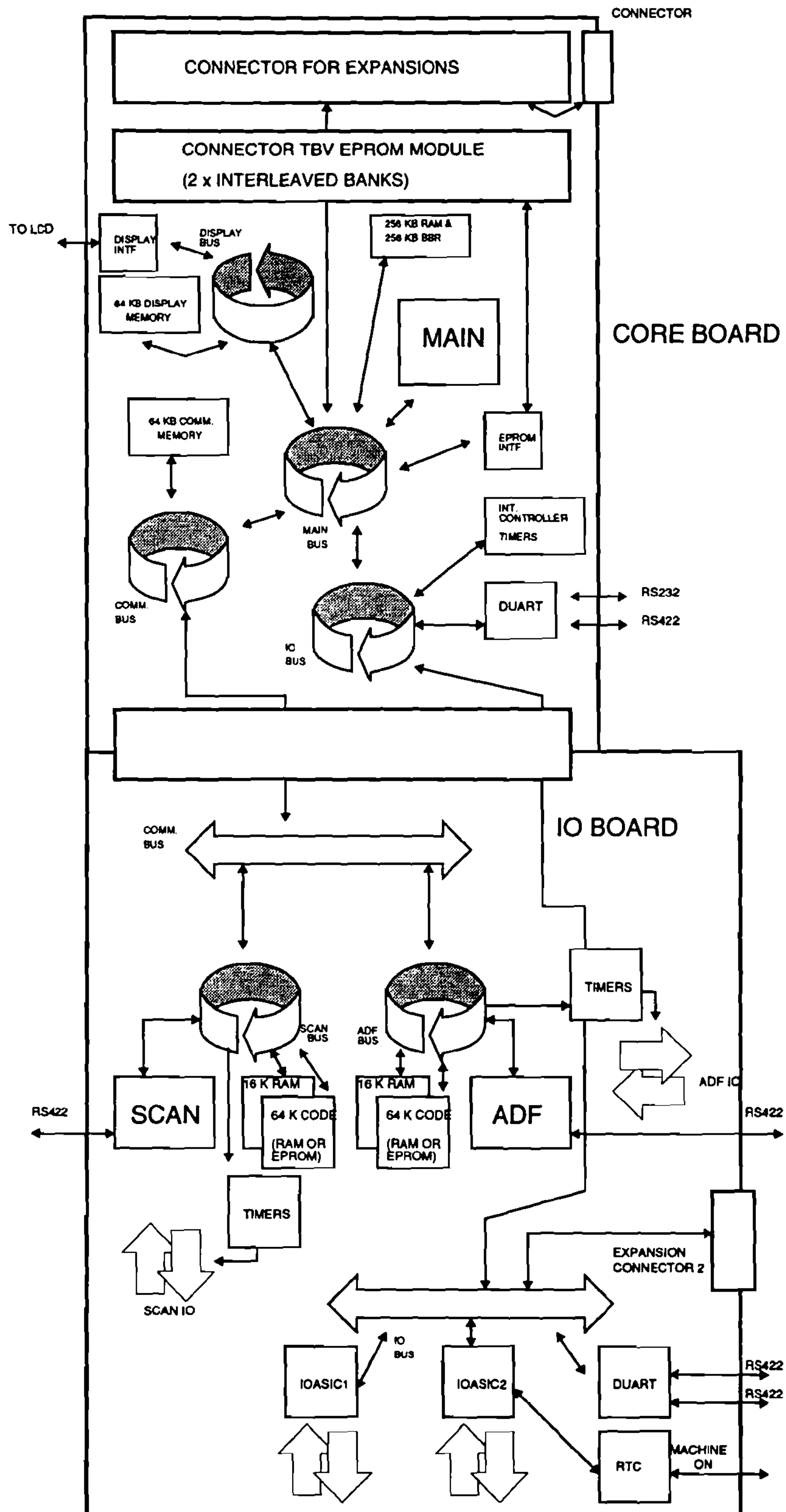


Figure 4.17: The hardware board architecture.

The previous figure shows the final implementation of the hardware. Note that two boards are used:

- The Core Board.

It contains the main processor, the communication memory, the display interface and the peripherals of the processor. By separating this board from the i/o functions (including the ADF/Scanner processors) a possible migration to other main processor can take place. At the core board an i/o bus is implemented to access the i/o devices located on the i/o board as well as the extension connector which has, besides an 8-bits i/o-data bus, a couple of chip select lines, an interrupt line and the four least significant address lines. This way, devices of low complexity can be added. The amount of wait states for this eventual devices can be programmed in one GAL located at the core board. There is also a connector available for expansions that would require a large amount of address lines (memory components).

- The I/O board.

This board contains the i/o devices. The Scanner and the ADF processors are tightly coupled to the main processor by means of the communication bus. One can decide whether to make use of EPROM's or to let the main processor down load the program code at dma basis through the communication bus.

This solution has so far proven to comply the requirements of the copier and the flexibility items in the design (expansion connectors) have been used to adapt some functions; one example: The control of one motor of the ADF function needed a 16 bits up/down counter as well as its related logic that generates the control signals from the phase of the rotary encoder that is coupled to the motor. The temporary solution was a module with such a counter that has been installed at the expansion connector of the i/o board; the value of the counter was retrieved by the main processor and stored in the communication memory. The ADF processor could access later on the communication memory and get the value of the counter.

5 Implications of using a RISC processor for the control of a digital copier.

RISC processors were intended to be used in applications where a great deal of data processing takes place. In this kind of applications, the processor remains in a certain routine for a long period of time until the necessary result is found. Some embedded applications show this behaviour, e.g. the description language interpreter of a laser printer. The RISC architecture becomes more and more popular due to its attractive features such as high speed.

The architecture of the processor can affect the behaviour of the software as well as its development. In the following paragraphs an investigation is done of the possible problems that the 80960 RISC processor introduces in this real time application.

5.1 Context Switching.

One of the factors which makes the RISC fast is the large number of internal registers. They enable the RISC processor to perform many operations internally, hence reduce the access time to the external memory. However, the Berkeley-RISC (sparc) and Standford-RISC (MIPS) architectures for instance, do not provide enough support for task switching. This is also the case for the 80960SA processor. The window structure of the Berkeley-RISC as well as the register organization of the 80960 makes them suitable for procedure handling, but if a context switch takes place, a great number of register contents should be moved to/from the processor. These processors make in general use of standard cisc techniques in saving/loading the task state.

In real time applications there are a lot of interactions with the external agents. Furthermore, in multitasking solutions, several tasks require processor services at the same time; for every task a stack is needed. The task state is stored in the stack when the task relinquishes the processor to other task.

Context switches can take two forms: synchronous and asynchronous. The first one occurs when the task voluntarily gives up the cpu for another task to run. An asynchronous context switch occurs when the task has to unexpectedly give up its cpu time, this usually happens as a result of an interrupt. In the last case the running task may or not get continued during interrupt, depending on its priority. Preemption takes place if a task is temporary suspended, for a higher priority task.

5.1.1 Asynchronous context switching.

We are going to examine the different asynchronous control transitions that take place upon receiving an interrupt. For this purpose, the first version of a real time multitasking kernel is going to be used. This kernel has been ported from an existing one and even though it needs to be optimized, it constitutes a valuable tool that allows us to analyse the behaviour of the processor. The kernel has, as far as the hardware related items are concerned, the following characteristics:

a) The time reference interrupt (or real time clock interrupt-RTCI) and master belt position interrupt (or master pulse interrupt-MPI) force the running task to relinquish the cpu to the scheduler, if preemption should take place, otherwise the interrupted task is resumed. If a higher priority task gets the cpu, it might be also preempted or it finishes its operation and gives up voluntarily the cpu. The scheduler will then reactivate the preempted task.

b) The other interrupts produce signals to the scheduler. The scheduler makes use of this information in order to activate the relevant tasks later on. The scheduler gets control of the cpu after the interrupted task completes operation.

c) The kernel has its own stack. However, this stack will only be used if the scheduler happens to be interrupted. For the other control transitions, the stack of the previous activated task is used. This approach limits the number of stack switchings and avoids eventual problems that might occur if the processor resumes its operation with invalid stacks.

d) The processor will operate constantly in the interrupt mode. This will ensure the robustness of the kernel.

The *latency time* is defined here as the time that elapses from the moment that one interrupt arrives until one task becomes operative. This definition covers the case that one task is preempted for the sake of a higher priority one, as well as the case that the scheduler is interrupted, and because of the interrupt, one task acquires a higher priority and is scheduled.

The latency time increases when a higher priority interrupt arrives than the one being serviced or by posting an equal or lower priority interrupt during an interrupt service routine.

In case a RTCI is being serviced and a MPI is requested, this last one will be immediately nested but no preemption takes place. The program will return to the RTCI service routine, which will eventually cause preemption, if the scheduler was not interrupted. In this particular case of preemption, the scheduler has the information of both interrupts in order to activate the proper task.

If the program is executing a MPI service routine and a TRCI arrives, this last one will be posted. The program will execute the TRCI service routine as soon as the scheduler performs a return from interrupt. The cpu handles the pending interrupt as if it has just arrived, however, the "latency" will be slightly lower. Note that the actual latency time with respect to the moment that the interrupt arrived may vary considerably due to preemption.

It is not allowed that a RTCI is posted if the cpu is processing a RTCI service routine (the same applies for the MPI) because it would mean that the time (or position) reference has been lost.

Multilple Peripheral Interrupt Controller (PIC) interrupts will cause naturally one interrupt being posted per interrupt service routine.

All the processor activity that is not related to the application software is called *overhead.* The software administration (the kernel), and the hardware characteristics (cpu-architecture, amount of wait states for external accesses, clock frequency) are responsible for both latency an overhead.

A calculation of the cpu contribution to the latency time follows. We are going to use these results to make later an estimation of the right balance between the task processing time and the overhead.

When a RTCI or a MPI arrives, three different situations can occur:

1) The scheduler is interrupted. This will only cause a restart of the scheduling mechanism in order to reinitiate the cpu-granting priorities. Then a waiting task can be activated.

2) A task was running when the interrupt arrived. The interrupt service routine gives the cpu to the scheduler, which may decide that the interrupted task should be preempted for the sake of a task with higher priority.

3) A RTCI service routine was running (or because of a RTCI-preemption one task was running) when a MPI arrives. The MPI service routine is nested. After the service routine notifies that a MPI has taken place to the scheduler (through increasing the MPI-counter), the program returns to the preempting task or to the RTCI service routine, in which case it will then call the scheduler for an eventual preemption. Nesting of RTCIs or MPIs within the other Peripheral Interrupt Controller interrupts result in immediate return to the previous interrupt routine.

These cases are depicted in the following figure:

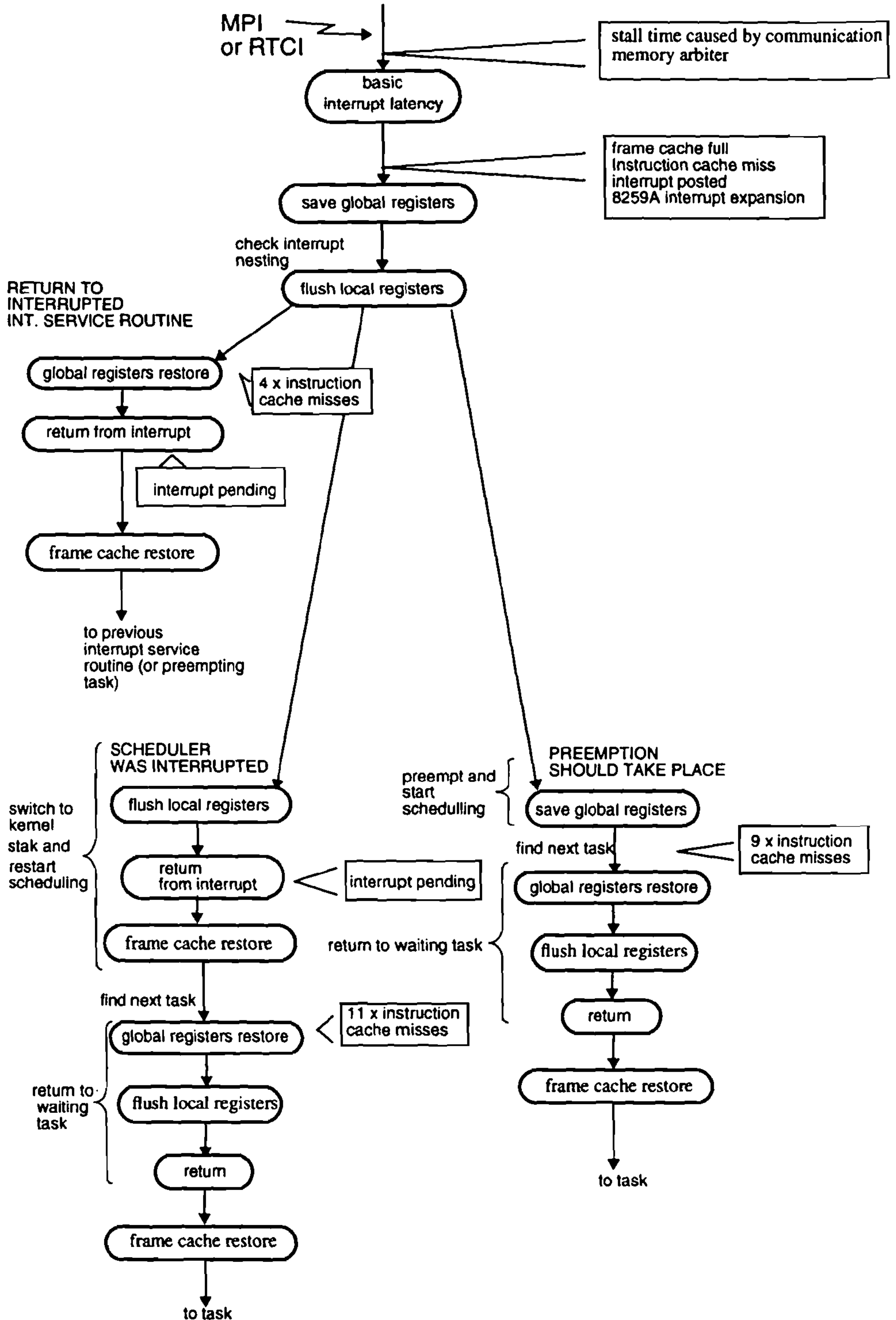


Figure 5.1: Asynchronous context switching (due to RTCI or MPI).

Possible expansions to the latency time are indicated by square blocks. The amount of cache misses, that is indicated in the figure gives an indication of the procedure nesting.

The timing calculations that follow are approximations, because of the internal parallelism of the processor; the introduced fault is marginal if we take into consideration that the timing is mainly determined by the external accesses. The ram accesses are performed with 2 wait states, and those to the eprom in a sequence of 2-1-1-1-2-1-1-1 wait states. The access times can be expressed as follows:

Access_time_ram = 2 + 3 x M ; M = amount of accesses = [1,8]

Access_time_eprom = 2 + 3 x M1 + 2 x M2 + 3 x M3 + 2 x M4

M = M1 + M2 + M3 + M4 = amount of access = [1,8]

M1 = 1

M2 = [1,3] if M>1

M3 = 1 if M>4

M4 = [1,3] if M>5

- Basic latency: Interrupt service routine in cache, Master pulse interrupt.

  Hardware recognition: 4 cycles

  Stop current instruction flow assuming a RISC instruction: 4 cycles

  Determine next IP and save:
  This value is taken from the instruction cache, afterwards 2 external accesses (M=2) are necessary to store this value onto the stack (at the location that corresponds to local register r2. The reference manual specifies 8 cycles for zero wait states. In case of 2 wait states: t = 8 + 2 x M= 12 cycles.

  Read interrupt vector number: 18 cycles.

  Check interrupt priority: 8 cycles.

  Read Interrupt table pointer (2 external accesses M = 2):
  15 cycles are specified in the reference manual [2] for zero wait states. In case of 2 wait states t = 15 + 2 x M= 19 cycles

  Check if processor already interrupted: 6 cycles.

- Current process in interrupt:
  The reference manual specifies 14 cycles. We can assume that no external accesses are performed. The processor might need this time to re-establish the local register allocation mechanism.

  Save process control and write interrupt record (M = 5 external accesses). 14 cycles are specified [2] for zero wait states. In case of 2 wait states t = 14 + 2 x M = 24 cycles

Compute interrupt record address of new local register set: 10 cycles.

Allocate new local register set: 3 cycles.

Fetch new instruction and start decoding: 2 cycles.

----

124 cycles

- Check if processor in interrupt mode: 6 cycles
- Save global registers:

To save the global registers the following sequence of instructions is assumed:

t = eff. add. calculation + external access + internal working

lda	64(sp),sp	M = 2; t = 2 + 8 + 1= 12 cycles.
stq	g0,64(fp)	M = 8; t = 2 + 26 + 1 = 29 cycles.
stq	g4,80(fp)	M = 8; t = 2 + 26 + 1 = 29 cycles.
stq	g8,96(fp)	M = 8; t = 2 + 26 + 1 = 29 cycles.
stt	g12,112(fp)	M = 6; t = 2 + 20 + 1 = 23 cycles.

----

123 cycles

- Return from interrupt:

The processor copies the arithmetic- and process-control information from the interrupt record into its internal registers, and then it would return to the execute state. In this application the processor does not perform this transition and remains in the interrupted mode. In consequence no stack switching takes place and the local register frame does not necessarily need to be restored from memory upon return from interrupt.

In the Intel reference manual [2], a typical time of 80 cycles is specified for the return from interrupt instruction. This is equal to the time it would be necessary to read and copy the interrupt record into the internal registers + the necessary time to restore the local registers frame and deallocate the interrupt frame that was used by the interrupt service routine. Approximately 40 cycles are required for restoring the local registers frame with zero wait states ram. 40 cycles remain to retrieve the information of the two control registers and deallocate the local register frame of the interrupt service routine.

With two wait states ram access, 8 cycles should be added to this latency time:

t = 40 + 8 = 48 cycles.

- Flush local registers: 3 frames = 312 cycles.

- Return from procedure: 7 cycles.

- frame cache restore:

This involves the restitution of the local registers that were stored on the stack (1 frame) upon a procedure call. On returning to the original procedure, a restitution of the local registers is necessary if the local registers have been flushed or another task has become active: 104 cycles.

- Global registers restore:

To load the global registers the following sequence of instructions is assumed (g0 contains the address of the first global register on the stack):

t = eff. add. calculation + external access + internal working

```
mov    g0, r3          t = 0 + 0 + 1= 1 cycles.
ldq    64(r3),g0       M = 8; t = 2 + 26 + 1 = 29 cycles.
ldq    80(r3),g4       M = 8; t = 2 + 26 + 1 = 29 cycles.
ldq    96(r3),g8       M = 8; t = 2 + 26 + 1 = 29 cycles.
ldq    112(r3),g12     M = 6; t = 2 + 20 + 1 = 23 cycles.
                       -----------------------------------
                                  111 cycles
```

- Processor stall time due to communication ram arbiter: 44 cycles.

- Frame cache full:

A frame cache full occurs when more than 4 procedures are nested, the oldest assigned register set is stored into ram to make place for the new procedure. The time required to perform this mechanism is almost equal to the amount of external accesses to write one frame into memory: 104 cycles.

- Instruction cache miss:

The processor performs 2 accesses to the eprom during one machine cycle =

M1 =1 , M2 = 1; t = 7 cycles

- Posting a pending interrupt:

This includes a comparison between the processor priority and that of the incoming interrupt. Two atomic read/write operations are performed to the pending priorities and pending interrupt records at the header of the Interrupt Table. The effect of posting one interrupt is an addition of 67 clock cycles [2], if zero wait states ram accesses are considered. In case of 2 wait states this time become: 67 + 8 = 73 cycles.

- 8259A interrupt expansion:

The interrupt controller interface is implemented with 6 wait states; during an interrupt acknowledge cycle, two accesses are performed to the interrupt controller. between both accesses, the processor introduces 5 idle cycles to compensate the timing of the controller:

t = 5(idle-cycles) + 2 x (2 + (amount_of_wait_states + 1)) = 23 cycles

- Upon returning from interrupt, the processor detects an interrupt pending.

When the processor finds a pending interrupt, it handles it as if it has just received the interrupt. According to the reference manual no latency is introduced. However, during the interrupt return execution the processor performs a check to the interrupt

pending fields, updates these fields accordingly, and the interrupt record is built up on the stack.

The reference manual specifies an effect of the pending interrupt to the return to 67 clock cycles if zero wait states ram accesses are implemented. To read and modify the interrupt priorities word and the appropriate pending interrupts word 4 accesses are needed. The interrupt record built up requires 5 accesses. For 2 wait states, the effect of the interrupt pending is: 67 + 2 x (4 + 5) = 85 cycles.

The latency time contributions for the three different situations are summarized in the following tables. The third column indicates the contribution of the cpu to the latency time if a second (pending) interrupt is serviced upon completion of the first one.

Table 2: CPU contribution to the latency time, asynchronous context switching, task is preempted

Processor activity	best case	worst case	Maximum contribution to second int.
	[# 16 MHz cycles]		
Stall time caused by comm. memory arbiter		44	
Interrupt basic latency	124	124	
8259A interrupt expansion		23	23
Frame cache full		104	104
Instruction cache miss		7	7
Interrupt posted		73	
Save global registers x 2	246	246	246
Instruction cache miss x 10		70	70
Restore global registers	111	111	111
Flush local registers x 2	624	624	624
Return	7	7	7
Frame cache restore	104	104	107
TOTAL - TA1	1216 cycles 76 usec	1537 cycles 96 usec	1299 cycles 81.1 usec

Table 3: CPU contribution to the latency time, asynchronous context switching, scheduler is interrupted

Processor activity	best case	worst case	Maximum contribution to 2nd int.
	[# 16 MHz cycles]		
Stall time caused by comm. memory arbiter		44	
Interrupt basic latency	124	124	
8259A interrupt expansion		23	23
Frame cache full		104	104
Instruction cache miss		7	7
Interrupt posted		73	
Save global registers	123	123	123
Instruction cache miss x 12		84	84
Flush local registers 3	936	936	936
Return from interrupt	48	48	48
A pending interrupt is detected during return		85	
Frame cache restore x 2	208	208	208
Return	7	7	7
TOTAL - TA2	1446 cycles 90.37 usec	1866 cycles 116.6 usec	1540 cycles 96.2 usec

Table 4: CPU contribution to the latency time, asynchronous context switching, nested interrupt.

Processor activity	best case	worst case	Maximum contribution to 2nd int.
	[# 16 MHz cycles]		
Stall time caused by comm. memory arbiter		44	
Interrupt basic latency	124	124	
8259A interrupt expansion		23	23
Frame cache full		104	104
Instruction cache miss		7	7
Interrupt posted		73	73

Table 4: CPU contribution to the latency time, asynchronous context switching, nested interrupt.

Processor activity	best case	worst case	Maximum contribution to 2nd int.
	[# 16 MHz cycles]		
Save global registers	123	123	123
Flush local registers	312	312	312
Restore global registers	111	111	111
Instruction cache miss x 4		28	28
Return from interrupt	48	48	48
A pending interrupt is detected during return		85	
Restore frame cache	104	104	104
TOTAL - TA3	822 cycles 51.37 usec	1186 cycles 74.1 usec	933 cycles 58.3 usec

The results of table 4 should be added to those of table 2, 3 or 5 depending on the state of the scheduler and the interrupt being serviced.

The PIC interrupts (except RTCI) do not pass the control to the kernel. Instead they produce signals to the scheduler. Hereafter they return to the interrupted program.

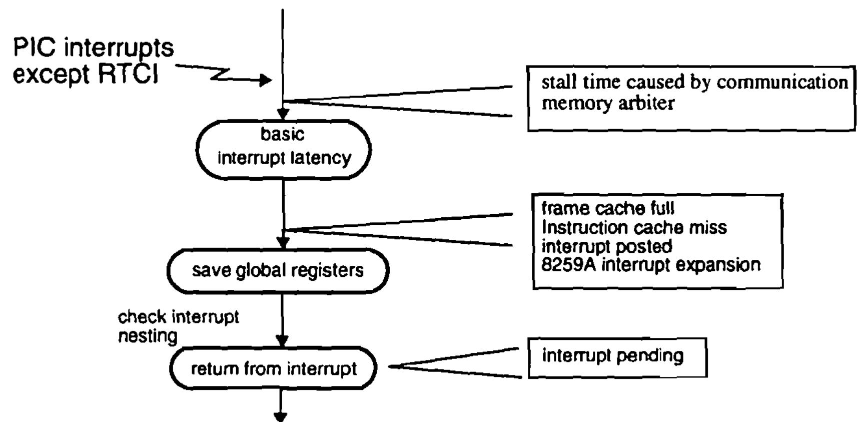


Figure 5.2: Asynchronous context switching (due to PIC).

Table 5: CPU contribution to the latency time, asynchronous context switching, PIC interrupts (except RTCI)

Processor activity	best case	worst case	Maximum contribution to 2nd interrupt
	[# 16 MHz cycles]		
Stall time caused by comm. memory arbiter		44	
Interrupt basic latency	124	124	
8259A interrupt expansion	23	23	23
Frame cache full		104	104
Instruction cache miss		7	7
Interrupt posted		73	73
Save global registers	123	123	123
Restore global registers	111	111	111
Instruction cache miss x 2		28	28
Return from interrupt	48	48	48
A pending interrupt is detected during return		85	
Restore frame cache	104	104	104
TOTAL - TA4	533 cycles 33.3 usec	874 cycles 54.6 usec	621 cycles 38.8 usec

5.1.2 Synchronous context switching.

If one task needs an event in order to keep on processing (a period of time, the activation of an external line, or the output of another task), it gives up the cpu and becomes inactive. The scheduler will then activate a waiting task or a task that has been preempted by a higher priority one upon interrupt, if the conditions for its activation are valid. The program will not switch to the system stack. During the control transitions, the stack of the previous task is used.

The *latency time* is defined here as the time that is required to switch from one task to the other.

The contribution of the processor to the latency time is depicted in the following figure:

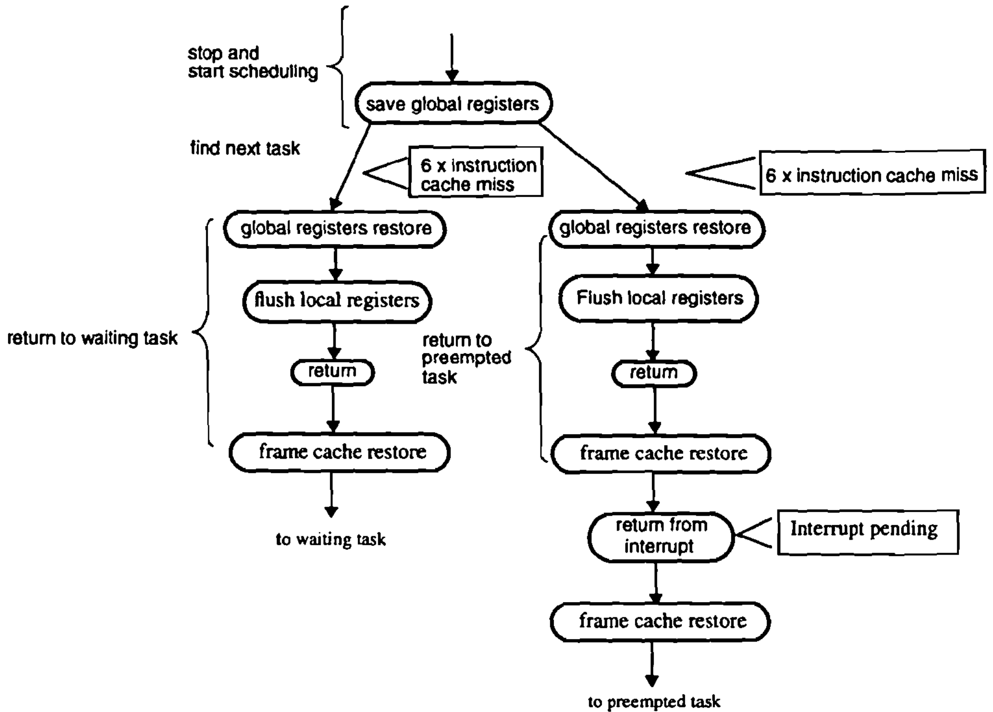


Figure 5.3: Synchronous context switching.

Table 6: CPU contribution to the latency time, synchronous context switching, a waiting task is activated

Processor activity	best case	worst case
	[# 16 MHz cycles]	
Save global registers	123	123
Instruction cache miss x 6		42
Restore global registers	111	111
Flush local registers	312	312
Return	7	7
Frame cache restore	104	104
TOTAL - TS1	657 cycles 41 usec	699 cycles 43.7 usec

Table 7: CPU contribution to the latency time, synchronous context switching, a preempted task is resumed

Processor activity	best case	worst case
	[# 16 MHz cycles]	
Save global registers	123	123
Instruction cache miss x 6		42
Restore global registers	111	111
Flush local registers	312	312
Return	7	7
Frame cache restore x 2	208	208
Return from interrupt	48	48
A pending interrupt is detected during return		85
TOTAL - TS2	809 cycles 50.5 usec	940 cycles 57.1 usec

5.1.3 Effects of the CPU to the context switching.

The results of the calculations of the latency times are summarized in the following table:

Table 8: Summary of the cpu contributions to the latency time

Type of context switching	best case	worst case	Maximum contribution to 2nd interrupt
	[usec]		
Asynchronous, task is preempted, TA1	76	96	81.1
Asynchronous, scheduler is interrupted, TA2	90.37	116.6	96.2
Asynchronous, nested interrupt, TA3 (*)	51.37	74.1	58.3
Asynchronous, PIC interrupts (except RTCI), TA4	33.3	54.6	38.8
Synchronous, a waiting task is activated, TS1	41	43.7	
Synchronous, a preempted task is activated TS2	50.5	58.7	

(*) This time should be added to the TA1, TA2 or TA4 if the interrupt is nested in the interrupt service routine of the respective cases.

When the machine is in stand-by the master belt stands still and in consequence no master pulse interrupts (MPI) are generated. The program performs polling tasks (to

the keyboard, the security sensors, etc.) and controls the temperature of the paper trajectory. Only the real time clock interrupt is generated.

It is very likely that the scheduler will be waiting for an event to occur when a RTCI arrives. One task will then be asynchronous scheduled. Subsequent tasks may be activated afterwards as a result of a signal generated by the first task, or because the conditions for their activation are valid but these tasks have a lower priority than the task that has been activated first. This situation is depicted in the following figure:

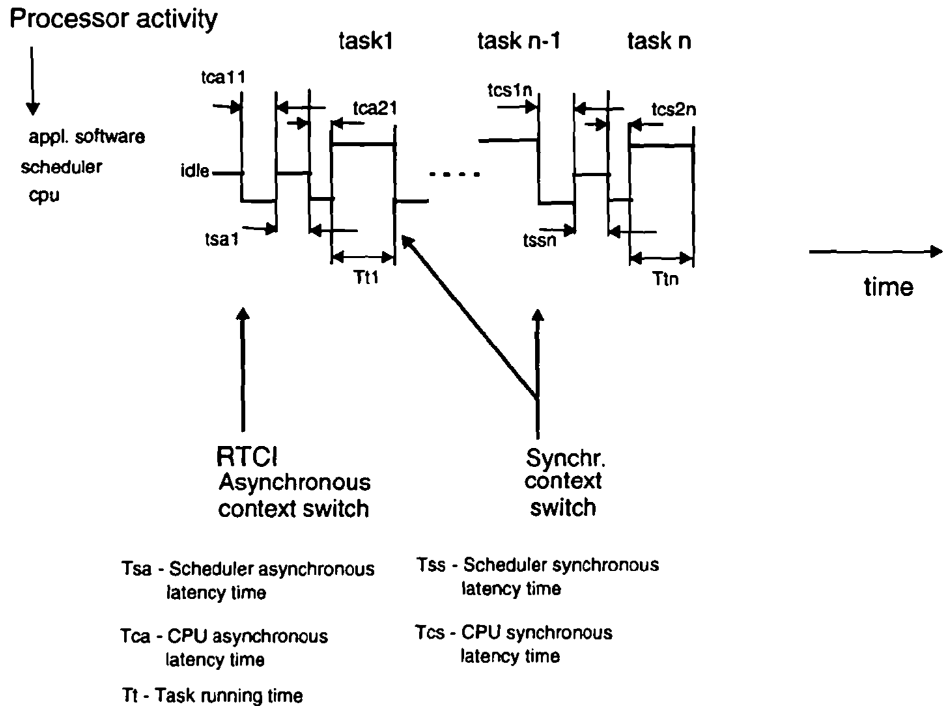


Figure 6.4: Context switching during stand-by.

In the figure above, only interrupts from the real time clock are received. If a MPI arrives, two situations may occur: two MPIs take place within 2 RTCIs (none, one or both will cause preemption), or a MPI-service routine is being processed when a RTCI is requested; in this situation the RTCI will be posted. A second MPI will be requested before the next RTCI is generated. This MPI may or may not cause preemption.

The latency time for the first RTCI is:

$$T_A = T_{sa1} + T_{ca11} + T_{ca21}$$

Measurements of T_A during stand-by (by means of a prototype version of the software) delivered an average value of T_A of 200 usec, with small deviation.

According to this measurements, the time between the RTCI and the acknowledge to the hardware has also a fairly constant value of 22 usec, which is 6 usec greater than the basic cpu latency + the PIC expansion + the time required to save the global registers. This time is needed for program execution; no local registers frame needed to be flushed in order to allocate a frame for the interrupt service routine.

If we assume that the typical case cpu contribution to the latency time is equal to the average of best and worst cases, the average case when the scheduler is interrupted would be equal to 102 usec. This accounts for 51% of the latency time.

The latency time for synchronous context switchings is:

$$T_S = T_{SS1} + T_{CS1} + T_{CS2}$$

The value of Ts according to the measurements is in average 200 usec. Even though the deviation was small, the spreading of Ts felt within the [168 usec, 336 usec] range. The typical cpu contribution is 42.35 usec which accounts for 21% of the average latency time.

If at least one synchronous context switching occurs after one interrupt, the contribution of the processor to the latency time will be in average less than 36%.

These results show that the latency is mainly caused by the software administration. The cpu latency will become greater if for instance an interrupt becomes pending. Obviously, no precision actions should be undertaken by only software means (the precision of the copy registration of 0.1 mm allows a latency time of 476 microseconds maximum). In this cases hardware timers should be used.

Between two RTCIs there is a period of no activity. No tasks are scheduled. If we only take into consideration the periods of time where application software activity takes place, one fraction of this period will be consumed by the kernel and the intrinsic latency of the processor. This fraction expressed in percentages is what we call the overhead.

Measurements during stand-by indicate that the average task processing time is about 98 usec. This low value is due to the tasks that monitor inputs or set/clear outputs. Low processing times down to 48 usec were measured.

During stand-by the overhead is then 200/(200 +98) = 67%.

During run time, the average task processing time is about 123 usec (tasks of more than 500 usec or even more than 700 usec are sporadically scheduled). The latency time is also somewhat greater than in the stand-by situation: about 239 usec. The overhead remains 67% (this can be explained by the fact that the complexity of the tasks increases with the same factor as the latency time; furthermore, the number of synchronous context switchings is greater than the amount of asynchronous switchings, so that the effect of the interrupts is limited).

If we take into account that, if at least one synchronous context switching occurs after one interrupt, less than 36% of the latency time during stand-by is due to the cpu-architecture, the processor overhead will be less than 24% (processor activity that is not related to the application software; the result is more favourable for the run situation because of the increase of latency time due to the scheduler). This result shows that the RISC architecture of the 80960SA is not the biggest responsible for this high overhead.

Actually, the modularity extent of the application software, in combination with the characteristics of the kernel determine in a great extent the efficiency of the system. If the application software is optimized for a lower administration overhead, the cpu overhead will automatically decrease. This can be accomplished by designing the system with the fewest number of tasks possible [12] (each task will do as much work as possible before relinquishing the cpu).

If for instance an overhead of 50% could be attained, this would imply a processor overhead of about 18%. The running time of the applications tasks would be about 200 usec.

The effect of the context switching to the latency time may be reduced through the use of faster memory devices. If zero wait states could be attained (tacc < 62.5ns), the cpu latency time could be reduced with factor ranging between 2 to 3, but this will increase the cost of the system.

Note1: The measurements during running time do not cover all the different situations, only the start of a job with steady master speed was considered.

Note 2: Interrupts from other sources were not produced. As long as the (combined) repetition rate of the interrupts is lower than the real time clock frequency, the results that have been found here will be valid. The PIC interrupts will not introduce a high latency; about 40 usec will be produced by the CPU; the administration software will not be involved in the interrupt handling, so that no scheduler latency will be introduced. The service routines for these interrupts should be kept as simple as possible.

Note 3: The processor was running constantly in the interrupted state. And the use of the system stack was limited to those situations that the scheduler is interrupted. This causes less frame manipulations.

Note 4. Cumulative measurements show that the application software activity is concentrated within a period equal to 1/3 of the interval between two RTCIs, with sporadical schedulings during the rest of the time.

Note 5. Only the asynchronous latency time when the scheduler is interrupted and the synchrounous cases have been measured. The effect of preemption has not been explicitly measured, but the effect of the cpu will be more limited because the scheduler does not need to be reinitialized.

5.2 Non deterministic behaviour.

For architectures based on pipelines (like MIPS), there is an intrinsic non deterministic behaviour, because it is difficult to predict the state of the pipelines upon reception of an interrupt. The working of the caches introduces a time uncertainty. These aspects are accentuated in high performance applications, where the tolerances that are introduced are not acceptable.

The resolution of the actions that are controlled by means of software, is related to the highest reference frequency, in this case the frequency of the signal generated by the master belt: 300 Hz.

The scheduler has an action table that assigns priorities to the different tasks depending on the status of the time reference (real time clock counter), the position reference (the master pulse counter) and the sensor information. This assignation table may be tuned to the characteristics of the system.

The uncertainty of the processor (in microseconds) compared to the tolerances of the system (hundreds of microseconds), is not an issue for this application. More relevant is the problem of testing and debugging.

5.2.1 Debug facilities.

As performance-enhancement technics such as the caching of parallel operations and pipelined instruction execution are applied to RISC processors, debugging support becomes increasingly difficult [7].

The i960 architecture provides on chip tracing. With tracing the i960 can detect and trap on any combination of the following events:

- Instruction execution (single step).
- Execution of a taken branch instruction.
- Execution of a subroutine call (through a call instruction).
- Execution of a subroutine return instruction.
- Detection that the next instruction is a subroutine return.
- Execution of a call to a supervisor routine.
- Software break point (mark instruction).
- Hardware event.

Event-detection is done at the instruction decode stage of the i960 pipeline. The processor executes at full speed until the instruction decoder detects an event. When an event is detected, the processor transfers control through a vector to a debugging-support routine. When the i960 traps to the debugging routine, the stack provides the details of the trace event, including the type of debugging event that caused the trap and the address at which the event occurred. At the same time the pipeline is automatically cleared and the execution state preserved. Program execution can continue without impact by executing a return instruction.

Access to these events is controlled through the trace-control register. In the lower half of the register, one bit is provided for each type of event; the kernel may set the appropriate bit. The upper half reports the occurrence of events. The kernel's debugging routine can read this register to determine what events have occurred.

Breakpoint events allow breaking execution of any instruction, even those from the on chip instruction cache.

To monitor stack usage, a trap can be taken on any call or return operation. The handler logs the call depth and the stack position, which later can be used to recreate a dynamic map of stack utilization.

5.2.2 Amount of software code.

The main characteristic of a RISC processor is its reduced instruction set. If complex instructions need to be emulated through multiple simple instructions, the amount of code may increase considerably with respect to that of a conventional cisc processor. Measurements of the produced code have shown, that due to the simplicity of the data manipulation, there is no increase in the amount of code with respect to an equivalent software running on a 80188-platform.

6 Conclusions.

- Architecture development, a flexible solution.

A Flexible solution ready for production with three processors, that communicate with each other by means of a high speed communication medium, has been implemented by means of two boards:

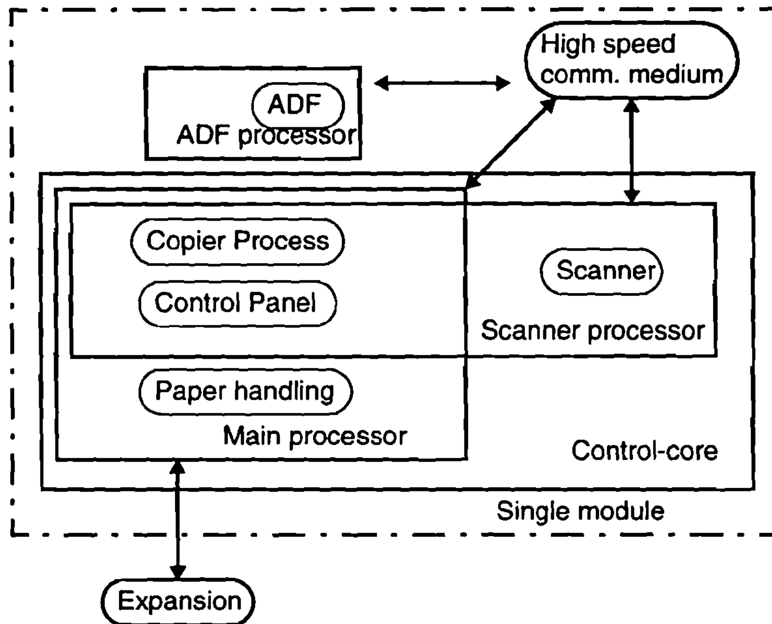


Figure 6.1: Flexible solution.

The Core Board.

It contains the main processor, the communication memory, the display interface and the peripherals of the processor. By separating this board from the i/o functions (including the ADF/Scanner processors) a possible migration to other main processor can take place. At the core board an i/o bus is implemented to access the i/o devices located on the i/o board as well as the extension connector which has, besides an 8-bits i/o-data bus, a couple of chip select lines, an interrupt line and the four least significant address lines. This way, devices of low complexity can be added. The amount of wait states for this eventual devices can be programmed in one GAL located at the core board. There is also a connector available for expansions that would require a large amount of address lines (memory components).

The I/O board.

This board contains the i/o devices. The Scanner and the ADF processors are tightly coupled to the main processor by means of the communication bus. One can decide whether to make use of EPROM's or to let the main processor down load the program code at dma basis through the communication bus.

This solution has so far proven to comply the requirements of the copier and the flexibility items in the design (expansion connectors) have been used to adapt some functions; one example: The control of one motor of the ADF function needed a 16 bits up/down counter as well as its related logic that generates the control signals

from the phase of the rotary encoder that is coupled to the motor. The temporary solution was a module with such a counter that has been installed at the expansion connector of the i/o board; the value of the counter was retrieved by the main processor and stored in the communication memory to be accessed by the ADF processor.

The flexibility of the design has made possible the use of some sections of the hardware by other projects. This has reduced their development time. However, the complexity of the solutions is high and the hardware is concentrated in one place. This results in a great deal of wiring.

- Architecture development, a modular solution.

For the most applications, based on this project, one solution has been found that compromises modularity with resource sharing. This is shown in the following figure:

.

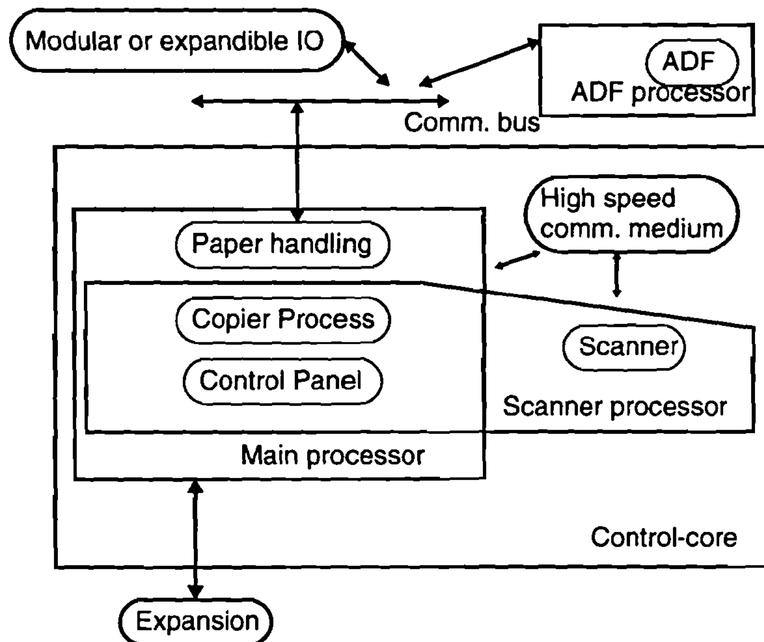


Figure 6.2: Modular solution.

A two processors-core functionality is in charge of the copier process, the control panel, the paper handling and the scanner functionality. This core is coupled to the ADF through a communication bus. Modular or remote io-functions can be coupled to implement the functionality of the different versions of the machine (to accommodate the different amount of paper reservoirs for instance).

This architecture has not been further investigated, but the expansion connectors of the implementation make possible the research of the real time implications of the communication bus.

- RISC combined with standard components.

Standard components have been used in order to keep low the price of the system. Some optimizations for the proper utilization of the RISC processor have been achieved:

1) The eprom interface makes use of an interleaved configuration that reduces the amount of wait states to an average of 1.25. A low eprom access time is important to maintain full the pipelines of the processor.

2) The communication memory arbiter grants the bus fairly in order to avoid stagnation of any of the processors.

However, the ram interface needs 2 wait states, this has repercussions on the context switching of the processor. The cpu overhead introduced by external accesses to ram can be reduced by a factor ranging between 2 to 3 if zero wait states are attained. This would imply access times shorter than 62.5 ns and the corresponding increase in price of the system.

- RISC and amount of software code.

The main characteristic of a RISC processor is its reduced instruction set. If complex instructions need to be emulated through multiple simple instructions, the amount of code may increase considerably with respect to that of a conventional cisc processor. Measurements of the produced code have shown, that due to the simplicity of the data manipulation, there is no increase in the amount of code with respect to an equivalent software running on a 80188-platform.

- RISC and Real Time.

One of the factors that make RISC fast is the large number of internal registers. They enable the RISC processor to perform many operations internally, hence reduce the access time to the external memory. In real time applications there are a lot of interactions with external agents. Furthermore in multitasking solutions, like this one, several tasks require processor services at the same time, for every task a stack is needed. The task state is stored in the stack when the task relinquishes the processor to another task (synchronous context switching) or when an interrupted is requested (asynchronous context switching).

The complexity of the tasks of the control of the copier is generally low, and therefore they do not require much processor power. The multiple switchings from one task to the other imply more overhead than the actual required application performance. The RISC processor architecture accounts for less than 24% overhead if at least one synchronous context switching occurs after one master pulse interrupt or one real time clock interrupt (the overhead is defined here as the percentage of time the processor performs activities that are not directly related to the application software, without taking into account the idle periods of the scheduler). The estimated total overhead of context switching is 67%.

The total (synchronous or asynchronous) latency time is in average 200 usec during stand-by and 239 usec during run time. This time may increase if an interrupt becomes pending. This latency time is acceptable.

The contribution of the RISC architecture to the latency time during stand-by is in average 51% for the asynchronous context switching and 21% for the synchronous context switching. During run time is the cpu contribution lower because of the increase of administration tasks of the scheduler.

The modularity extent of the application software, in combination with the characteristics of the kernel determine in a great extent the efficiency of the system. If the application software is optimized for lower administration overhead, the cpu overhead will automatically decrease. This can be accomplished by designing the system with

the fewest number of tasks possible (each task will do as much work as possible before relinquishing the cpu). If for instance an overhead of 50% could be attained, this would imply a processor overhead of about 18% in average. The average running time of the application tasks would be 200 usec.

There is no need at this moment to tune the software to the RISC-architecture. Measurements on the prototype software show that the application activity is concentrated during one third of the time between two real time clocks with sporadical schedulings during the other time periods.

- CPU architecture and future developments.

The 80960SA satisfies the requirements of this application. Even though the software is not tailored to the architecture of the CPU there is still processor power for eventual expansions of the system.

The cpu vendors have realized the possible drawbacks of the RISC architecture in real time applications. Papers [5] have been produced about techniques to improve the behaviour of the RISC processor (in silicon). This kind of architecture is gaining more and more terrain. It is recommended to spend more time to investigate the debug- and development-techniques involved.

Acknowledgments.

I would like to thank the Océ organization for the opportunity and support they gave me to realize this work, in particular Mr. Cees van Tilborg and Mr. Lou Dohmen for the time they spent to correct this report, as well as Mr. Erik Willemsen, Mr. John Arts, Mr. Hans Wierink and Mr. Chris Wieckardt for their advice during the preparation of the presentation.

I would like also to express my gratitude to Prof. Stevens for his valuable recommendations.

My special thanks goes to my family in the Netherlands for their patience during all these years of toil.

7 Literature Index.

1 Intel Corporation,
80960 Application Engineering.
i960 MICROPROCESSOR;
COMPETITIVE BENCHMARK REPORT.
July 29, 1993.

2 Intel.
i960SA/SB Microprocessor REFERENCE MANUAL.
Order Number: 270929-003, 1991.

3 Simpson, David.
REAL TIME RISCs.
System Integration, vol. 22, july 1989, p.34-38.

4 Intel, (Feldkrigen, West Germany).
ÜBER RISC HINAUS.
Elektronik Industrie, vol. 20 (1989), Iss. 10 p.23-24.

5 Elkateeb, Ali & Theo le-Ngoc.
A PRIORITY STRATEGY ON RISC FOR REAL TIME.
MULTITASKING SOFTWARE APPLICATIONS.
Computer Architecture news, vol. 17 (1989), p.62-68.

6 Goering, Richard.
IS RISC READY FOR REAL TIME APPLICATIONS?
High Performance Systems, vol. 11 (march 1990), p.22-24,28-29.

7 Baker, T.
MAXIMIZING HARDWARE/SOFTWARE TRADEOFFS IN
REAL TIME RISC-BASED SYSTEMS.
High Performance Systems, vol. 11 (1990), Iss. 3, p.49,52-54.

8 Willenz, Avigdor & Philip Baurekas.
CHOOSING THE RIGHT RISC ARCHITECTURE.
High Performace Systems, vol. 11 (1990), Iss. 3, p.42-43,46-48.

9 Bursky, David.
REGISTER WINDOWS SPEED REAL TIME CONTROL TASKS.
Electronic Design, vol. 38, Nov. 1990, p.57-58,60-61.

10 Wynia, Todd.
RISC AND CISC PROCESSORS TARGET EMBEDDED SYSTEMS.
Electronic Design, vol.39, 1991, Iss. 2, p.55-56,58, 62,67,69-70.

11 Ball, T.
REAL TIME OPERATING SYSTEM PERFORMANCE ISSUES & TECHNIQUES WITH RISC.
In: IEE colloquium on "RISC Architectures and Applications", London: IEE, 1991, IEE Colloquium,
Disert No. 163, p.7/1-7/8.

12 Burnel, Mitchel.
MAXIMIZING PERFORMANCE OF REAL TIME RISC APPLICATIONS.
Dr. Dobb's Journal, vol.19 (1994) Iss. 1, p.54,56,58,60,62,64,90,94,96.

13 Intel Corporation.
960SA EMBEDDED 32-BIT MICROPROCESSOR WITH 16-BIT BURST DATA BUS.
Data Sheets
Order Number: 272206-002, 1994.

14 Intel Corporation.
80C196KC
User's Guide.
Order Number: 270704-003, 1990.

Communication memory Interface

```
(Do not use for production).
MODULE arbiter;

FLAG '-r3';

TITLE 'communication ram arbiter '

"Device declaration

_arbiter                                device 'P22V10C';


"Pin declarations

"clock pin
        CLK             pin             2;


"input pins
        QREQ1n          pin             3;
        QREQ2n          pin             4;
        QREADY1n        pin             5;
        QREADY2n        pin             6;
        AH19            pin             7;
        AH20            pin             9;
        ALE960          pin             10;
        NRESETIO        pin             11;
        CRAMn           pin             12;
        OE              pin             13;
        DMA1ENn         pin             16;

"output pins
        GRANT1n         pin             27 ;
        GRANT2n         pin             26 ;
        DMARDREQn       pin             25 ;
        DMAWRREQn       pin             24 ;
        GRANTQ0         pin             23 istype 'reg,neg';
        GRANTQ2         pin             21 istype 'reg,neg';
        GRANTQ1         pin             20 istype 'reg,neg';
        GRANTQ3         pin             19 istype 'reg,neg';
```

```
            CYCLEEND        pin              18 ;

"i/o pins
            DMA2ENn         pin              17;


L,H         =       0,1;
Z,X,C       =       .Z.,.X.,.C.;

RDREQn      =       [DMARDREQn];
WRREQn      =       [DMAWRREQn];
ADDRESS     =       [CRAMn,AH20,AH19];
COMM        =       [L,L,L];
DMAREAD     =       [L,L,H];
DMAWRITE    =       [L,H,L];
notused     =       [L,H,H];
dontcare    =       [H,X,X];

" A dema request to the processors is done with a combination of the DMA
" READ or WRITE signals with AH18 which is also present on the IO-board
" AH18 DMARDREQn DMAWRREQn
"  0     0      1     dma read  access to coprocessor 1 memory
"  0     1      0     dma write access to coprocessor 1 memory
"  1     0      1     dma read  access to coprocessor 2 memory
"  1     1      0     dma write access to coprocessor 2 memory
" The dma interface waits until a confirmation of the hardware returns
" that a dma can take place, by means of the DMAENABLE signals.
" The choice to use two different addresses for dma-read and dma-write
" has been done in order to limit the overhead to place the transceivers
" in the correct mode, notice that the DMA accesses are not different from
" the accesses to the communication memory (the amount of wait states is
" the same).

GRANT = [GRANTQ3,GRANTQ2,GRANTQ1,GRANTQ0];

P1_GRANTED2     = ^b0000; " 0
P1_IDLE2        = ^b1001; " 9
P1_GRANTED1     = ^b0001; " 1
P1_IDLE1        = ^b1100; " c
P2_GRANTED2     = ^b0011; " 3
P2_IDLE2        = ^b1010; " a
P2_GRANTED1     = ^b0111; " 7
```

```
P2_IDLE1          = ^b1110; " e
MAIN_IDLE         = ^b1111; " f
MAIN_GRANTED      = ^b0110; " 6
DMA_READ          = ^b0100; " 4
DMA_WRITE         = ^b0010; " 2


BEGINMAINACCESS = ^b0101; " 5

reserve2          = ^b1000; " 8
reserve3          = ^b1101; " d
reserve4          = ^b1011; " b
random            = [X,X,X,X];


" main_idle -> main granted -> p2_idle1
"           dmaread -> main_idle
"           dmawrite -> main_idle
"           p2_idle1
" p2_idle1  -> p2_granted1 -> p2_idle2
"           P1_idle1
" p2_idle2  -> p2_granted2 -> p1_idle1
"           p1_idle1
" p1_idle1  -> p1_granted1 -> p1_idle2
"           main_idle
" p1_idle2  -> p2_granted2 -> main_idle
"           main_idle


STATE_DIAGRAM GRANT

state reserve2: goto MAIN_IDLE;
state reserve3: goto MAIN_IDLE;
state reserve4: goto MAIN_IDLE;

state MAIN_IDLE:
            if !CYCLEEND & !CRAMn  & NRESETIO then BEGINMAINACCESS
            else P2_IDLE1;
state MAIN_GRANTED:
            if CRAMn then P2_IDLE1
            else MAIN_GRANTED;
state P2_IDLE1:
```

```
                if !CYCLEEND & !QREQ2n  then P2_GRANTED1
                else P1_IDLE1;
state P2_IDLE2:
                if  CYCLEEND then P2_IDLE2 else
                        if !CYCLEEND & !QREQ2n  then P2_GRANTED2
                        else P1_IDLE1;
state P1_IDLE1:
                        if !CYCLEEND & !QREQ1n  then P1_GRANTED1
                        else MAIN_IDLE;
state P1_IDLE2:
                if  CYCLEEND then P1_IDLE2 else
                        if !CYCLEEND & !QREQ1n  then P1_GRANTED2
                        else MAIN_IDLE;

state P2_GRANTED1:
                if (CYCLEEND) then P2_IDLE2
                ELSE P2_GRANTED1;
state P2_GRANTED2:
                if (CYCLEEND) then P1_IDLE1
                ELSE P2_GRANTED2;
state P1_GRANTED1:
                if (CYCLEEND) then P1_IDLE2
                ELSE P1_GRANTED1;
state P1_GRANTED2:
                if (CYCLEEND) then MAIN_IDLE
                ELSE P1_GRANTED2;
state DMA_READ:
                if (CRAMn) then MAIN_IDLE
                else DMA_READ;
state DMA_WRITE:
                if (CRAMn) then MAIN_IDLE
                else DMA_WRITE;
state BEGINMAINACCESS:
                if (ADDRESS == COMM)  then MAIN_GRANTED
                else if (ADDRESS == DMAREAD) & (!DMA1ENn # !DMA2ENn)
                then DMA_READ
                        else if (ADDRESS == DMAWRITE) & (!DMA1ENn # !DMA2ENn)
                                then DMA_WRITE;
                                else BEGINMAINACCESS ;

EQUATIONS
```

```
!GRANT1n = ((GRANT.FB == P1_GRANTED1) # (GRANT.FB == P1_GRANTED2))
            & !CLK & !CYCLEEND & NRESETIO
            # !GRANT1n & !GRANTQ3.FB & NRESETIO
            & !CYCLEEND & QREADY1n;
!GRANT2n = ((GRANT.FB == P2_GRANTED1) # (GRANT.FB == P2_GRANTED2))
            & !CLK & !CYCLEEND & NRESETIO
            # !GRANT2n & !GRANTQ3.FB & NRESETIO & !CYCLEEND & QREADY2n;
!DMARDREQn =
            (ADDRESS == DMAREAD) & DMA1ENn & DMA2ENn
            # !DMARDREQn & !CYCLEEND & NRESETIO;
!DMAWRREQn =
            (ADDRESS == DMAWRITE) & DMA1ENn & DMA2ENn
            # !DMAWRREQn & !CYCLEEND & NRESETIO;

"A dma access is started every time ADDRESS == DMAREAD of DMAWRITE.
" The 80196's can be reset under 80960 control.
" The main processor then accesses the memory.
"

CYCLEEND =
            CRAMn & (GRANT.FB == MAIN_GRANTED)
            # CRAMn & (GRANT.FB == DMA_READ)
            # CRAMn & (GRANT.FB == DMA_WRITE)
            # !QREADY1n & (GRANT.FB == P1_GRANTED1)
            # !QREADY1n & (GRANT.FB == P1_GRANTED2)
            # !QREADY2n & (GRANT.FB == P2_GRANTED1)
            # !QREADY2n & (GRANT.FB == P2_GRANTED2)
            # CYCLEEND & NRESETIO & !GRANTQ3.FB
            # CYCLEEND & !QREADY1n & NRESETIO
            # CYCLEEND & !QREADY2n & NRESETIO
            # CYCLEEND & !DMA1ENn  & NRESETIO
            # CYCLEEND & !DMA2ENn  & NRESETIO
            ;

            GRANTQ0.C =    CLK;
            GRANTQ1.C =    CLK;
            GRANTQ2.C =    CLK;
            GRANTQ3.C =    CLK;
            GRANTQ3.AR =   !NRESETIO;

END arbiter;
```

```
MODULE crcntr;

FLAG '-r3';

TITLE 'communication ram controller'

"Device declaration

_crcntr                                 device 'P22V10C';


"Pin declarations

"clock pin
            CLK             pin         2;


"input pins
            NRESET          pin         3;
            CEND            pin         4;
            GRANTQ0         pin         5;
            GRANTQ2         pin         6;
            GRANTQ1         pin         7;
            GRANTQ3         pin         9;
            WE0n            pin         10;
            WE1n            pin         11;
            DEVOEn          pin         12;
            ENDMAn          pin         13;
            OE              pin         16;

"output pins
            CRAMCSn         pin         27;
            CRAMRDn         pin         26;
            CRAMW1n         pin         25;
            CRAMW2n         pin         24;
            POSTPONEn       pin         23;
            DIRBUFEXT       pin         21;
            WAIT            pin         20 istype 'reg';
            M2ERDn          pin         19;
            M2EWRLn         pin         18;
            M2EWRHn         pin         17;
```

```
L,H              =      0,1;
Z,X,C            =      .Z.,.X.,.C.;


GRANT                  =     [GRANTQ3,GRANTQ2,GRANTQ1,GRANTQ0];

dontcare               =     [X,X,X,X];
P1_GRANTED2            =     ^b0000; " 0
P1_IDLE2               =     ^b1001; " 9
P1_GRANTED1            =     ^b0001; " 1
P1_IDLE1               =     ^b1100; " c
P2_GRANTED2            =     ^b0011; " 3
P2_IDLE2               =     ^b1010; " a
P2_GRANTED1            =     ^b0111; " 7
P2_IDLE1               =     ^b1110; " e
MAIN_IDLE              =     ^b1111; " f
MAIN_GRANTED           =     ^b0110; " 6
DMA_READ               =     ^b0100; " 4
DMA_WRITE              =     ^b0010; " 2
BEGINMAINACCESS =            ^b0101; " 5

reserve2               =     ^b1000; " 8
reserve3               =     ^b1101; " d
reserve4               =     ^b1011; " b

CSn = [CRAMCSn];
RDn = [CRAMRDn];
W1n = [CRAMW1n];
W2n = [CRAMW2n];
DIR  = [DIRBUFEXT];
WAITn = [POSTPONEn];
EXRDn = [M2ERDn];
EXW1n = [M2EWRLn];
EXW2n = [M2EWRHn];


" main_idle -> main granted -> p2_idle1
"          dmaread -> main_idle
"          dmawrite -> main_idle
```

```
"           p2_idle1
" p2_idle1  -> p2_granted1 -> p2_idle2
"           P1_idle1
" p2_idle2  -> p2_granted2 -> p1_idle1
"           p1_idle1
" p1_idle1  -> p1_granted1 -> p1_idle2
"           main_idle
" p1_idle2  -> p2_granted2 -> main_idle
"           main_idle
```

EQUATIONS

```
!CRAMCSn = ((GRANT == MAIN_GRANTED) & !CEND
            # (GRANT == P1_GRANTED1)  & !CEND
            # (GRANT == P1_GRANTED2)  & !CEND
            # (GRANT == P2_GRANTED1)  & !CEND
            # (GRANT == P2_GRANTED2)  & !CEND) & !CLK
            # !CRAMCSn & !GRANTQ3   & !CEND;


!CRAMRDn =  (GRANT == MAIN_GRANTED) & !DEVOEn & !CEND
           # (GRANT == P1_GRANTED1) & !M2ERDn & !CEND
           & !ENDMAn & !DIRBUFEXT
           # (GRANT == P1_GRANTED2) & !M2ERDn & !CEND & !ENDMAn
           & !DIRBUFEXT
           # (GRANT == P2_GRANTED1) & !M2ERDn & !CEND
           & !ENDMAn & !DIRBUFEXT
           # (GRANT == P2_GRANTED2) & !M2ERDn & !CEND
           & !ENDMAn & !DIRBUFEXT;

!CRAMW1n =  (GRANT == MAIN_GRANTED) & !WE0n & !CEND
           # (GRANT == P1_GRANTED1) & !M2EWRLn & !CEND
           # (GRANT == P1_GRANTED2) & !M2EWRLn & !CEND
           # (GRANT == P2_GRANTED1) & !M2EWRLn & !CEND
           # (GRANT == P2_GRANTED2) & !M2EWRLn & !CEND ;

!CRAMW2n =  (GRANT == MAIN_GRANTED) & !WE1n & !CEND
           # (GRANT == P1_GRANTED1) & !M2EWRHn & !CEND
           # (GRANT == P1_GRANTED2) & !M2EWRHn & !CEND
           # (GRANT == P2_GRANTED1) & !M2EWRHn & !CEND
           # (GRANT == P2_GRANTED2) & !M2EWRHn & !CEND ;
```

```
WAIT :=
        (GRANT == BEGINMAINACCESS)
        # (!ENDMAn) & ((GRANT == P1_GRANTED1)
        # (GRANT == P1_GRANTED2)
        # (GRANT == P2_GRANTED1)
        # (GRANT == P2_GRANTED2))
        & !GRANTQ3 & (!M2ERDn)
        # WAIT.FB & !GRANTQ3;

POSTPONEn = WAIT.FB & (
        (GRANT == MAIN_GRANTED) #
        (GRANT == DMA_READ   ) #
        (GRANT == DMA_WRITE  ) ) #
        POSTPONEn & !CEND;

!M2ERDn = !DEVOEn;

!M2EWRLn = !WE0n;

!M2EWRHn = !WE1n;

DIRBUFEXT = ((GRANT == P1_GRANTED1) & (M2ERDn ) & !WAIT.FB
        # (GRANT == P1_GRANTED2) & (M2ERDn ) & !WAIT.FB
        # (GRANT == P2_GRANTED1) & (M2ERDn ) & !WAIT.FB
        # (GRANT == P2_GRANTED2) & (M2ERDn ) & !WAIT.FB
        # (GRANT == DMA_READ   ))
         ;

M2ERDn.OE       = (GRANT == DMA_READ);
M2EWRLn.OE      = (GRANT == DMA_WRITE) ;
M2EWRHn.OE      = (GRANT == DMA_WRITE);
CRAMCSn.OE      = OE;
CRAMRDn.OE      = OE;
CRAMW1n.OE      = OE;
CRAMW2n.OE      = OE;
POSTPONEn.OE    = OE;
WAIT.OE = OE;
DIRBUFEXT.OE    = OE;

END crcntr;
```

```
MODULE bufcnt;

TITLE 'communcation ram, buffers controller'

"Device declaration
"******************

_bufcnt             device 'P16V8R';          "GAL16V8B-7LJ

"Pin declarations
"****************

"Input/clock pin
"---------------

        SYSCLK2         pin             1;
        ENGALn          pin             11;

"Input pins
"----------

        CYCLEEND        pin             2;
        GRANTQ0         pin             3;
        GRANTQ2         pin             4;
        GRANTQ1         pin             5;
        GRANTQ3         pin             6;
        DEVOEn          pin             7;
        WE0n            pin             8;
        WE1n            pin              9;

"output pins
"-----------------

        ADRDIREXT       pin             19 istype 'invert';
        OEBUFEXTn       pin             17 istype 'invert';
        ENABLEDMAn      pin             16 istype 'invert';
        NOECOMn         pin             15  istype 'invert';
        ACCESSISREADn pin               14 istype 'reg,neg';
        _1STCYCLEn      pin             13 istype 'reg,neg';
        _2NDCYCLEn      pin             18 istype 'reg,neg';

"Input/output pins
```

```
                NRESETIO        pin                     12;

        "Constant declarations
        "**********************



        L,H        =  0,1;
        Z,X,C,P    =  .Z.,.X.,.C.,.P.;



        GRANT = [GRANTQ3,GRANTQ2,GRANTQ1,GRANTQ0];

        P1_GRANTED2        = ^b0000; " 0
        P1_IDLE2           = ^b1001; " 9
        P1_GRANTED1        = ^b0001; " 1
        P1_IDLE1           = ^b1100; " c
        P2_GRANTED2        = ^b0011; " 3
        P2_IDLE2           = ^b1010; " a
        P2_GRANTED1        = ^b0111; " 7
        P2_IDLE1           = ^b1110; " e
        MAIN_IDLE          = ^b1111; " f
        MAIN_GRANTED       = ^b0110; " 6
        DMA_READ           = ^b0100; " 4
        DMA_WRITE          = ^b0010; " 2

        reserve2           = ^b1000; " 8
        reserve3           = ^b1101; " d
        reserve4           = ^b1011; " b


        dontcare           = [X,X,X,X];

        _1STn = [_1STCYCLEn];
        _2NDn = [_2NDCYCLEn];

        BEGINMAINACCESS       = ^b0101; " 5

        " main_idle -> main granted -> p2_idle1
        "          dmaread -> main_idle
        "          dmawrite -> main_idle
```

```
"          p2_idle1
" p2_idle1  -> p2_granted1 -> p2_idle2
"          P1_idle1
" p2_idle2  -> p2_granted2 -> p1_idle1
"          p1_idle1
" p1_idle1  -> p1_granted1 -> p1_idle2
"          main_idle
" p1_idle2  -> p2_granted2 -> main_idle
"          main_idle
```

EQUATIONS

```
!ACCESSISREADn := (GRANT == MAIN_GRANTED) & !DEVOEn #
                  (GRANT == DMA_READ)     & !DEVOEn ;


" direction signals for coprocessors address transceivers.
" dma is default
ADRDIREXT = !_2NDCYCLEn.FB & ((GRANT == P1_GRANTED1)
          # (GRANT == P1_GRANTED2)
          # (GRANT == P2_GRANTED1)
          # (GRANT == P2_GRANTED2)) & NRESETIO & !CYCLEEND
          # ADRDIREXT & !OEBUFEXTn;

" coprocessor's data/address transceivers
!OEBUFEXTn =
          ((( GRANT == P1_GRANTED1)
          # (GRANT == P1_GRANTED2)
          # (GRANT == P2_GRANTED1)
          # (GRANT == P2_GRANTED2)) & ADRDIREXT & !CYCLEEND
          # (GRANT == DMA_READ) & !_2NDCYCLEn.FB
          # (GRANT == DMA_WRITE)& !_2NDCYCLEn.FB) & NRESETIO;

" transceiver for:
" 1. control signals from coprocessors
" 2. dma control signals
!ENABLEDMAn =
```

```
            ((( GRANT == P1_GRANTED1)
            # (GRANT == P1_GRANTED2)
            # (GRANT == P2_GRANTED1)
            # (GRANT == P2_GRANTED2)) & ADRDIREXT & !CYCLEEND
            # (GRANT == DMA_READ) & !_2NDCYCLEn.FB
            # (GRANT == DMA_WRITE)& !_2NDCYCLEn.FB) & NRESETIO;

" 80960  data/address transceivers (direction = DRn)
!_1STCYCLEn := (GRANT == BEGINMAINACCESS) & !GRANTQ3 ;
!_2NDCYCLEn := !_1STCYCLEn.FB  & !GRANTQ3
            # (GRANT == P1_GRANTED1) & !GRANTQ3
            # (GRANT == P1_GRANTED2) & !GRANTQ3
            # (GRANT == P2_GRANTED1) & !GRANTQ3
            # (GRANT == P2_GRANTED2) & !GRANTQ3
            # (GRANT == DMA_READ) & !GRANTQ3
            # (GRANT == DMA_WRITE) & !GRANTQ3;
" BEGINCYCLEn prevents spikes on NOECOMn due to transitions between states.

!NOECOMn =  !_1STCYCLEn.FB & !CYCLEEND
            # !NOECOMn & !GRANTQ3 & ACCESSISREADn.FB
            & NRESETIO & !CYCLEEND
            # !NOECOMn & !GRANTQ3 & !DEVOEn &
            NRESETIO & !CYCLEEND;

ADRDIREXT.OE = !ENGALn & H ;
OEBUFEXTn.OE = !ENGALn  & H;
ENABLEDMAn.OE = !ENGALn  & H;
NOECOMn.OE = !ENGALn  & H;
ACCESSISREADn.OE = !ENGALn;
_1STCYCLEn.OE = !ENGALn;
_2NDCYCLEn.OE = !ENGALn;



END bufcnt
```

Eprom interface.

```
(Do not use for production).
MODULE memcnt;

FLAG '-r3';

TITLE 'memory controller gal '

"Device declaration

_memcnt          device                'P22V10C';

" The state machines contained in this gal will be also used for flash eproms,
" The read signal for the RAM's and the related RDY line are also generated
" here.
" The EPROM is interleaved, the amount of wait states is as follows:
" 2-1-1-1.
" For the RAM, there are 2 wait states introduced.

"Pin declarations

"clock pin
        CLK3            pin             2;


"input pins
        !EPROMCS        pin             3;
        !PCYCLE         pin             4;
        DISNRDY         pin             5;
        !RD             pin             6;
        !BE0            pin             7;
        !BE1            pin             9;
        A1PROL          pin             10;
        A2PROL          pin             11;
        !POSTPONE       pin             12;
        !DISRAMCS       pin             13;
        !COMRAMCS       pin             16;

"output pins
        !ENABLERDY      pin             27 istype 'neg';
        !SYSRAM         pin             26;
```

```
            !IORDY        pin         25;
            !RDY          pin          24 ;
            WE0           pin         23 istype 'reg,neg';
            WE1           pin         21 istype 'reg,neg';
            !MEMRDY       pin         20 istype 'reg,neg';
            S1            pin         18 istype 'reg,neg';
            S0            pin         19 istype 'reg,neg';
            DEVOE         pin         17 ;


L,H    =    0,1;
Z,X,C  =    .Z.,.X.,.C.;

ADDR = [A2PROL,A1PROL];

MEM_CNT = [S1,S0,!MEMRDY];
IDLE        = ^b111;
W2          = ^b101;
W1          = ^b001;
D1          = ^b100;
D2          = ^b110;
SP1         = ^b010;
SP2         = ^b011;
SP3         = ^b000;
dontcare    = [X,X,X];


ENRDY = [ENABLERDY];

STATE_DIAGRAM  MEM_CNT

state IDLE:
            WE0  := 0;
            WE1  := 0;
            if !PCYCLE then IDLE
            else     if (EPROMCS & !SYSRAM & !DISRAMCS
            #  SYSRAM & (!EPROMCS & !DISRAMCS & !COMRAMCS ))
                          then W1
                          else if
                                   !SYSRAM & (DISRAMCS &
                                   !EPROMCS # COMRAMCS) then SP2;
else IDLE;
```

```
" If the communication ram or the display ram are selected, the state
" machine goes to SP2 to wait until the bus is granted (POSTPONE or DISplay
" NotReaDY active)


state W1:
            if
            (DISRAMCS # COMRAMCS) & (WE0.FB # WE1.FB ) then D1
                  with
                  WE0 := 0;
                  WE1 := 0;
                  endwith;
                  else W2 with
                        WE0 :=  !RD & BE0;
                        WE1 :=  !RD & BE1;
                  endwith;

state W2:
            if (DISRAMCS # COMRAMCS) & !RD
            then W1 with
            WE0 := !RD & BE0;
            WE1 := !RD & BE1;
            endwith;
            else D1 with
                  WE0 := 0;
                  WE1 := 0;
                  endwith;

state D1:
            WE0  := 0;
            WE1  := 0;
            if !PCYCLE then IDLE
            else if DISRAMCS then SP2
                  else if (EPROMCS & A1PROL &
                        A2PROL # !EPROMCS) then W1
                        else W2;


state D2:
            goto IDLE;
```

```
state SP2:
        if (COMRAMCS & POSTPONE  #  DISRAMCS & DISNRDY) then SP2
        with
                WE0  := 0;
                WE1  := 0;
        endwith;
        else W1 with
                WE0 :=  0;
                WE1 :=  0;
        endwith;
state SP1: goto IDLE;
state SP3: goto IDLE;

EQUATIONS

RDY = ENABLERDY ;

ENABLERDY = MEMRDY.FB # IORDY  ;
" the ready signal is enabled from the first wait state until the recovery
" cycle, inclusive this last one. Another device may enable this line
" during an overlaping period, from the first wait- or data state.
" SYSRAM is activated from the beginning of the cycle in case of RAM accesses,
" for accesses to the other local devices, SYSRAM is activated after
" the address cycle. In both situations, this signal will stay active
" during the whole cycle until BLASTn is deactivated in the recovery cycle.

RDY.OE = SYSRAM;

DEVOE =
        RD
        & ((MEM_CNT.FB == W1))  #
          DEVOE & !S0.FB;

" The outputs are enabled after the state machine starts the access
" until the recovery time. This signal is not clocked in order to compensate
" the clocks skew.
END memcnt
```

```
MODULE romcnt;

TITLE 'rom controller gal'

"Device declaration
"******************

_romcnt             device              'P22V10C';

"Pin declarations
"****************

"Input/clock pin
"---------------

        SYSCLK3     pin         2;

"Input pins
"----------

        !PCYCLE     pin         3;
        !RD         pin         4;
        A1PROL      pin         5;
        A2PROL      pin         6;
        A3          pin         7;
        !BE1        pin         9;
        !BE0        pin         10;
        !ROMRDY     pin         11;
        !EPROMCS    pin         12;
        !ASPROL     pin         13;
        AH21        pin         16;

"output pins
"-----------------

        !MA2EV      pin         27 istype 'reg';
        !MA2ODD     pin         26 istype 'reg';
        !MA3        pin         25 istype 'reg';
        !ROMEVOE    pin         24;
        !ROMODDOE   pin         23;
        !EPROM1CS   pin         21;
        !EPROM2CS   pin         20;
```

```
                PROCYCLEQn      pin             19 ;

" i/o pins

                nc              pin             18;
                AH22            pin             17;

"Constant declarations
"*********************

ADD_EV_STATE        = [!MA2EV];
ADD_OD_STATE        = [!MA2ODD];
MA2PROL_0           = ^b1;
MA2PROL_1           = ^b0;

MA3_STATE           = [!MA3];
MA3_0               = ^b1;
MA3_1               = ^b0;

ACTIVE              = ^b0;
INACTIVE            = ^b1;
ADDR                = [A3,A2PROL,A1PROL];
ADD                 = [AH22,AH21];
EPROM1              = ^b00;
EPROM2              = ^b01;
EPROM3              = ^b10;
" the eproms are placed contiguously, the size of the banks is 2 MBytes.
" In case components of higher density are used modify the address combinations

L,H      = 0,1;
Z,X,C,P  = .Z.,.X.,.C.,.P.;

STATE_DIAGRAM ADD_EV_STATE

state MA2PROL_0:
                if (ASPROL & A2PROL & !A1PROL
                # ASPROL & !A2PROL & A1PROL
                # !ASPROL & !ROMRDY & !A2PROL & A1PROL)
                then MA2PROL_1
                else MA2PROL_0;

state MA2PROL_1:
```

```
        if (ASPROL & !A2PROL  & !A1PROL
                # ASPROL &  A2PROL  &  A1PROL
                # !ASPROL & !ROMRDY & A2PROL & A1PROL)
        then MA2PROL_0
        else MA2PROL_1;

STATE_DIAGRAM ADD_OD_STATE

state MA2PROL_0:
        if (ASPROL & A2PROL #
                !ASPROL & !ROMRDY & A2PROL & !A1PROL) then MA2PROL_1
        else MA2PROL_0;
state MA2PROL_1:
        if (ASPROL & !A2PROL #
                !ASPROL & !ROMRDY & !A2PROL & !A1PROL) then MA2PROL_0
        else MA2PROL_1;

STATE_DIAGRAM MA3_STATE

state MA3_0:
        if (ASPROL & A3 # !ASPROL & !ROMRDY & A3) then MA3_1
        else MA3_0;
state MA3_1:
        if (ASPROL & !A3 # !ASPROL & !ROMRDY & !A3) then MA3_0
        else MA3_1;

EQUATIONS
!PROCYCLEQn = PCYCLE
        # (!PROCYCLEQn) & (BE0 # BE1);

ROMEVOE  =  EPROMCS & !A1PROL & RD & ROMRDY & !PROCYCLEQn
    # ROMEVOE & !A1PROL & EPROMCS  & !PROCYCLEQn ;
ROMODDOE =  EPROMCS &  A1PROL & RD & ROMRDY & !PROCYCLEQn
    # ROMODDOE & A1PROL & EPROMCS  & !PROCYCLEQn;

EPROM1CS = EPROMCS & (ADD == EPROM1);
EPROM2CS = EPROMCS & (ADD == EPROM2);

END romcnt
```