Realisation of the Area Segmentation Processor for ESPRIT project #2017 : TRIOS

de Winne, P.T.M.

Award date:
1993

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
Measurement and Control Section

REALISATION OF THE

AREA SEGMENTATION PROCESSOR

FOR ESPRIT PROJECT #2017: TRIOS

by P.T.M. de Winne

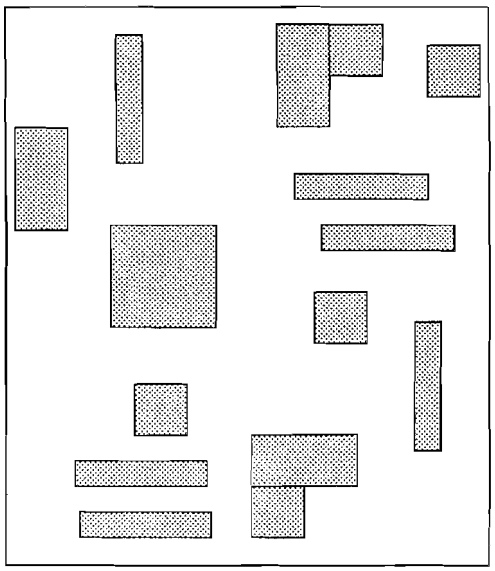
M.Sc. Thesis/Report on Practical training period
carried out from september 1992 to june 1993
commissioned by prof.dr.ir. A.C.P.M. Backx
under supervision of ir. R. van Vliet of the TUE
and ing. A.J.M. van Lier of Philips CFT
date: 7 june 1993

Realisation of the

Area Segmentation Processor

for

Esprit Project #2017: TRIOS



author :	P.T.M. de Winne
id. nr. :	318778
institute :	University of Technology Eindhoven
department :	Information Engineering
professor :	prof.dr.ir A.C.P.M. Backx
coach :	ir. R. van Vliet
company :	Nederlandse Philips Bedrijven B.V.
department :	CFT-ISP-IMI
coach :	ing. A.J.M. van Lier

period : september 1992 - june 1993

Summary

Philips CFT (Centre For manufacturing Technology) is partner in the ESPRIT project #2017: TRIOS, for the development of an in-line inspection system. The inspection system checks PCBs, which are suitable for surface mounted devices, after solder paste has been laid on the PCB. The purpose of the system is to decide wether or not the solder paste is on the right position and has the right volume.

A laser scanner extracts height and intensity information from a PCB by means of a laser beam. This information is sent to a data processing unit. The data processing unit calculates the position and the height of the solder paste and checks that those values are between certain bounds. The results are given to the host which controls the whole system.

The Area Segmentation Processor (ASP) segments the image information. This to distribute the data over the calculation processors and to reduce the data for the calculation processors. The areas of interest (AOIs), which have to be segmented, are specified in the CAD-data. The ASP composes the stored AOIs into images (called screens), which have a suitable format for the calculation processors and sends these screens to the calculation processors.

The controller of the ASP converts the specification of the areas of interest into addresses on which the data is stored in the circular buffer. Information within an AOI is placed on a unique address in the circular buffer. Information outside an AOI is placed on address zero. After the AOIs are stored, the AOIs are composed to one or more screens by the screen handler. The screens are sent to the calculation processors. The composition of the AOIs is calculated by software.

The hardware of the Area Segmentation Processor is improved and implemented in the system. Software is written and tested so that 512^2 images are received and sent. The segmentation software is partly written. The processor, a RISC processor, can be programmed in assembly or in C. Using local variables instead of global variables will save processor time.

Preface

This report is written by Peter de Winne, student information engineering (ITE) at the University of Technology Eindhoven (TUE), as graduation report. The graduation was carried out at Philips Centre For manufacturing Technology (CFT), section Industrial Signal Processing (ISP), group Industrial Measurement and Inspection (IMI).

The section Industrial Signal Processing provides support for the application of electronic technologies in production processes and products. The main activity areas are Industrial Measurement and Inspection with the emphasis on Machine Vision and Motion Control Technology. The object of the group Industrial Measurement and Inspection is to provide in solving automatic optical inspection problems in production and develop methods, tools and modules for applications in this field.

My graduation assignment was to realize a processor board (the area segmentation processor) that segments areas of an image and places these areas into screens. Therefor the schemes of the hardware are checked, the hardware is build, tested and improved where necessary and the software is written.

I want to thank ir. N.G.M. Kouwenberg who gave me the opportunity to fulfil my graduation assignment at Philips CFT. I want to say special thanks to ing. A.J.M. van Lier who gave me the graduation assignment and did the coaching of my work at Philips CFT and ir. R. van Vliet who did the coaching at the TUE. Further I want to thank ir. F.G.M. Smeets, ing. J.F.J. Hendriks, ir. L.H.D. Geraats, ing. M.F.W. Pechler and H.M. van Meurs for their support during my graduation period.

Content

List of figures

List of tables

1 Introduction

In the production of PCBs much has changed during the last years. The sizes of resistors, capacitors, integrated circuits etc. are getting smaller and smaller. They are placed on the print by machines and nowadays the production of a PCB is checked after every process phase to maintain quality. For this last step Philips CFT (Centre For manufacturing Technology) is partner in the ESPRIT project #2017: TRIOS. Within this project Philips CFT is developing an in-line PCB inspection system. The PCBs are suitable for surface mounted devices (SMD) technology. The inspection system will check the PCBs after the solder paste (= a composure of solder tin and solder flux) has been laid on the solder pads. The purpose of the system is to decide wether or not the solder paste on the PCB is on the right position and has the right volume. When for instance the solder paste isn't at the right position it can cause short circuits or when the volume is too less the SMD will not be connected properly.

As a part of the inspection system the Area Segmentation Processor (ASP) segments the data information from a PCB. This segmentation is necessary for the distribution of the data over the calculation processors and to reduce the data of a PCB so that only interested areas are checked.

The hardware of the Area Segmentation Processor is designed by F.G.M. Smeets who graduated in august 1992 on this design ([SME][1]). I continued the work with the Area Segmentation Processor. My graduation assignment was:

- build and test the hardware of the ASP,
- develop the software and
- integrate the ASP in the inspection system.

The assignment was carried out by:

- studying the inspection system,
- studying the functions and the schemes of the ASP,
- studying the processor of the ASP,
- making worst case calculations of the timing of the different components,
- connecting and testing the ASP part by part,
- integrate the ASP in the system and writing software for receiving small images so that no segmentation has to be done and
- writing the segmentation software after studying the system development schemes.

[1] Reference to literature list at page 42.

The results at the end of the graduation period are that the Area Segmentation Processor has been built and tested. It has been implemented in the system and for small images where no segmentation is necessary the software is written and tested. The segmentation software is written but not tested.

This report describes the realisation of the Area Segmentation Processor. Chapter 2 gives the different functions of the inspection system. In chapter 3 the hardware of the ASP is split into functional blocks. An outline of the software is given in chapter 4. Changes in the hardware are given in chapter 5 and the implementation of the software is given in chapter 6.

2 In-line PCB inspection system

The PCB inspection system must be suitable to inspect PCBs after the solder paste has been laid on the PCB. The volume and position of the solder paste must be between certain bounds. PCB sizes are up to 50x30 cm and the inspection time for the largest PCB is maximal 50 seconds. The inspection has to be done with a resolution of 25 µm. With these demands the system is designed. In the first paragraph the design of the inspection system is split into blocks. The function of each block is explained there. In the second paragraph the data processing system, which is a block of the inspection system, is further split into functional blocks and these blocks are explained.

2.1 Global explanation of the system

A global diagram of the in-line inspection system is given in figure 2.1. The lines represent data flows and the dotted lines represent control flows.

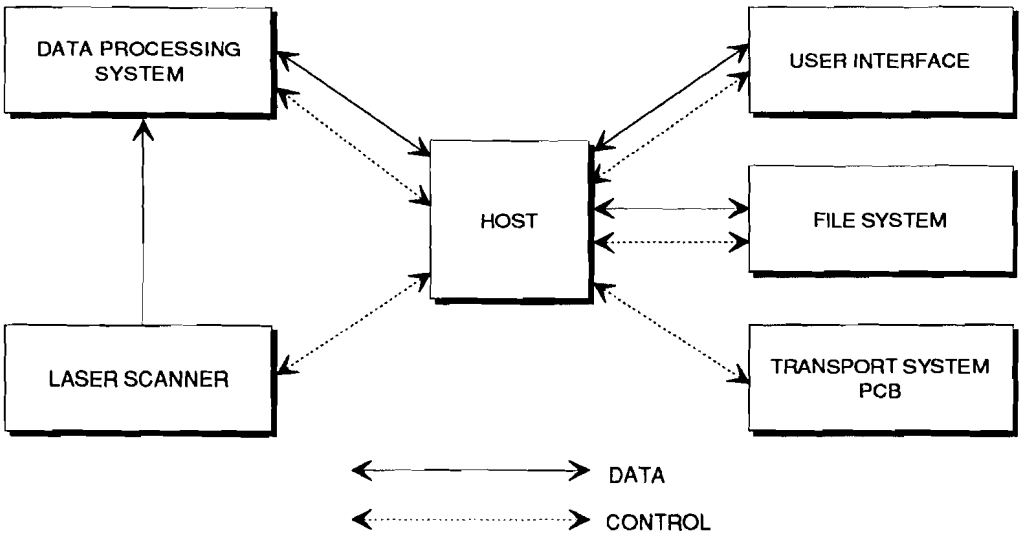


Figure 2.1 *Diagram in-line inspection system*

The function of each block is:

- host. The host controls the entire system.
- laser scanner. The laser scanner extracts image information from the PCB by means of a laser beam. The reflected light is translated into two data streams: height and intensity data. The height data is used for the volume measurement and the intensity data for the position measurement. These two streams are sent to the data processing system.

- data processing system. The data processing system selects the solder pad information from the data streams and calculates the position and volume of the solder paste. The solder pads are found using data from the host (which gets it from the file system), called CAD-data. What these data are will be explained later. The results of the calculations are sent to the host.
- user interface. The user interface handles the communication between an operator and the host computer. It is necessary to give information about the status of the running process to the operator or to give commands to the host about starting the inspection of a new PCB. The data stream from the user interface to the host represents new CAD-data, which is necessary when inspection of a new PCB starts.
- file system. The file system is used to store data about the status of the process and to store CAD-data.
- transport system. The transport system brings PCBs to the laser scanner. After inspection of the PCB, it sorts the PCBs into approved and disapproved ones. The sorting is controlled by the host which gets the information from the data processing system.

2.2 Data processing system

A diagram of the data processing system is given in figure 2.2. The function of each block is:

- image acquisition & filtering. The image acquisition & filtering block has a pre processing function. The laser data are sent to this block. The data are sent through a look up table for conversion of the height and intensity data. This can be used to obtain better contrast. The other function of this block is to put the images on the data bus with the right control signals.

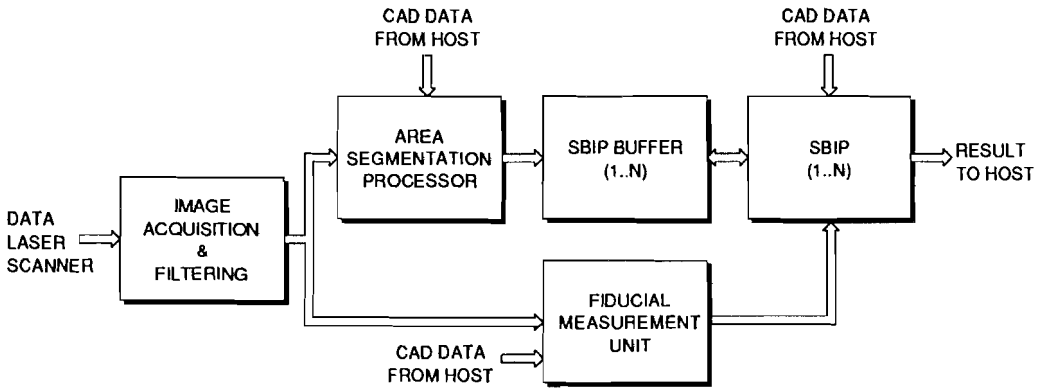


Figure 2.2 *Diagram of the data processing system*

- area segmentation processor (ASP). The function of the area segmentation processor is to buffer the data received from the acquisition and filtering block, distribute the data over the SBIPs and reduce the data send to the SBIPs. The distribution is done by combining Areas of Interest (AOIs). The AOIs are specified in the CAD-data and is received from the host. The combined AOIs (called screens) are sent to the SBIPs in a suitable format (512 x 512 pixels). The host controls to which SBIP the screen is sent. The laser scanner doesn't have to be stopped with the buffer and distribution function of the ASP.
- SBIP buffer (1..N). The SBIP buffer is used to store a screen so that the SBIP can be active continuously. In this way the memory of the ASP can be small.
- SBIP (1..N). The SBIP does the volume and the position calculations of the solder paste. These calculations are done by software algorithms and are therefor relatively slow. To compensate this, a number of SBIPs work in parallel. The host controls the assignment of screens to the SBIPs. The results of the calculations in the SBIPs are sent to the host.
- fiducial measurement unit (FMU). The position accuracy of the PCB in the laser scanner is limited. A small displacement can lead to a deviation which amounts to several pixels. The second problem is that the PCB isn't always perfectly flat. This gives a deviation that varies with the position on the print. The fiducial measurement unit, locates fiducials (characteristic shapes, for example corners in a copper track) using the CAD-data received from the host (originating from the file system) and calculates the displacement between the expected positions and the measured. The displacement vectors are sent to one of the SBIPs.

3 Hardware of the Area Segmentation Processor

As written in the previous chapter the ASP receives an image from the acquisition board and CAD-data from the host. It segments the specified AOIs and composes them into new screens so that only interested areas are sent to the SBIPs. Combining the AOIs into screens is done by software. The function of the ASP depicted in figure 3.1.

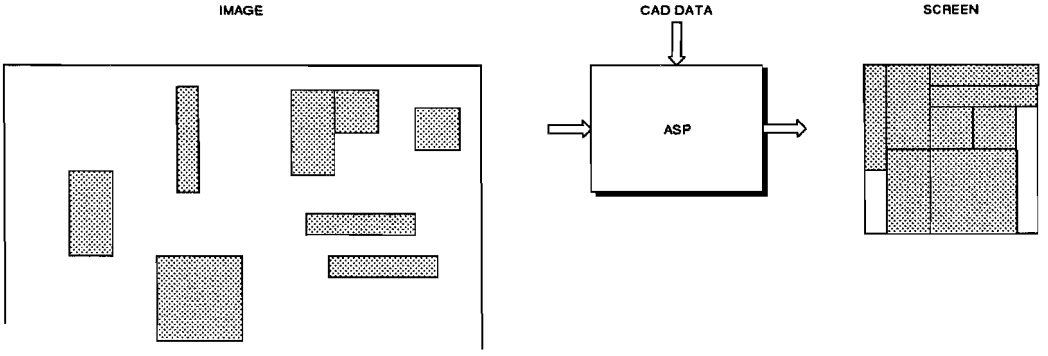


Figure 3.1 *Example of image to screen transformation*

In order to reduce the work for the ASP processor an image is divided into units. A unit is a square which contains 16, 64, 256 or 1024 pixels, depending on the chosen unit size[1] (4, 8, 16 or 32). The reason for this is to give the processor more time for other calculations. When a unit has to be saved, the processor must change the stored information only once per four (eight, sixteen or thirty two) lines. The relation between pixels, units and AOI can be seen in figure 3.2(B). In this figure the definition of an AOI is also given.

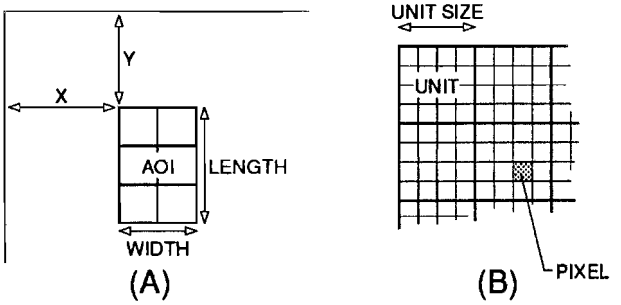


Figure 3.2 *(A) Definition of an AOI, (B) Relation between pixel, unit and AOI*

The hardware of the ASP is split into blocks. This is given in figure 3.3. Receiving and segmenting is done by the PI-bus interface and the image handler. After storing the AOIs into a circular buffer a screen is composed by the screen handler and sent to an SBIP by the PI-bus interface. A further explanation of the function of each block is given in the next paragraphs.

[1] It is an agreement that the length of a unit is called unit size.

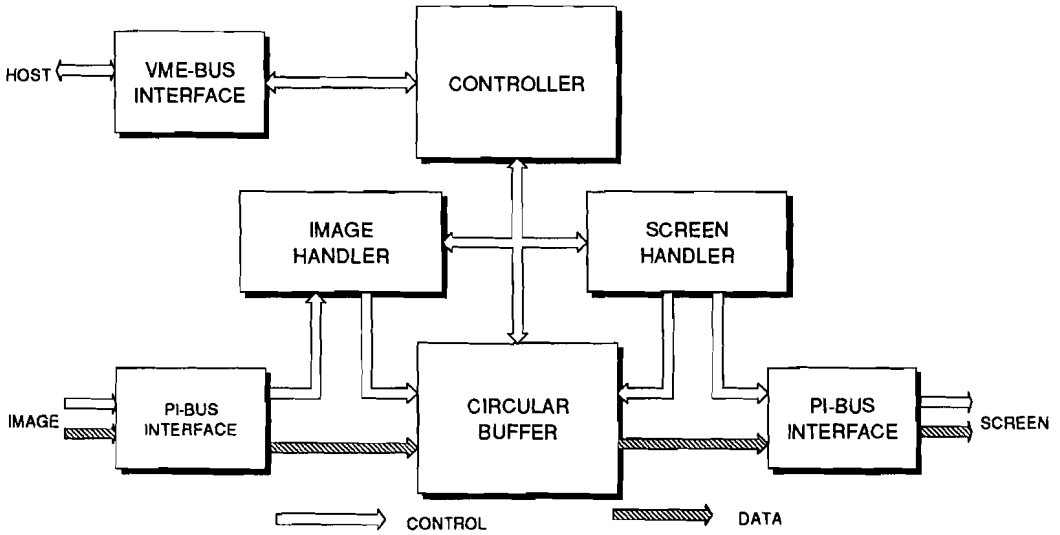


Figure 3.3 *Block diagram of the Area Segmentation Processor*

3.1 Controller

The controller controls the segmentation of the images and the composition of the screens. The processor used is a RISC processor. More about this processor in appendix A. The control block contains two RAMs, one for the program and one for the CAD-data. These RAMs are initially filled via the VME-bus interface. The tasks of the controller are:

- controlling the image handler
- controlling the circular buffer
- controlling the screen handler
- communication with the host (via the VME-bus interface)
- calculating the composition of a screen.

3.2 VME-bus interface

The functions of the VME-bus interface are:

- downloading the program
- downloading the CAD-data
- interrupt handling between the ASP and host
- transferring CAD-data from host to ASP
- transferring screen composition information from ASP to host.

During the downloading process, the processor is switched off and the VME-bus interface controls the data- and address bus of the ASP. Normally the processor can't be switched off very easily so when more CAD-data is needed from the host, the host puts this

information in a specific memory, the communication buffer. When sending CAD-data, the processor reads the communication buffer and puts the data in the RAM.

The communication buffer is also used when the screen composition is sent to the host. The screen composition data is data which specifies in which screen an AOI is placed and where in that screen the AOI is placed. Asking more CAD-data or signalling that composition data is ready in the communication buffer is done on interrupt basis.

3.3 Image handler

The segmentation takes place in the image handler. From the CAD-data the image handler knows which units have to be saved. Therefor it is necessary to give a unique identifier to every incoming pixel. This is done by a line counter and a column counter. The values of column counter and line counter can be seen as a coordinate. Dividing the counter values by the unit size will give the unique identity of the units. This can also be seen in figure 3.4.

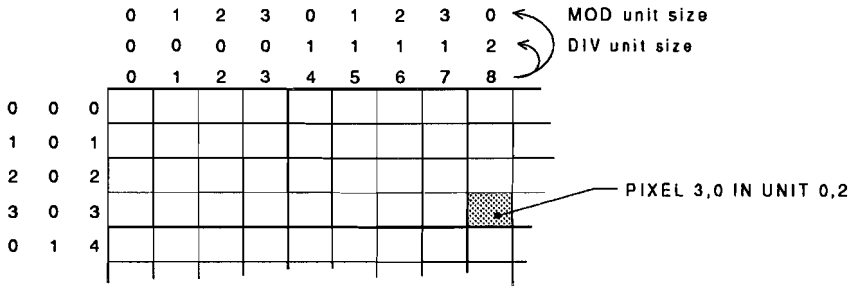


Figure 3.4 *Unique identifier pixel and unit*

The positions of the AOIs are known from the CAD-data. The processor has to convert this information to the current scan line. This is done by coupling an address to each unit and placing it into a line buffer. This address denotes the start address of each unit in the circular buffer (most significant part). If a unit doesn't have to be stored the start address will be set to a dummy value of zero. The least significant part of the address in the circular buffer is formed by the least significant bits of the line counter and column counter. The principle is shown in figure 3.5.

The line buffer is filled by the processor. Because the pixels arrive continuously two line buffers are used, one for filling the circular buffer and one is filled by the processor. The two line buffers swap their function when "unit size" lines are received.

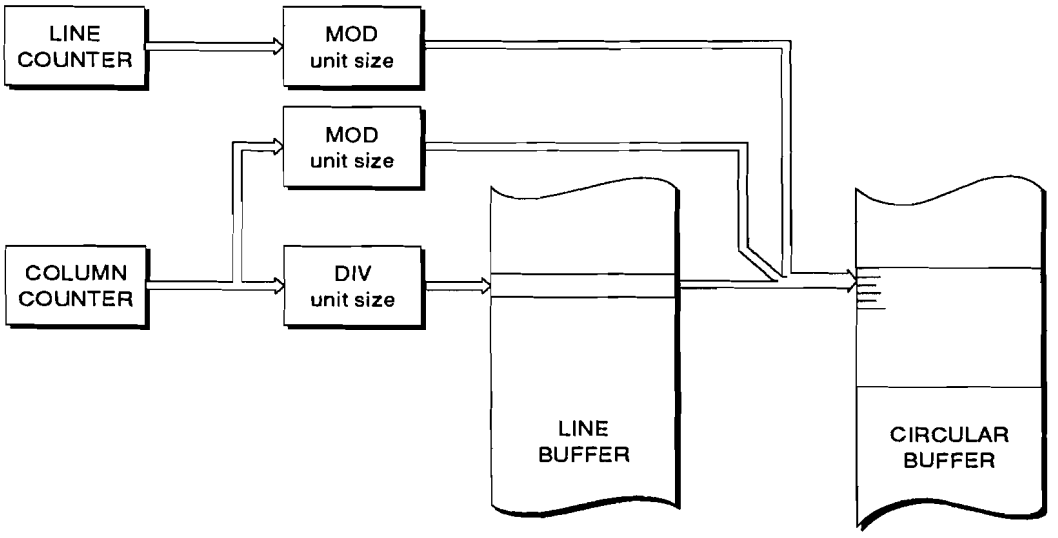


Figure 3.5 *Address generation for writing units*

3.4 Screen handler

After storage in the circular buffer a screen is sent to an SBIP. The screen handler composes a screen. A line counter and a column counter are used for identification of a pixel in the screen (same principle as image handler). The address for the circular buffer is now generated by a two dimensional Look Up Table (LUT). Here it is chosen for two dimensions because the LUT needs only 16 k addresses (512*512 pixels / minimal unit). When using a LUT in the image handler, the LUT should have more then 15 million addresses. The advantage of a LUT is that only once per screen the LUT has to be filled. The address generation of the screen handler is given in figure 3.6. The other function of the screen handler is to generate the screen control signals.

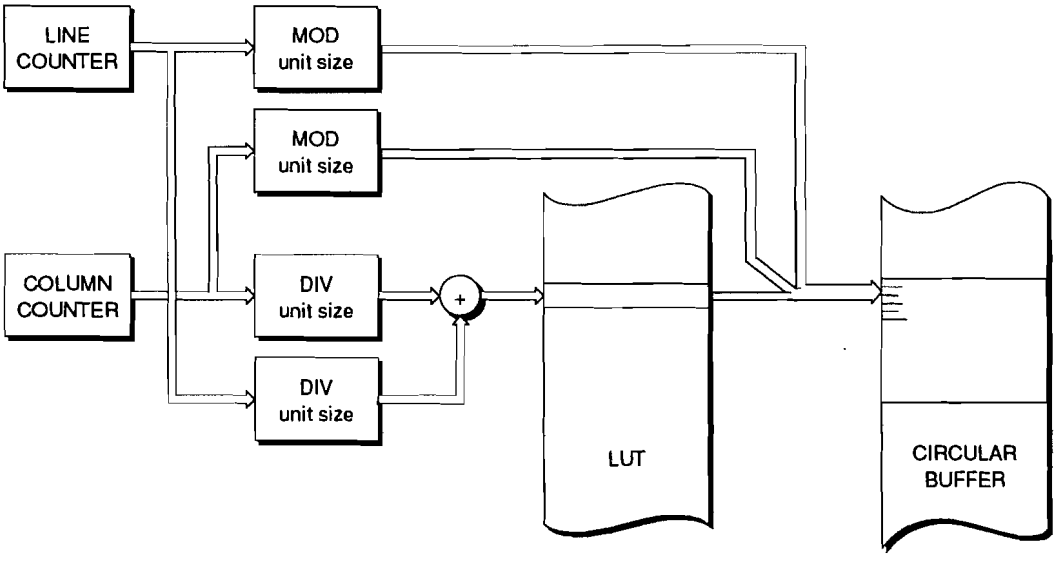


Figure 3.6 *Address generation for reading units*

3.5 Circular buffer

The circular buffer is used as storage for the AOIs of an image. The circular buffer contains 256k x 16 bits, so one screen fits into the memory. Each incoming AOI is stored in the buffer until it has to be sent as a part of a screen. The memory is circular and divided into eight blocks. The eight blocks are related to the hardware components. The RAMs are 32k x 16 bits separated I/O (two separated data busses per RAM). The advantage is that when writing one block the other blocks can be read. The principle of the circular buffer is given in figure 3.7.

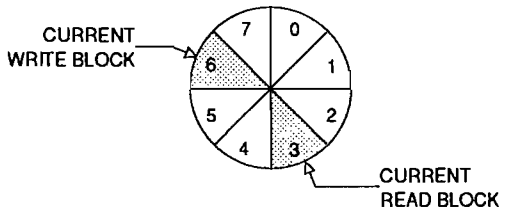


Figure 3.7 *Principle of circular buffer*

The blocks are random access which gives the opportunity to read and write in eight blocks randomly. In this way the screen composition doesn't depend on the sequence of the AOIs in the image.

The addresses are generated by the image handler and the screen handler. The number of bits of the addresses generated by the handlers is such that one RAM block can be addressed. Two crossbars lead the address to the correct RAM block. The controller configures the crossbars and sets the write and read block selection. A functional diagram of the circular buffer is given in figure 3.8.

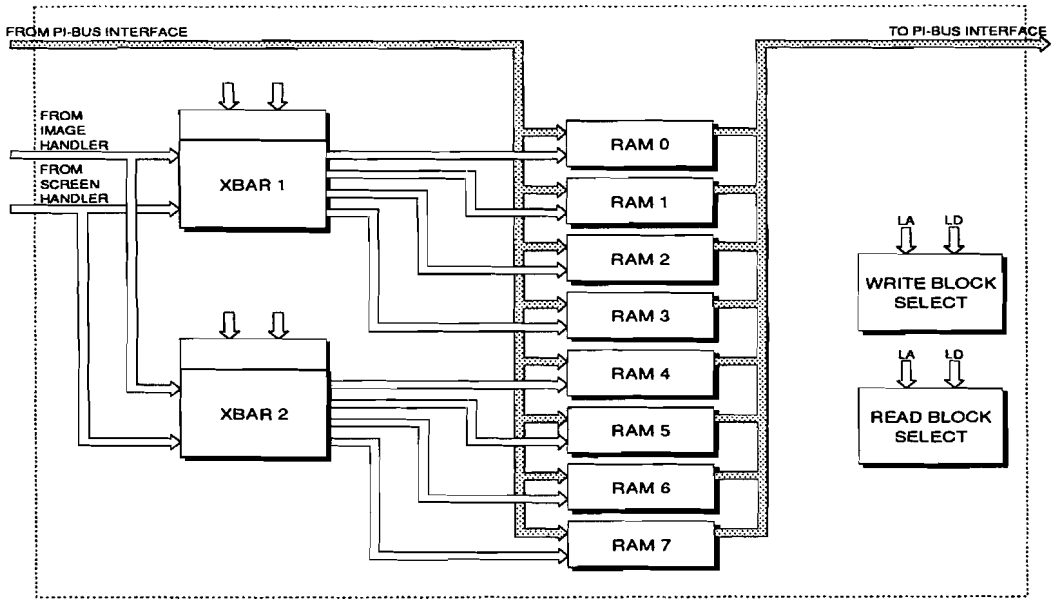


Figure 3.8 *Functional diagram of circular buffer*

4 Software for the Area Segmentation Processor

In the previous chapter the hardware of the area segmentation processor was explained. This chapter gives an outline of the software. This is done by schemes made with Promod. Promod is a highly automated system engineering environment to develop a detailed specification for systems. The solid lines in the schemes represent data flows, the dotted lines represent control flows, the circles represent functions, two parallel lines with data flows to or from it, represent data stores and the lines with control flows to/from it represent state machines (see figure 4.2). The functions in the schemes are implemented in hardware or in software.

A functional diagram of the ASP in Promod is given in figure 4.1. The Promod diagrams in this chapter are explained so far that the function of the software is evident. For more information see the report by P. Boots ([BOO]).

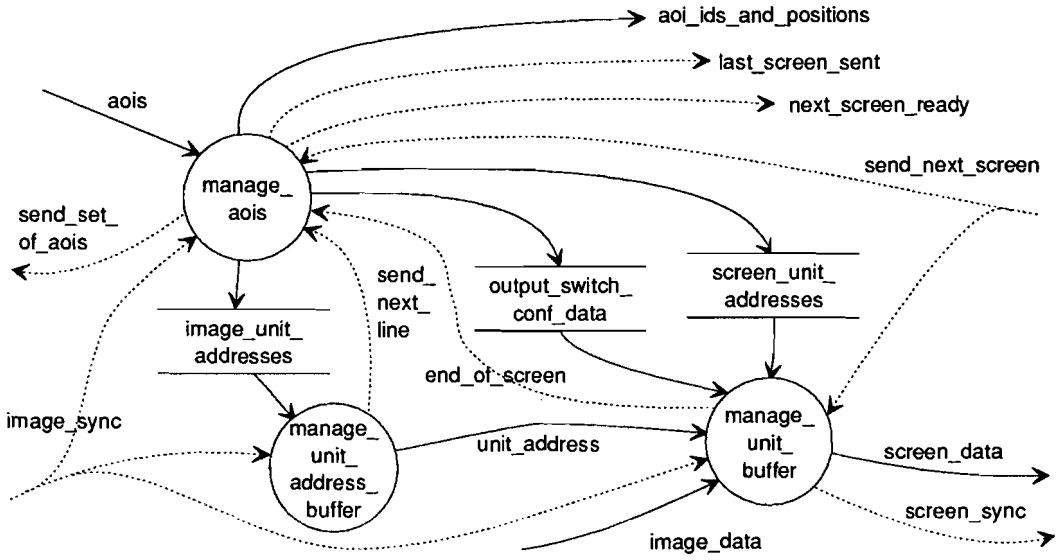


Figure 4.1 *Functional diagram of the Area Segmentation Processor*

The data flow image_data and the control flow image_sync come from the image acquisition and filter block (see figure 2.2). Screen_data and screen_sync go to one of the SBIPs (see figure 2.2). The other flows are to or from the host. A further explanation of the functions and flows is given in the following paragraphs.

4.1 Manage AOIs

The function manage_aois has several tasks. These are:

- calculating the screen composition, i.e. how and in which screen the AOIs are placed,
- calculating the line buffer content,
- calculating the screen content, which is stored in screen_unit_addresses,
- asking the host for more AOIs,
- signalling when a screen is ready,
- signalling when the last screen is sent and
- signalling in which screen the AOI is placed and its position in the screen.

Synchronising the screen calculations is done with send_next_screen. The relation between the tasks of manage_aois is given in figure 4.2.

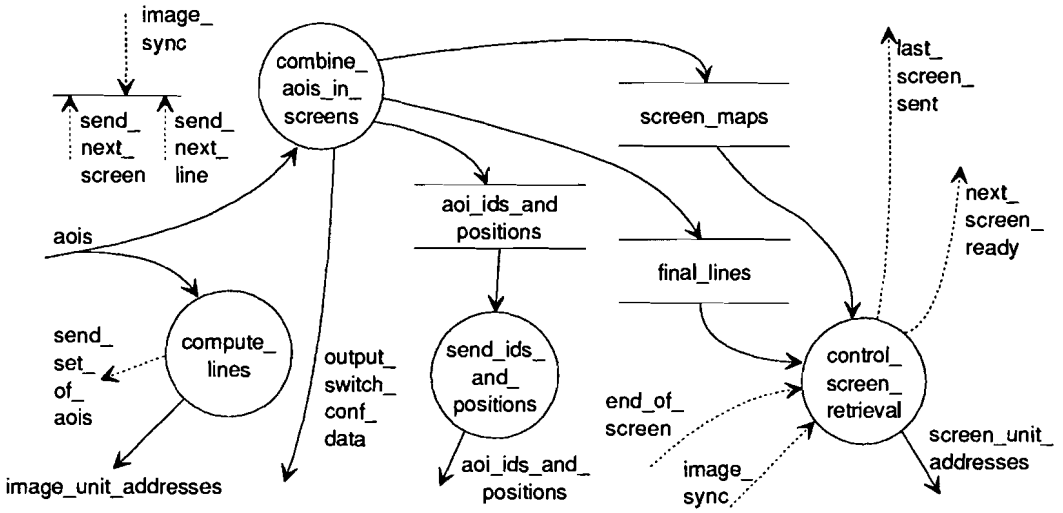


Figure 4.2 *Functional diagram of manage_aois*

All the functions in manage_aois are implemented in software.

4.1.1 Compute Lines

The function compute_lines is responsible for segmenting the image. It computes the addresses on which the units of the AOIs are stored. The addresses are stored, line by line, in image_unit_addresses. These lines are later placed in the line buffer. A unit of an AOI gets a unique address within a block of the circular buffer. A unit which doesn't have to be saved receives address zero. The addresses for the units within an AOI are given in the order they arrive. Units which lay in two or more AOIs (overlap of AOIs or doubles) get only one number. When the list of AOIs becomes to short a signal send_set_of_aois is sent to the host. An example of the address assignment is given in figure 4.3.

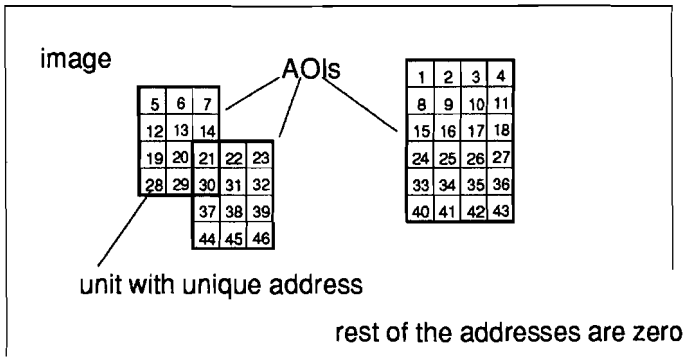


Figure 4.3 *Addresses for the units in an AOI*

Calculating a line for image_unit_addresses is done by checking for every unit in the image if that unit is in an AOI. The list of AOIs is sorted to shorten this searching. It is first sorted on increasing y-coordinate because the image arrives row by row (the y). AOIs with the same y-coordinate are then sorted on x-coordinate because the units in one line arrive with increasing x-coordinate.

The store image_unit_addresses is stored in a RAM on the ASP. Every line needs maximal 3000 half words (16 bits) of the RAM. Calculating and storing the whole image (maximal 5000 lines) in advance would cost too much RAM. Therefor only a few lines are calculated in advance.

As written in paragraph 3.5 the circular buffer consists of eight blocks. Switching or reconfiguring these blocks is activated by setting bit 11 of the unit address. Setting this bit is done by compute_lines. Switching can only be done when there is enough time to reconfigure the blocks. So the switch bit may only be set:

- at the end of a line or
- at the last unit of an AOI in a line or
- between two AOIs when the time between them is enough.

Hereby it is assumed that the time between the end of the image line and the start of the next line is enough to reconfigure. This is also depicted in figure 4.4.

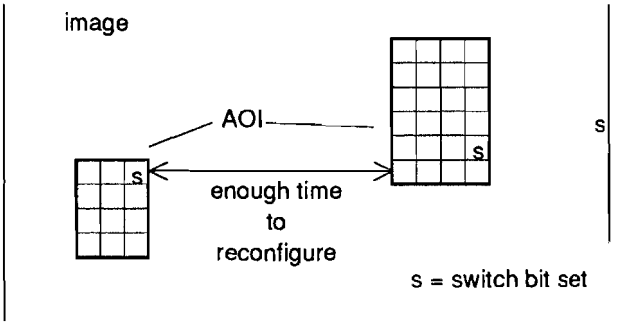


Figure 4.4 *Examples of places for the switch bit*

4.1.2 Combine AOIs In Screens

The function combine_aois_in_screens combines the received AOIs into one or more screens. An example has already been given in figure 3.1 where the function of the ASP is depicted. This combining of AOIs results in four data flows:

- screen_maps. Each screen map is a two dimensional array holding addresses from the units of the AOIs in the screen. This screen map can later be placed in the LUT. The screen map of figure 4.3 is given in figure 4.5.

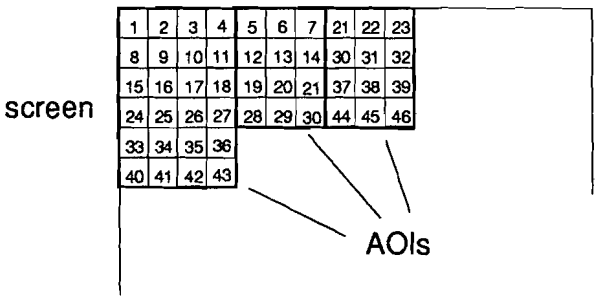


Figure 4.5 *Example of a screen map with the addresses*

- output_switch_conf_data. The circular buffer is divided into eight blocks. Therefor it is necessary to reconfigure the crossbar switch. In the screen map, bit 11 of a unit_address, activates this switching. In output_switch_conf_data is specified to which block of the circular buffer must be switched.
- aoi_ids_and_positions. Where the AOIs are placed in the screen depends on the filling algorithm. Because of the calculations, the SBIP must know which AOI is in which screen and on which position in the screen. This information is held in aoi_ids_and_positions.
- final_lines. In order to signal the SBIP that the next screen can be sent, the final_lines of each screen i.e. the image line number that contains the last pixels of the screen, must be known.

In compute_lines the switch bit could only be set when there is enough time. Here it must be possible to set the switch bit of every unit address, because the filling algorithm decides where the AOI is placed. Thus it is possible that one AOI is stored in block 0 of the circular buffer and that the AOI which lays next to it in the screen is stored in block 1. The hardware is made so that the sending of a screen stops if the switch bit is detected. The crossbar can then be reconfigured by the software and the software restarts the sending.

Combine_aois_in_screens searches a way to fill the screens with AOIs. This filling has conflicting demands:

- The screen must be filled as much as possible because the unused units take time when a screen is sent to the SBIP.

- The calculation time and algorithm length increase a lot when a high filling grade is wanted. Another aspect here is that when the "best" fitting is found units are kept for a long time in the circular buffer. The circular buffer on the ASP is limited so it is desirable that the units are sent away as soon as possible.
- The traffic on the VME-bus must be limited because this will delay the other VME traffic.

With these demands the function combine_aois_in_screen can be located and performed in three different ways:

- An off line process on the host. The host calculates the filling of the screens and the unit addresses for the line buffers and LUT. Advantage of this is that there is enough calculation time to find the "best" fitting and this has to be done only once. Disadvantage is that a mass store is needed and that the down loading gives an enormous VME traffic. Remember that an image can have 5000 lines with 3000 units in it. This means that in this case 15 mega addresses must be down loaded plus 1.5 mega addresses for the screens, assuming that 10 % of the image are AOIs and that the screens are full.
- An off line process on the host for the filling of the screens and the address calculation on the ASP. The advantage is that there is enough calculation time and the filling calculation has to be done only once. The address calculation must be repeated for every image. The only data that must be down loaded are the AOIs and the positions in a screen.
- Everything on the ASP. Disadvantage is that the search algorithm can't be complex because of time and algorithm length. Advantage is that only the AOIs must be down loaded to the ASP and the positions must be loaded to the host.

Testing must decide what the best option is but so far it seems that the second option is the best.

4.1.3 Send Ids And Positions

The function send_ids_and_positions signals to the SBIPs which AOIs are in the screen that is sent to that SBIP and the positions of the AOIs within the screen. The sending is activated with send_next_screen. This routine runs on the host if the combining algorithm runs on the host.

4.1.4 Control Screen Retrieval

The function control_screen_retrieval waits until the received image line is greater than the next final line in final_lines and then signals next_screen_ready and fills the store screen_unit_addresses (the LUT) with information from screen_maps. These actions may be started only if manage_unit_buffer is not sending a (part of a) screen (seen by

end_of_screen).

Combine_aois_in_screens calculates screen maps which are stored in memory. Parts of these maps are later stored into the LUT if that part of the LUT is free. One screen map takes 16 k half words. The memory on the ASP is not so big, so only a part of the screen map can be stored here. A functional diagram of control_screen_retrieval is given in figure 4.6.

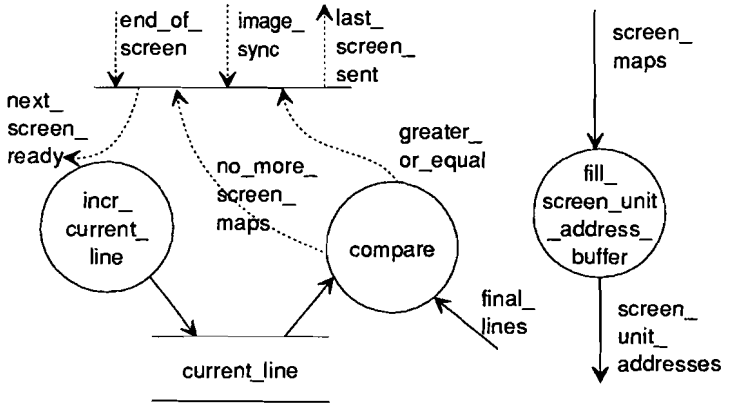


Figure 4.6 *Functional diagram of control_screen_retrieval*

The function incr_current_line updates the store current_line every time image_sync is activated. Current_line keeps the number of scanned lines of the image. When current_lines is greater then final_lines, compare signals greater_or_equal what results in an active next_screen_ready. When the last element of final_lines is used no_more_screen_maps is activated. With this signal last_screen_send is activated after end_of_screen returns. Fill_screen_unit_address_buffer takes a part of the screen_map and places it into the screen_unit_addresses.

4.2 Manage Unit Address Buffer

Manage_unit_address_buffer gets one line of the image_unit_addresses, transmits unit_address one by one and asks for the next line at manage_aois when a line is sent. This is done under control of the image_sync signal. A functional diagram of manage_unit_address_buffer is given in figure 4.7.

One line of image_unit_addresses is stored into one of the two line buffers "unit_address_buffer_1" or "unit_address_buffer_2" by "fill_ua_buffer". If a new line is wanted the signal send_next_line is sent to manage_aois. Send_next_line is activated after every 4, 8, 16 or 32 lines of the image depending of the chosen unit size. The "empty_ua_buffer" functions empties the store.

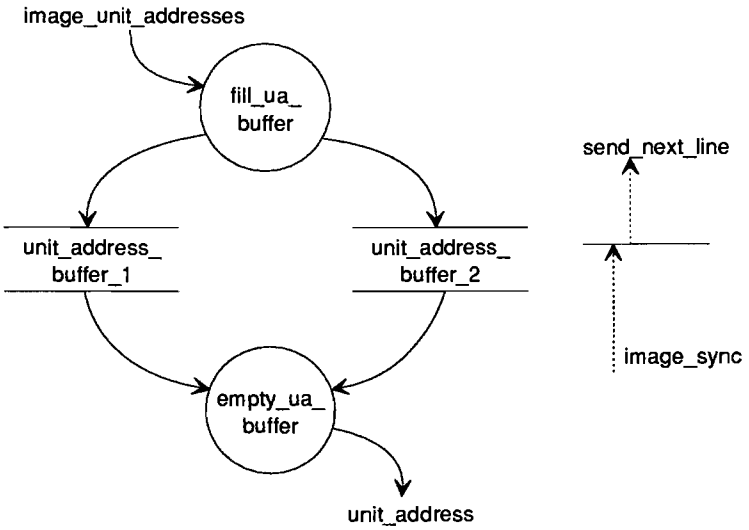


Figure 4.7 *Functional diagram of manage_unit_address_buffer*

The stores are RAMs (the line buffer in figure 3.5) of which each can hold one image line. The empty function is implemented in the hardware. Counters generate an address for reading the RAMs. The filling function is implemented in software. The fill function copies one line of unit_addresses to the RAM. Multiplexers control which buffer is filled and which buffer is emptied. The signal send_next_line is implemented as an interrupt to the processor on the ASP.

4.3 Manage Unit Buffer

Manage_unit_buffer stores the pixels from image_data at addresses specified by unit_address of manage_unit_address_buffer at a rate controlled by image_sync. The other function of manage_unit_buffer is sending screens under control of the screen map in screen_unit_addresses. The sending starts after the signal send_next_screen. When a screen is sent, manage_aois is signalled by end_of_screen. The functional diagram of manage_unit_buffer is given in figure 4.8.

Keep in mind here figure 3.8. The stores unit_block_0 to unit_block_7 in figure 4.8 are RAM0 to RAM7 in figure 3.8. The functions direct_to_current_write_block an collect_from_current_read_block are performed by the two crossbars (xbars). Generate_pixel_in_unit_address_input and -output are two counters which generate an address used to write or read the pixel within a unit. Decode_image_unit_address receives the unit_address and signals by reconf_input_code_encountered (an interrupt to the processor) if the reconfigure code (bit 11 of the address) has encountered. Decode_screen_code_address does the same but then for screen_unit_addresses.

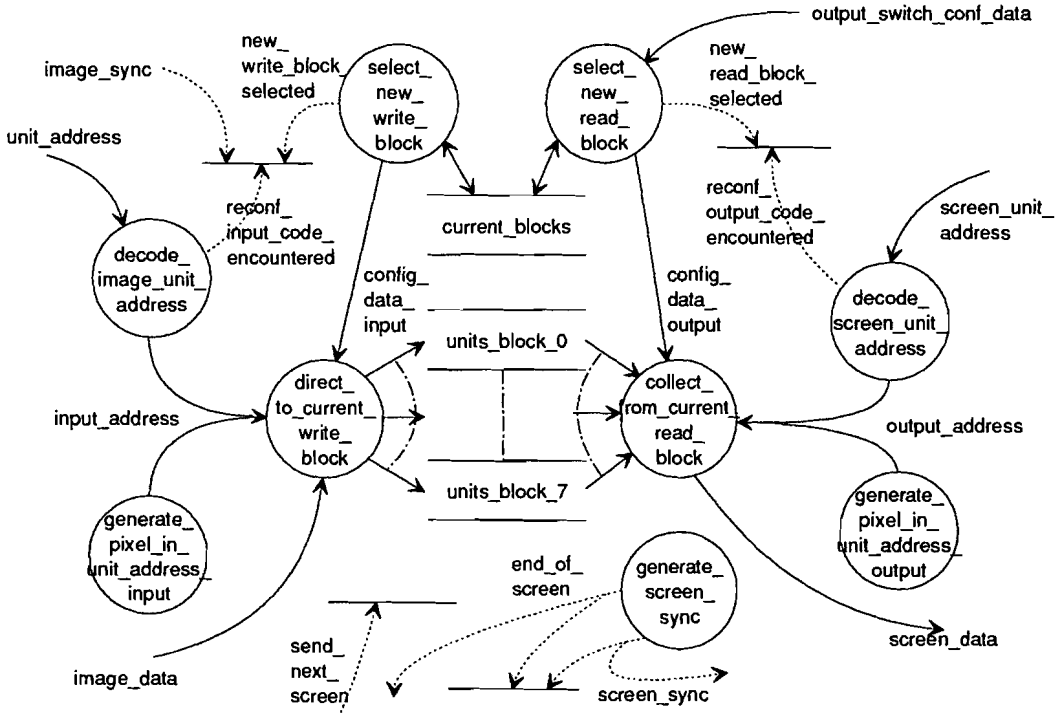


Figure 4.8 *Functional diagram of manage_unit_buffer*

Select_new_write_block is implemented in software. As soon as a block is full or almost full a new write block must be selected. Select_new_write_block is activated by reconf_input_code_encountered. Select_new_read_block does the same, but here the data to which block must be switched comes from output_switch_conf_data. The selected read and write block are stored in current_blocks. This store is a hardware store, which enables read en write signals for the appropriate block and a variable for the program.

5 Implementation of the hardware

The work during my graduation period was the realisation of the Area Segmentation Processor. First the timing conditions (set up time, hold time, delay time) were theoretical checked with worst case calculations. Secondly the hardware was measured. This to check the wire wrap connections. Where necessary improvements were made.

For testing the ASP several programs are written. The programs which run on the ERM system (the host), are written in C. The programs which run on the ASP, are written in assembly or in C.

In the following paragraphs several parts of the hardware are discussed. Changes and additions in the design which are mentioned in the following paragraphs are made by the author of this report. Further outlines about the programs for testing the hardware are given. In appendix B the schemes of the area segmentation processor are given.

5.1 VME-interface

The different cards in the system (host, acquisition card, ASP, SBIPs) communicate via a VME-bus. The image and screen data are sent via two picture busses. VME (= Versa Module Eurocard) uses a protocol for data transfer and interrupt handling (see [VMS]). For correct handling of the protocol a special prototype board (see [VMP]), was used. Data send to the ASP appears on the prototype board. Placing the data on the correct address is done by the VME-interface (scheme in appendix B.9). Getting data from the correct address on the ASP, starting an interrupt action to the host and starting the acknowledge of an interrupt is also done by the VME-interface. The hardware of the interrupt handling is depicted in appendix B.1 (upper right and lower right in the scheme). Testing and improving the VME-interface had to be done first because downloading programs to the ASP can only be done when this part works.

The circuit of the VME-interface was built first with discrete components and was asynchronously clocked. It was very difficult to add new functions and very difficult to get the correct functionality. For these reasons a redesign has been made. The circuit is implemented in an Erasable Programable Logic Device (EPLD). The design has been made, simulated and programmed with MAX+plus.

Some functions are added to the new design. The several functions are reached by a card address plus an offset address. The different functions with the offset address are given in table 5.1.

Table 5.1 *Functions of VME-interface with offset address*

Function	offset address
disable CPU	2
enable CPU	4
read/write counter RAMs	6
read/write counter communication buffer	8
read/write communication buffer	10
read/write bit 15..0 program RAM	12
read/write bit 15..0 CAD RAM	14
read/write bit 31..16 program RAM	16
read/write bit 31..16 CAD RAM	18
write bus allocation register	20

The address range for the host to access the ASP is only 2 k big. This is not enough to map the whole memory of the ASP within the address range of the host. Therefor a counter is used which generates the address on the ASP during downloading. These counters have to be filled with the start address.

The ASP has a 32 bit data bus. The ERM system has only a 16 bit data bus. Downloading data is done by first downloading the lower 16 bit word and than the higher 16 bits. After this second action the data is written in the RAM and the counter is incremented. Reading is done by first reading the lower 16 bits and than reading the higher 16 bits.

Testing the VME-interface is done by writing (downloading) data to the RAMs on the ASP, reading back and comparing the written and read data. With these tests also the read and write signals for the RAMs are verified.

The circuit for the interrupt from the ASP to the host and from the host to the ASP was originally not designed. This part is necessary, for several signals (send_set_of_aois, last_screen_send, next_screen_ready and send_next_screen in figure 4.1) and therefor added.

The communication buffer is used for sending information between the host and the ASP at the time the ASP is running. After power up the data is downloaded to the ASP and the communication buffer isn't used then. The processor must be disabled at that time. When the processor is started the processor can't be disabled because after enabling, the processor won't continue were it was stopped. To send information to the ASP, if the

processor runs, the communication buffer is used. The communication buffer is tested by means of writing data into it and reading back. This writing and reading is done by the host and the ASP.

5.2 Controller

A SPARC RISC processor CY7C611 controls the ASP. This processor works with a clock frequency of 25 MHz. More information about the processor, especially for programming the processor in assembly, is given in appendix A.

When downloading data to the ASP or reading data from the ASP to the host, the processor must be switched off (disabled). All output drivers are then in high impedance state (tri-state). Therefor it is necessary to connect pull up or pull down resistors at all outputs of the processor which are used in other parts of the circuit.

Two RAMs are connected to the processor. One to store the program and one to store the CAD-data. After downloading, the processor is started by the reset circuit. To test the communication between processor and RAM a program can write a certain data pattern into the RAMs (except the part where the program is stored). The host can read and evaluate the data.

To give information about the status of the program there is a circuit which communicates with a terminal (RS232). When a program is running it can be useful to know in which function the program is and what the values of certain variables are. The conversion from parallel to serial, visa versa, is done by a DUART. The timing of the selection, write and read signals for the DUART are improved because they did not fulfil the specifications.

An other problem which showed up when working with the DUART was that the RAMs are not byte accessible. The program RAM is only 32 bit accessible and the CAD RAM is 16 bit accessible. The standard function "printf" in C, which prints text on the terminal, writes bytes to the RAM to store temporary information. Writing one byte in the program RAM overwrites also three other bytes (the ones with the same address except the two least significant bits). Routines which use byte or half word variables can thus not be used on the ASP. New routines are written to solve this problem because it is not possible to make the RAMs byte accessible.

Not all the components are fast enough for the used clock frequency. Therefor there is a circuit that can stretch the clock. In figure 5.1 an example of the stretched clock is given with the clock 2 times stretched. Bit 23 of the address bus enables the stretch circuit. The number of stretch cycles is the value of the difference: 8 - the value of bit 22..20.

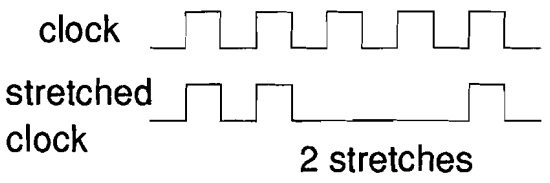


Figure 5.1 *Example of a 2 times stretched clock*

The different components are selected with bit 19..16. In table 5.2 the different functions are given with the base addresses.

Table 5.2 *Function selection with base addresses*

function	base address bit 23..0	function	base address bit 23..0
program RAM	0x000000	enable continue screen	0x080002
CAD RAM	0x010000	line buffer	0x090002
DUART	0x920003	unit size/mode	0x0a0003
crossbar 1	0xf30002	interrupt host	0x8b0003
crossbar 2	0xf40002	communication buffer	0xfc0002
LUT	0x050002	line buffer selection	0x0d0003
LUT selection	0x060003	data circular buffer	0x8e0002
enable send screen	0x070002	address circular buffer	0x8f0002

In the previous chapter a number of control signals are given. Reconf_input_code_encountered, reconf_output_code_encountered, end_of_screen, next_line and the signals from the host are implemented as interrupts to the processor. The interrupt handler converts an interrupt into an interrupt address. This handler is implemented in an EPLD. After power up all flip flops in the EPLD should be zero. Measuring learned that this was not so. Therefor a reset signal is connected to the flip flops in the EPLD so that after the power supply is stable the flip flops are reset.

5.3 Image handler

Multiplexers decide to which line buffer the processor can write and which line buffer is read by the counters. The position of the multiplexer depends on a flip flop. In the original design the processor was not able to control that flip flop. After power up the value of the flip flop is random 0 or 1, so which line buffer is selected for the processor is unpredictable. It is necessary to control the flip flop because the line buffer which is connected to the counters, must be switched to the RAM block in the circular buffer by

the crossbars. Therefor a circuit is added which enables the processor to set the selection of the flip flop depending on data bit 0 (base address line buffer selection). In table 5.3 is the selection given with data bit 0.

Table 5.3 *Line buffer selection with data bit 0*

data bit 0	processor to:	counter to:
0	line buffer A	line buffer B
1	line buffer B	line buffer A

The line buffers are tested by writing data in the line buffers and then reading the data back. Reading back can not be done directly because the outputs of the line buffers are not connected to the data bus of the processor. This is solved by adding drivers between the address lines of RAM block0 and the processor data bus. The processor addresses the driver (the offset of the address is lead to the line buffer) and reads the data of the line buffer through the crossbars (when configured in the right way).

The RAMs of the line buffers have only a chip select input and a combined read/write input. When reading or when writing while the write pulse is not active the output drivers are active (an other RAM is not possible as separate I/O is needed). The combination of bit 11 of the line buffers is the reconf_input_code_encountered signal from figure 4.8. When one of the bit 11 is set, the processor is interrupted. The addresses of the processor are always lead to one of the line buffers. Therefor it is possible that an address of the processor is an address of which bit 11 is set. This will interrupt the processor but the processor may only be interrupted when the counter reaches such an address. Components are added to the scheme so that only an interrupt occurs from the line buffer that is connected to the counters and only after the data is valid.

5.4 Screen handler

Testing the LUT is done in the same way as testing the line buffers. Data is written in it and then read back through the address driver of RAM block0 of the circular buffer (this is the reason why they are placed after the crossbars).

Reading the LUT back works only when the screen clock runs because a latch between the LUT and the crossbars is clocked by the screen clock. The screen clock (defines the pixel rate) is a clock which is controlled by screen handler. When the crossbar must be reconfigured this clock is stopped by bit 11 of the LUT (reconf_output_code_encountered in figure 4.8). After power up when the LUT isn't initialised, the clock can be stopped

when an address appears where bit 11 is one. To solve this problem two things are changed:

- first: bit 11 may only interrupt the processor and stop the clock when the counters are connected to the LUT
- second: the EPLD screen handler is changed so that there is a possibility to start the clock.

This second improvement is necessary because the clock can still be stopped when bit 11 of address 0 of the LUT (0 is start value of the counters) is 1.

When the clock is stopped by bit 11 the processor can restart the screen sending if it writes to address "enable continue screen" in table 5.2. This is not useful when the screen clock is stopped after power up. Therefor data bit 0 is used now when writing to "enable continue screen":

data bit 0 = 0: continue sending
data bit 0 = 1: enable only the clock

5.5 Circular buffer

The circuit that generates a write pulse for the RAM blocks was not designed. This had to be done first. Further the timing of the write pulse for the crossbar is improved.

As written in the two previous paragraphs there is an address driver which can read the addresses of RAM block0. The processor can't read (in the first design) the data that is written in the circular buffer. This makes it very difficult to test if there is an error made when writing the data or when reading the data. To solve this problem a second driver is added which connects the output data bus (screen data bus) to the processor data bus. To read the data a number of actions has to be done. For example reading the data in the first block sequentially the following action must be done:

- the RAM block must be selected
- the line buffer must be filled
- the crossbars must be configured in the right way.

The line buffer has 11 data lines connected to the crossbar. A RAM block of the circular buffer has 15 address lines. Normally 4 address lines (when the unit size is 4) come from the counter which generate the addresses within a unit. Now these address lines must be set by the crossbars. This is possible because the crossbar can set every output in different mode: flow through (used normally), pipeline, fixed output levels (0 or 1). With the line buffer outputs connected to the first 11 outputs of the crossbar and the following 4 outputs set to zero the first part of the RAM block can be read. Reconfiguring the 4 output makes it possible to read the other parts of the RAM block.

5.6 Picture bus interface

In the first design there was only one input channel on which the image data is received from the acquisition board and one output channel on which the screen data is sent to the SBIPs. In the system there are two channels and every card is made so that the host decides on which channel each card receives data and on which channel it may send data. To fulfil this option the picture bus interface is extended with components and the VME-interface handler is extended with an address that writes data in the bus allocation register. This bus allocation register selects the read and write channel.

6 Implementation of the software

In this chapter some remarks on the implementation of the software are given. A program which receives and sends images of 512x512 pixels is used for testing the ASP hardware in the system. In the first paragraph an outline and remarks are given about this program. The second paragraph gives some points for the implementation of the segmentation software. This software is not completely built because there was not enough time to do this. The reason herefor was that the improvement and implementation of the hardware took more time than was expected. The third paragraph shows how C programs can be improved so that less time is needed for execution.

6.1 Software 512x512 images

The first major software that is written, except the small test programs, is for receiving 512x512 images. These images are just as big as the screen which is sent by the ASP to the SBIPs. The tasks of the ASP are:

- receiving the image from the acquisition board,
- storing the image in the circular buffer and
- sending the screen to a SBIP.

Here the image doesn't have to be segmented and the CAD-data don't have to be downloaded.

The purpose of this program is to integrate the ASP into the system and test the hardware of the data path (the data path from the PB-bus interface to the circular buffer and from the circular buffer to the PB-bus interface). The data written in the circular buffer can be tested with the method explained in paragraph 5.5. The addresses lead to the circular buffer are tested in an earlier phase.

The program is mostly written in C. Only the trap table, the processor initialisation routine and the routines which communicate with the terminal for debug information, are written in assembly. The sequence of the program is given in figure 6.1.

The line buffers and the LUT are cleared so that no interrupts occur (this is also protected by hardware except for address 0). After the interrupt level is set an interrupt can occur which was pending from before the line buffer and LUT were cleared.

The program sequence when receiving an image is given in figure 6.2. After "unit size" lines are received, an interrupt occurs. The interrupt routine that is started then, selects a new write and read block if necessary, configures the crossbars so that the other line

```
initialisation:     processor
                    DUART (terminal)
                    crossbars
clear:              line buffers
                    look up table
                    communication buffer
set:                unit size
                    interrupt level
initialisation:     variables
receive:            image
test:               image data
send:               screen
```

Figure 6.1 *Sequence of 512x512 program*

buffer is connected to the RAM block and fills the just used line buffer. Selecting a new read block is necessary because the write and read block may not be the same when writing to that block.

```
fill the line buffers
select the first write block
configure the crossbars
wait until image is received
```

Figure 6.2 *Sequence of receiving the image*

```
fill the LUT
for i = 0 .. 7 do
        select read block i
        configure the crossbars
        start sending the block
        wait until block is sent
```

Figure 6.3 *Sequence of sending the screen*

The program sequence of sending an image is given in figure 6.3. The listing of the program is given in appendix C. The program given here receives the whole image and then starts sending the screen. Of course this program can be improved. For example sending a block can start after the block has been received and the host can be signalled to start sending an other image. This is not done because it is not necessary for testing and it makes the software more difficult.

6.2 Segmentation software

In chapter 4 an outline of the segmentation software for the Area Segmentation Processor is given. The combining algorithm and the address calculation can run on the host or on the ASP. We choose to run the combining algorithm on the host and the address calculation on the ASP because this has the most advantages (much calculation time

available on the host, it is done only once and it gives a low VME-traffic). In this situation the specifications of the AOIs (x, y, length and width) and the position within the screen (screen number, x and y) must be downloaded.

The combining algorithm runs only one time for every PCB type and can be run in advance. Therefor the time needed for the calculation can be large. The limit for the algorithm is the time that AOIs are kept in the circular buffer. The circular buffer can hold only one screen. Because the addresses for the units of an AOI (in compute_lines figure 4.2) are given the way they arrive, the AOIs in one block must be sent away before this block is selected again as write block. This to prevent a large adminstration for the addresses which may not be used for writing new AOIs in that block.

The first implementation of the combine algorithm (combine_aois_in_screens in figure 4.2) is a simple one. The AOIs are placed in the screen in the way they arrive (and are sorted). When an AOI doesn't fit at the end of the row, it is placed on the next row below the largest AOI of the "full" row. When the AOI doesn't fit into the screen, because it is almost "full", the AOI is placed in the next screen and the "full" screen is not filled further. This algorithm doesn't fill the screen optimal (fill grade between 40 and 60%), but this is enough for the first tests.

Combine_aois_in_screens runs partly on the host (the combining) and partly on the ASP (the address generation). The address generation must know of every unit which address it has been given in compute_lines. This can be done by storing the addresses given to the units of the first column of each AOI. With these addresses the rest of the addresses in the AOI can be calculated. The problem hereby is that the length of the AOIs are different. To store each first column, dynamic memory allocation is necessary. The standard routine herefor is available but a difficult memory administration will be necessary. This adminstration must register for which AOI memory is allocated and where this is allocated. Using a simple allocation and free routine will result in a memory of which small parts are occupied and small parts are free. The second possibility is to recalculate the addresses given to the units. This means that compute_lines is done twice. The third possibility's implementation is given in appendix D. Here the address generation of the line buffers and the LUT are combined. Compute_lines computes the line buffers one by one. At the same time lines of the screen map are calculated. When an address is given to a unit of an AOI in the line buffer, this address is also placed in the screen map. The calculations here use the minimum unit size (4). When all AOI are for example a multiple of 8, it is useful to use a unit size of 8.

The memory on the ASP is limited. CAD-data, image lines and screen lines are stored in the CAD RAM. This means that only a limited number of AOIs and lines can be stored. This limitation restricts the combining algorithm. If an address of a unit has to be placed on a screen line which isn't used so far and all the memory for the screen lines is used,

the oldest screen line is placed in the LUT. After that addresses can't be placed on that line any more. This restriction is no problem for the first implementation of the combine algorithm.

```
for y = 0 to PCB length
   for x = 0 to PCB width
      for every AOI so that AOI.y <= y < AOI.y+AOI.length
         if (x,y) in AOI
            place address in image line (also handle the switch bit)
            if screen lines all occupied
               copy line to LUT
            place the address in the screen line
```

Figure 6.4 *Sequence of the address generation*

The sequence of the address calculation is given in figure 6.4. The inner loop "for every AOI ...." is necessary because AOIs can overlap or be double. This loop is not necessary if only the line buffers are calculated because a unit which is member of two or more AOIs, gets only one address. This loop is only necessary for placing more than one address in the screen map. This loop is restricted to the AOIs which have units on the current image line (the y variable of the loop).

6.3 Programming in C

The biggest part of the program for the ASP can be written in C. Programming in C is of course easier then in assembly. When a program takes too much time, improvements can be made before the program has to be written in assembly (and hope that it is fast enough in assembly). In this paragraph the implementation of the fill_ua_buffer_1 of figure 4.7 is given as an example for improvements. This routine copies a line of image_unit_addresses into the line buffer. The listing of this function is given in figure 6.5. The assembly listing is given in appendix E.1. The 4*i is necessary so that the addresses written on, are word aligned (the processor has a 32 bit data bus; see appendix A.4).

Several parts of this routine can be improved, which results in a lot of time saving. This improvements costs a few variables (read: working registers of the processor). In this function this is no problem because there are not many variables. If a function has more variables than working registers, variables are stored on the stack in the memory. Variables stored on the stack cost of course more time too access than variables in working registers.

The first improvement is using a local variable instead of the global variable current_image_line. Current_image_line is the index to the oldest, in advance, calculated

```
#define LINE_LENGTH                3000
#define BUFFER_SIZE                 4
#define LINE_BUFFER_BASE_ADDRESS   0x090002
extern int current_image_line;
extern int image_unit_addresses[BUFFER_SIZE][LINE_LENGTH];

void fill_ua_buffer ( )
   {
   int i;
   short *linebuffer;

   linebuffer = LINE_BUFFER_BASE_ADDRESS;
   for( i = 0; i < LINE_LENGTH; i++ )
      *(linebuffer + 4*i)= image_unit_addresses[current_image_line][i];
   }
```

Figure 6.5 *Listing fill_ua_buffer*

line buffer. This global variable is stored in the memory and it takes two instructions (3 clock cycles[1]) to load the value in a register. When this variable is given as a parameter to the function, the variable doesn't have to be loaded in the function (only once when calling).

The second improvement is not to use the two dimensional array image_unit_addresses but only a one dimensional array. Accessing an element in the two dimensional array is done by calculation of the address: "image_line + 12000*current_image_line+(i<<2)" (<< means shift left) and then perform an action to that address. The array is an integer array so 12000 comes from 3000 elements * 4 bytes. The variable i is shifted to align on integers. This calculation of the array element address contains a multiplication. The multiplication is carried out by the standard function "mul", which uses many clock cycles. When the function fill_ua_buffer receives an address of the first element of the one dimensional array, the multiplication, the load of image_unit_addresses and the load of current_image_unit_addresses, are not necessary. Only the offset calculation with i and the load of the element stay.

The third improvement can be done with the expression "4*i". If the multiplication is carried out by a shift left operation (which is possible because 4 is a power of 2) the result is the same but there is only one clock cycle needed.

The forth improvement is also with the "4*i" statement. This is already improved to "*(linebuffer + (i<<2)) = ...", but the calculation of the pointer address still needs 2 instructions (shift left and add). If the line buffer address is incremented by 4 in a separate statement every cycle of the loop and the "4*i" is removed, the offset calculation costs only one instruction. When these improvements are carried out, it results in a program

[1] Some instructions need more clock cycles per instruction (see appendix A.2).

which is given in figure 6.6. The assembly listing is given in appendix E.2.

```
#define LINE_LENGTH                 3000
#define BUFFER_SIZE                  4
#define LINE_BUFFER_BASE_ADDRESS    0x090002
extern int current_image_line;
extern int image_unit_addresses[BUFFER_SIZE][LINE_LENGTH];

void fill()
   {
   full_ua_buffer(&(image_unit_addresses[current_image_line][0]));
   }

void fill_ua_buffer ( image_l )
int image_l[LINE_LENGTH];
   {
   int i;
   short *linebuffer;

   linebuffer = LINE_BUFFER_BASE_ADDRESS;
   for( i = 0; i < LINE_LENGTH; i++ )
      {
      *(linebuffer) = image_l[i];
      linebuffer += 4;
      ]
   }
```

Figure 6.6 *Listing fill_ua_buffer improved*

The loop of the first implementation costs 20 instructions (24 clock cycles) and the last implementation needs 7 instructions (10 clock cycles). The time needed by the function __mul which is called twice is not taken into account. So using local variable instead of global variables can have great influence on the processing time.

Project status and conclusions

The hardware of the Area Segmentation Processor was built and tested. The hardware was changed on several places: a complete new design was made for the VME-interface, the communication with the terminal was improved, the signalling of the reconfigure bits of the image handler and the screen handler were improved, a write pulse for the circular buffer was made and the PB-bus interface was extended so that it has the same functions as the other cards.

The ASP was integrated in the in-line PCB inspection system. Herefor a program was designed, which receives and sends 512x512 images.

Parts of the segmentation software were built. This is not finished because there was not enough time. The improvement of the hardware cost more time then was expected. The combining algorithm of the segmentation software can best be run on the host because there is enough time and it has only be done only once. The address generation for the screen maps can best be done on the ASP because this gives the lowest VME-traffic. In the first implementation the address generation for the line buffers and the screen maps is combined into one function.

Use of global variables delays C programs, especially when used in a loop. Replacing them by local variables can decrease the execution time of a function.

Abbreviations

AOI	Area Of Interest
ASP	Area Segmentation Processor
CAD	Computer Aided Design
CFT	Centre For manufacturing Technology
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DUART	DUal Asynchronous Receiver Transmitter
EPLD	Erasable Programmable Logic Device
ERM	Embedded Real-time Monitor
ESPRIT	European Strategic Programme for Research and development in Information Technology
FMU	Fiducial Measurement Unit
IMI	Industrial Measurement and Inspection
ISP	Industrial Signal Processing
ITE	Information Engineering
LUT	Look Up Table
PCB	Printed Circuit Board
RISC	Reduced Instruction Set Computer
SBIP	Single Board Image Processor
SMD	Surface Mounted Device technology
SPARC	Scalable Processor ARChitecture
TRIOS	TRiangulation based Inspection Optical System
TUE	University of Technology Eindhoven
VME	Versa Module Eurocard

Literature

[BOO] Boots P.; 3D data processing for PCB inspection; TRIOS TB1; Eindhoven 13 march 1992

[CMO] CMOS - BiCMOS Data Book; Cypress Semiconductor; Sonetech Nederland bv, Gulberg, 5674 TE Nuenen; March 1, 1991

[ERM] Embedded Real_Time Monitor System PG4100; Philips Export BV, Eindhoven, the Netherlands; 1990;

[FAS] Fast TTL Logic Series; Philips Data Handbook, Integrated Circuits 15 (IC15); July 1990

[Lee] Leeuwen Frank van; Development of the CAD Data Correction & Expansion Unit for Esprit Project #2017; Philips CFT Technology; Eindhoven september 1990

[LSI] LSI Logic; Digital Signal Processing (DSP) Data book; June 1990

[MIC] Micro processors, micro controllers and peripherals; Philips Data Handbook, Integrated Circuits 14N (IC14N); september 1985

[SAC] SPARCV1EW and utilities; Assembly reference manual; C reference manual; Flame Computer Corporation, 5301 Commerce Avenue, suite 4, Moorpark Ca. USA; 1989 1990

[SME] Smeets F.G.M.; Hardware development of the Area Segmentation Processor for Esprit Project #2017: TRIOS; Philips CFT Technology; Eindhoven june 1992

[SPA] SPARC RISC User's Guide; Cypress Semiconductor Corporation, ROSS Technology, Inc.; Second Edition, February 1990

[VMP] VMEbus Prototyping Board PG2750/PG2751 Users manual; Philips Export B.V.; 1987

[VMS] VME bus specification manual, revision B; Philips Export bv; Eindhoven, the Netherlands; 1983

Appendix A The processor CY7C611

The processor used on the ASP is the SPARC RISC processor CY7C611 integer unit. The processor is a reduced version (fewer address lines and fewer control signals) of the CY7C601 integer unit. The processor can be used in combination with cache memory, cache memory controller/memory management unit and floating point unit. On the ASP only the reduced processor and cache memory is used.

The processor is designed according to the Scalable Processor ARChitecture (SPARC). SPARC is an architecturally driven standard with binary compatibility of software between processor versions.

The processor is a Reduced Instruction Set Computer (RISC). Most of the instructions are carried out with an execution rate of one clock cycle. The goal of a RISC is to obtain a maximal data processing performance through an intelligently selected set of instructions. A RISC differs from a CISC (Complex Instruction Set Computer). A short overview of the three mayor differences are given in table A.1.

Table A.1 *Differences between RISC and CISC*

	RISC	CISC
model	load/store	memory/register
source/destination instruction	non destructive triadic register file	accumulator/register file
instruction length	normalized	variable

A RISC has a LOAD/STORE model. The only two instructions that can access the main memory are LOAD and STORE. All the other instructions act upon internal registers and not upon the main memory. A CISC can carry out an operation on operands directly from the main memory. This means that for example an ADD instruction, using an operand from the main memory, is delayed until the content of the address is loaded into the processor. The time slot on the data bus between the LOAD an the ADD when the address is on the bus, cannot be filled.

A RISC uses a non destructive triadic register file. This means that three registers are used for an instruction and that the input registers remain unchanged after the result is written into the output register. On the other hand the CISC fundamental model uses a accumulator/register model where the accumulator is used as a input and output register

thereby destroying the original contents.

The third major difference is that a RISC has a normalized instruction length. Therefor no contextual fields in the instruction are necessary which greatly simplifies the addressing modes. The CISC instruction length depends on the addressing mode. An ADD instruction with the accumulator and a register will have a shorter instruction than an ADD instruction which uses the accumulator and an address in the main memory.

In the following paragraphs headlines of the CY7C611 are explained. These headlines are important to know when programming the processor (especially when the processor is programmed in assembly).

A.1 Registers

The processor has 136 working registers and six special purpose registers. In the next two paragraphs these registers are explained.

A.1.1 Working registers

A number of 8 registers out of the 136 working registers are so called global registers, which can always be accessed. The other 124 lie in a circular stack. The stack is divided into 8 windows. A window consists of 24 registers. The registers in a window are divided into three groups: 8 "in", 8 "local" and 8 "out" registers. All 24 registers can be read and written the same way. The only difference in the names is the fact that the "in"-registers of window i are the same registers as the "out"s of window i+1. The "out"s of window i are the same as the "in"s of window i-1. The local registers don't overlap. The principle of the circular stack is drawn in figure A.1(A).

At any given moment a program can address 32 registers: 8 global registers and 24 registers of the current window (pointed by the current window pointer: CWP). The current window pointer can be changed in three ways (see also figure A.1(B)):

SAVE instruction: The CWP is decremented by one
RESTORE instruction: The CWP is incremented by one
write to the window pointer field in the PSR register: Any change

The special way of the working registers organisation can be efficiently used in a program. When a program runs it can use the "in" and "local" registers for variables and temporary results. When that program calls a function it can put the outgoing parameters in the out registers. When the first action of the called function is a SAVE instruction the incoming parameters are stored into the in-registers. The locals are free to be used by the

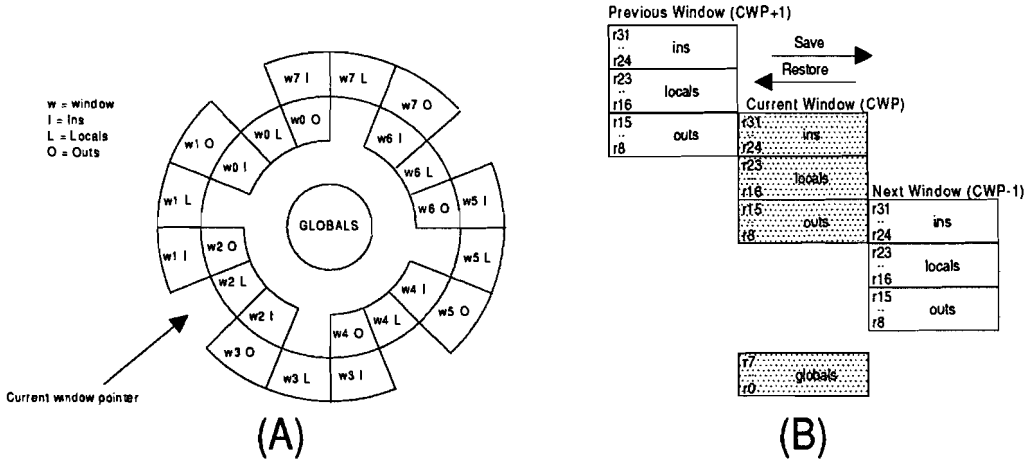


Figure A.1 *(A) Window stack (B) Window overlap*

called function. Passing parameters back can be done by putting the parameters in the in-registers followed by a RESTORE instruction.

A.1.2 Special purpose registers

The Processor Status Register (PSR), drawn in figure A.2, contains bits for enabling/disabling, flags, interrupt level, processor mode and the current window pointer (CWP).

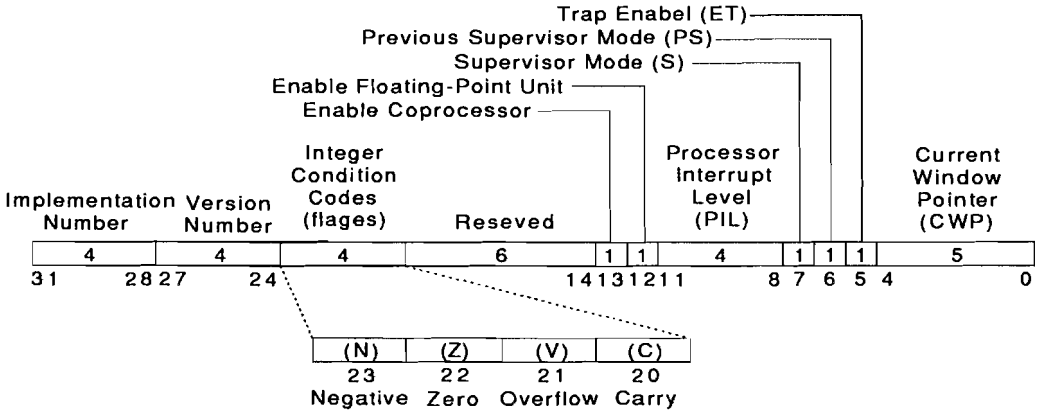


Figure A.2 *Processor State Register (PSR)*

The processor can operate in two different modes: supervisor mode and user mode. For example access to the special purpose registers is only allowed in supervisor mode. Two bits in the PSR, Supervisor mode bit (S) and Previous Supervisor mode bit (PS), indicate the mode. The Supervisor mode bit contains the current mode of the processor, the PS the mode before the last recent trap. A trap (hardware/software interrupt) always sets the processor in supervisor mode. The PS bit is set according to the previous mode. When

returning from a trap the mode can be restored.

As for the interrupt procedure, the level of the incoming interrupt must be higher than the level specified in the Processor Interrupt Level bits (when the interrupt is enabled). The Current Window pointer points to the current window in the register stack. The bits, except for the implementation number and version number, can be changed in the supervisor mode.

The content of the Window Invalid Mask register (figure A.3) determines which window(s) will cause generation of a trap when pointed to by the CWP as the result of a SAVE, RESTORE or RETT (RETurn from Trap) instruction. Each bit in WIM corresponds to a window. If this bit is 1 the window corresponding to that bit is marked as invalid and will cause a trap when that window is pointed to as a result of one of the three instructions. The bits corresponding to a window can be changed by an instruction in the supervisor mode.

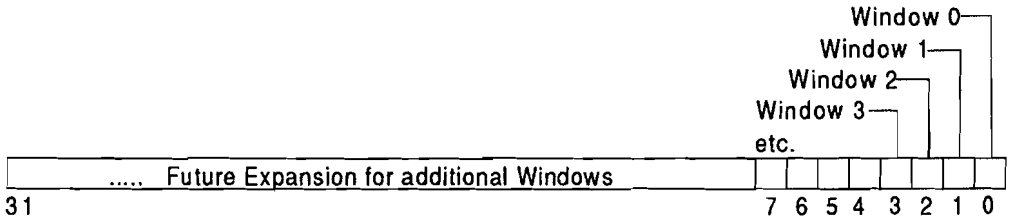


Figure A.3 *Window Invalid Mask (WIM)*

The Trap Base Register (figure A.4) consists of two parts. The Trap Base Address contains the trap table address. The trap table contains jump instructions to the corresponding trap routines. The Trap Type is the offset in the trap table and is filled by the hardware just before the trap is taken. The trap base address can be changed by an instruction in the supervisor mode.

Figure A.4 *Trap Base Register (TBR)*

The Y-register can be used in both modes. The Y-register is used by the multi step instruction to create 64 bits products. The bits in the Y-register have the same function as in a working register.

Two registers, which can't be reached directly, are the Program Counter (PC) an the next Program Counter (nPC). The PC contains the address currently being executed and the nPC holds the address of the next instruction to be executed. The nPC is necessary to

implement delayed control transfers. This will be further explained in A.3.

Some of the working registers have a special function which is used by the instructions. For example global register 0 is a read only register and contains value zero. The value zero is the mostly used constant in a program and easily available now. Data written to this register is lost.

When a CALL instruction is executed the address of the CALL instruction (the return address) is placed in "out" register 7 of the calling procedure window. By means of a SAVE instruction in the called function the return address is available in in-register 7. Local register 1 and 2 of the trap window are used to save the PC and the nPC at the time a trap is accepted. These two register can be used to return to the address before the trap was taken.

A.2 Pipeline

The SPARC RISC processor has a mean execution rate approaching one instruction per clock cycle. To achieve this the processor has a four stage instruction pipeline that permits parallel execution of multiple instructions. The processor pipeline can be seen in figure A.5.

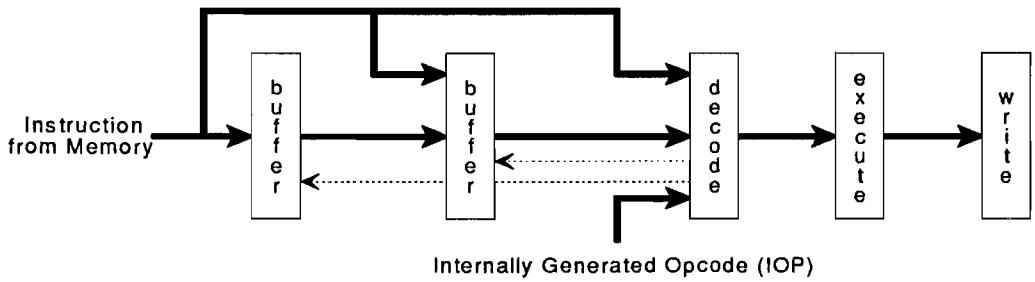


Figure A.5 *Processor instruction pipeline*

The instruction execution is broken into four pipeline stages:

1. Fetch: The processor outputs the instruction address to fetch the instruction.
2. Decode: The instruction is read and decoded. The processor reads the operands from the register file and computes the next instruction address.
3. Execute: The processor executes the instruction and saves the result in temporary registers. Pending traps are prioritized and taken during this stage.
4. Write: If no trap is encountered the processor writes the result into the destination register.

In figure A.6 an example is displayed of an instruction pipeline.

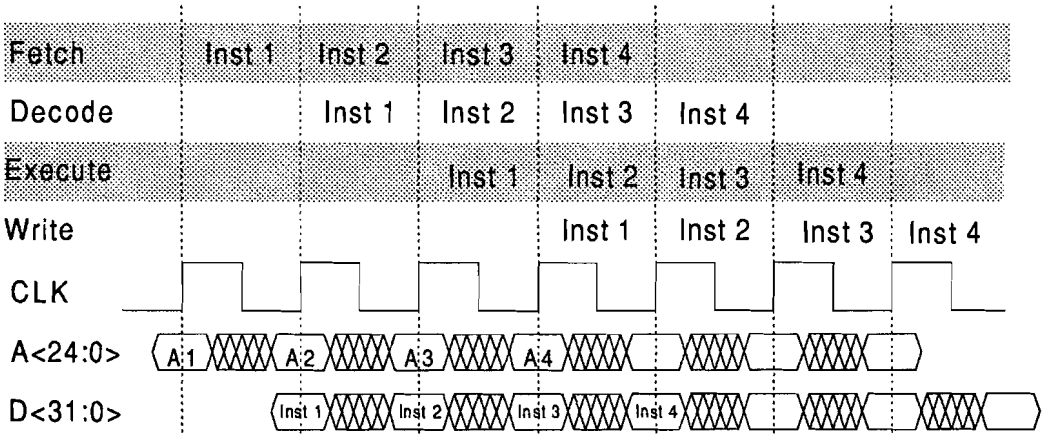


Figure A.6 *Pipeline with all single-cycle instructions*

The instructions in figure A.6 are called single-cycle instructions because they access the buses only once. Using this type of instruction yields an execution rate of one instruction per clock cycle. Instructions that require extra cycles automatically insert Inter cycle OPcodes (IOP) into the decode stage as they move into the execution stage. For example a LOAD instruction needs an IOP because the LOAD address must be placed on the address bus and the data must be transferred on the data bus. The instruction pipeline of the LOAD instruction in displayed in figure A.7. The multi cycle instructions which generate IOPs are listed in table A.2.

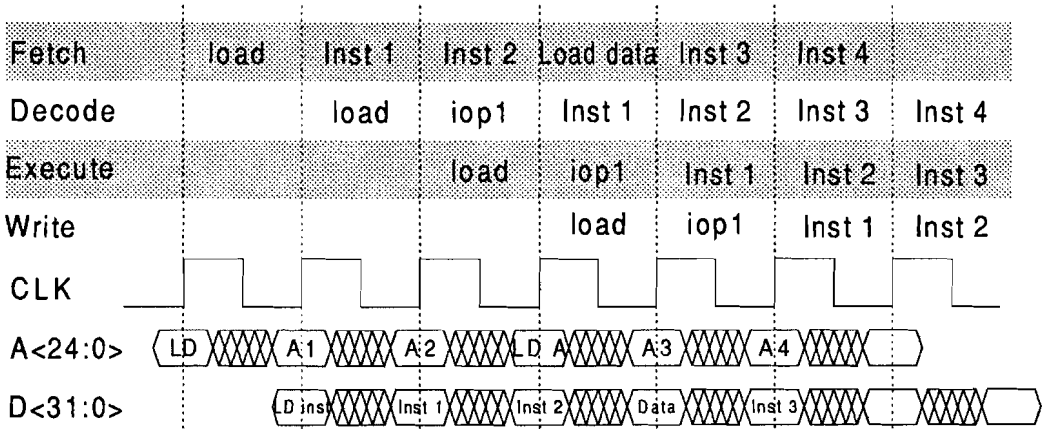


Figure A.7 *Pipeline with a LOAD instruction*

At the moment an IOP is inserted the next instruction is already in the fetch stage. To keep the fetched instruction the buffers before the decoder in figure A.5 are used to store this instruction.

Table A.2 *Internally generated opcodes*

Instruction	Number of Internal Opcodes
Single loads	1
Double loads	2
Single stores	2
Double stores	3
Atomic load-store	3
Jump	1
Return from trap	1

A.3 Delayed control transfer

Normally the instruction following a control transfer (jump or branch) is the target instruction (the instruction jumped to). In a pipeline model this means that the instruction following the control transfer, which is already fetched, has to be ignored. This is a waste of time. To avoid this the processor delays the execution of the target instruction until the instruction following the control transfer instruction is executed. The instruction in this delay slot is called a delay instruction. An example of the instructions flow is given in figure A.8. In this figure the contents of the PC and nPC are given. The nPC is necessary to contain the address of the next instruction. At the moment the branch instruction is decoded the delay instruction is fetched. The result of the decode is placed in the nPC at the moment the contents of the nPC is copied to the PC.

PC	nPC	Instruction
8	12	Non-control transfer
12	16	Control transfer (target = 40)
16	40	Non-control transfer (delay instruction) (transfer control to 40)
40	44	.... (target instruction)

Figure A.8 *Delay instruction*

A.4 Data types

The CY7C611 supports ten data types:

(unsigned) byte	8 bits,
(unsigned) half word	16 bits,
(unsigned) word	32 bits,
tagged data	30 bits + 2 tag bits,
double word	64 bits,
single precision floating point	32 bits and
double precision floating point	64 bits.

Tagged data is a special data type for languages as LISP, Smalltalk and Prolog.

When reading/writing data from/to the memory the address must be aligned with the data type. This means that half words can only be read/written from/to an address which is divisible by 2 (words only when the address is divisible by 4).

Because the processor works with a 32 bits data bus not all data bits will be used with a data type smaller then 32 bits. The bits valid for bytes and half words with a certain address are displayed in table A.3.

Table A.3 *Valid data bus for bytes and half words*

BYTES

address	data bus bits valid
N	31 .. 24
N + 1	23 .. 16
N + 2	15 .. 8
N + 3	7 .. 0

HALF WORDS

address	data bus bits valid
N	31 .. 16
N + 2	15 .. 0

N is divisible by 4.

A.5 Instructions

The processor has three basic instruction formats and three subformats. These formats are drawn in figure A.9.

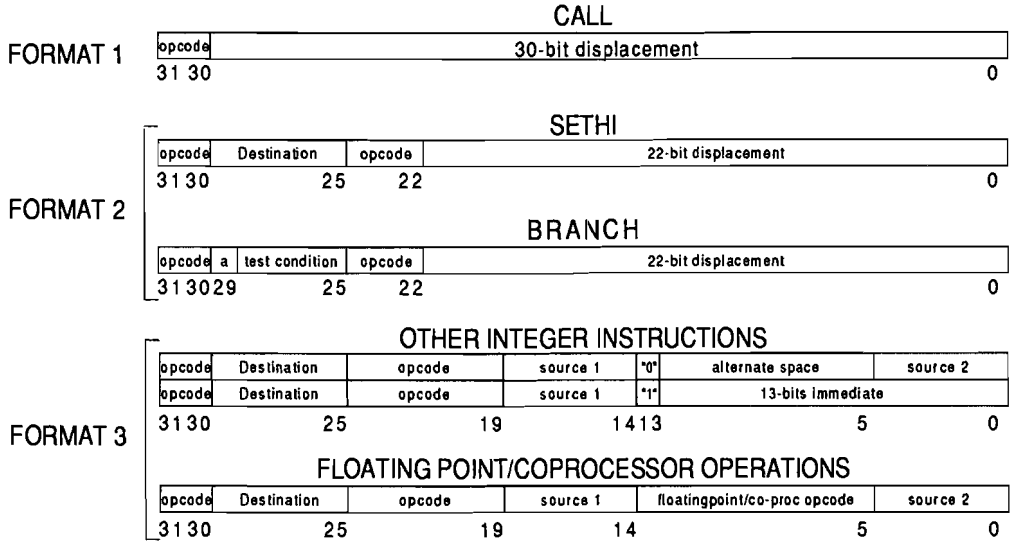


Figure A.9 *Instruction format*

The first format is for the CALL instruction (seen by the two most significant bits). The 30-bit displacement contains the offset from the current position (in the program counter) to the called function (in words). To reach the called function the 30 bit displacement is shifted two bits to the left to make a 32 bit address and thereby word aligned and then added to the program counter to form the actual call address.

In the SETHI format the 22 bit are placed immediate into the 22 most significant bits of the destination register.

The BRANCH format contains a test condition which indicates which flag(s) in the processor state register has(ve) to be tested. The 22 bits displacement is shifted two bits, subsequently sign extended for the direction and then added to the program counter as soon as the branch is made.

The format for the other integer instructions depends on bit 13. If this bit is zero source 1 and source 2 are the addresses for the input registers. The operation is specified in the right opcode (bit 24..19) and the result is placed in the register specified by the address in destination. If bit 13 is 'one' only one input register is used and the second input is a 13 bit constant.

The format for the floating point/coprocessor instructions always uses two input registers and a destination register.

A.6 Traps

The traps on the CY7C611 can occur as synchronous and asynchronous traps (also called external interrupts). The synchronous traps are divided into hardware traps and software traps. The hardware traps occur when an instruction during the execution goes wrong or is not allowed. The software traps are caused by certain instructions in the program.

When a trap occurs the following actions take place:

- further traps are disabled
- the S bit of the PSR is copied to the PS bit
- the CWP is decremented
- the PC and nPC are saved into local register 1 and 2 of the trap window
- the trap type is stored in the TBR register
- if the trap is not a reset the TBR is copied into the PC and TBR + 4 into nPC
  else the PC is filled with zero and nPC with 4

In the trap table four 32-bit words are reserved for every trap. The instructions in the trap table are a jump or branch to the trap routine.

To return normally, the routine must end with the instructions JMPL (JuMP and Link) and RETT (RETurn from Trap). The actions of JMPL are:

- the PC is copied into the destination register
- the nPC is copied into the PC
- the sum of the contents of the two source registers (take here local register 1 and global register 0) is placed into nPC

The actions done by RETT are:

- the CWP is incremented and temporary saved
- the traps are enabled
- the nPC is copied into the PC
- the sum of the contents of the two source registers (take here local register 2 and global 0) is placed into the nPC
- the CWP is filled with the temporary result
- the PS bit is copied into the S bit

The synchronous hardware traps and the interrupt have a priority level. The levels are given in table A.4. The interrupt priority level in the processor state register only specifies the minimum level of the interrupts. When a trap occurs and the traps are disabled, the processor enters the ERROR mode and stops executing. Interrupts are ignored when traps are disabled.

Table A.4 *Trap levels*

Trap	Priority	Trap type	Synchronous or asynchronous
reset	1 (highest)	-	Async.
instruction access	2	1	Sync.
illegal instruction	3	2	Sync.
privileged instruction	4	3	Sync.
floating-point disabled	5	4	Sync.
coprocessor disabled	6	36	Sync.
window overflow	7	5	Sync.
window underflow	8	6	Sync.
memory address not aligned	9	7	Sync.
floating-point exception	10	8	Sync.
coprocessor exception	11	40	Sync.
data access exception	12	9	Sync.
tag overflow	13	10	Sync.
trap instructions	14	128..255	Sync.
interrupt level 15 .. 1	15..29	31..17	Async.

Appendix B Schemes Area Segmentation Processor

B.1 Controller

B.2 EPLD interrupt handler

B.3 Image handler

B.4 Screen handler

B.5 EPLD screen handler

B.6 EPLD divider

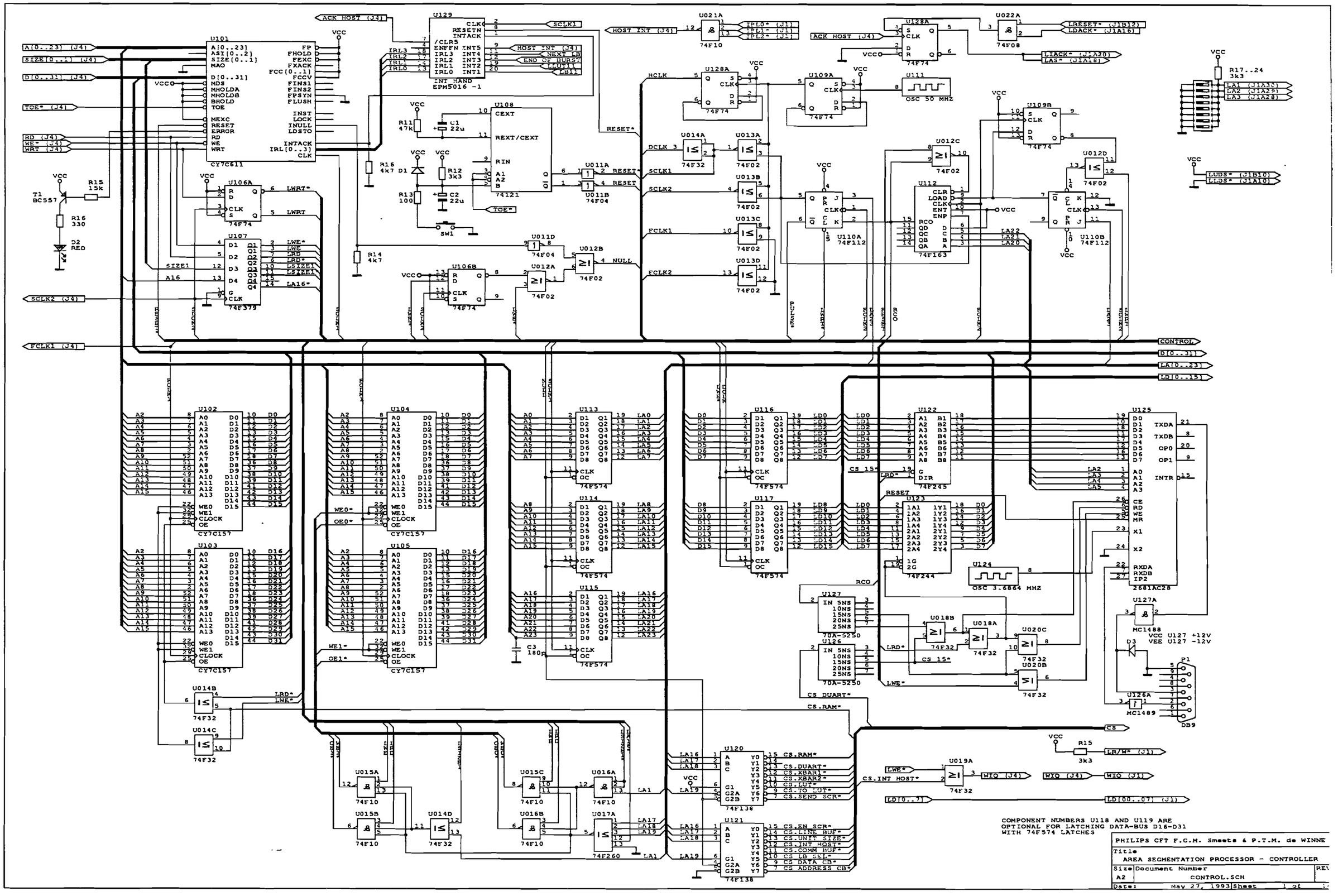
B.7 Circular buffer

B.8 Picture bus interface

B.9 VME-interface

B.10 VME-interface handler

B.11 Communication buffer



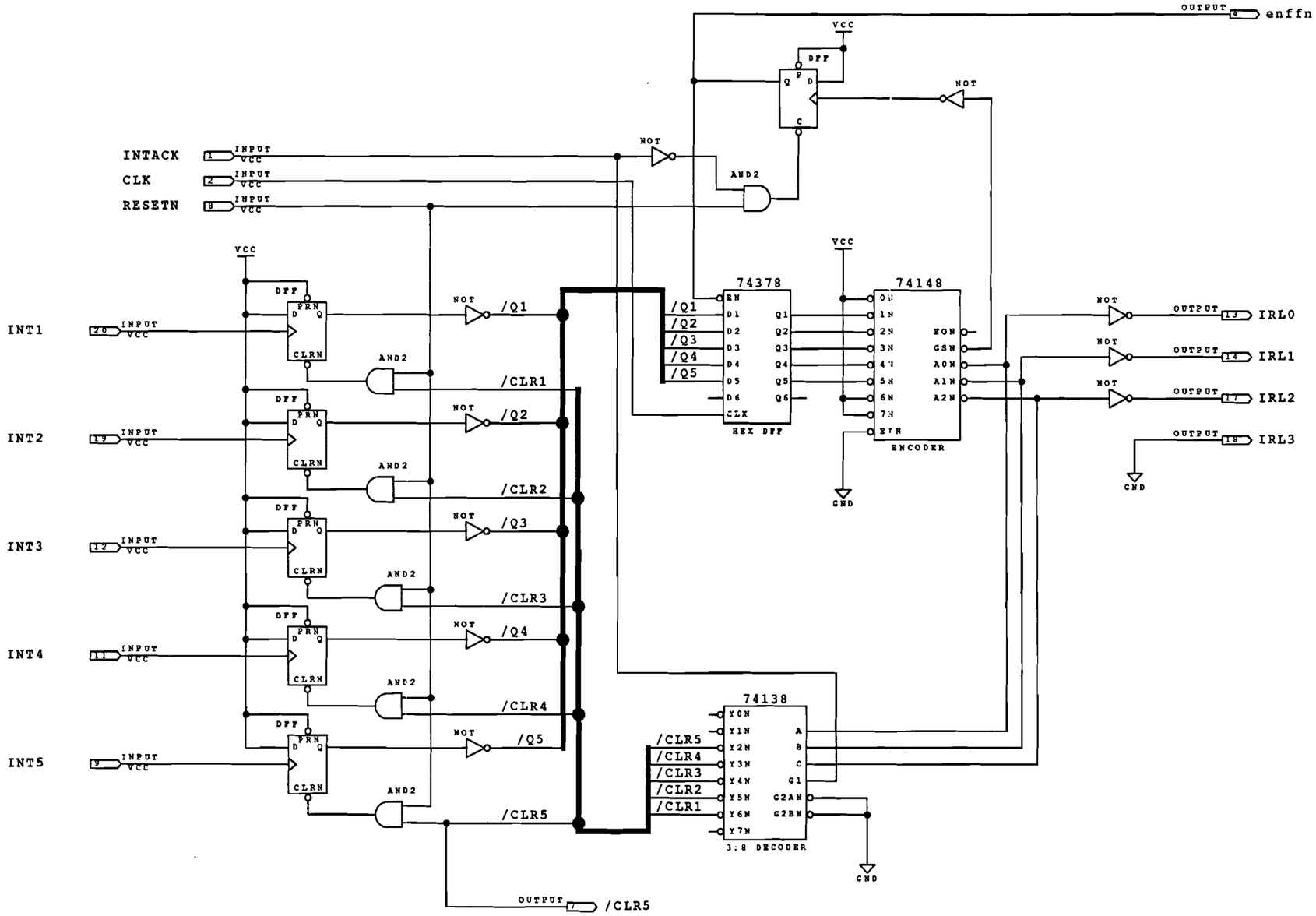
COMPONENT NUMBERS U118 AND U119 ARE
OPTIONAL FOR LATCHING DATA-BUS D16-D31
WITH 74F574 LATCHES

PHILIPS CFT F.G.M. Smeets & P.T.M. de WINNE

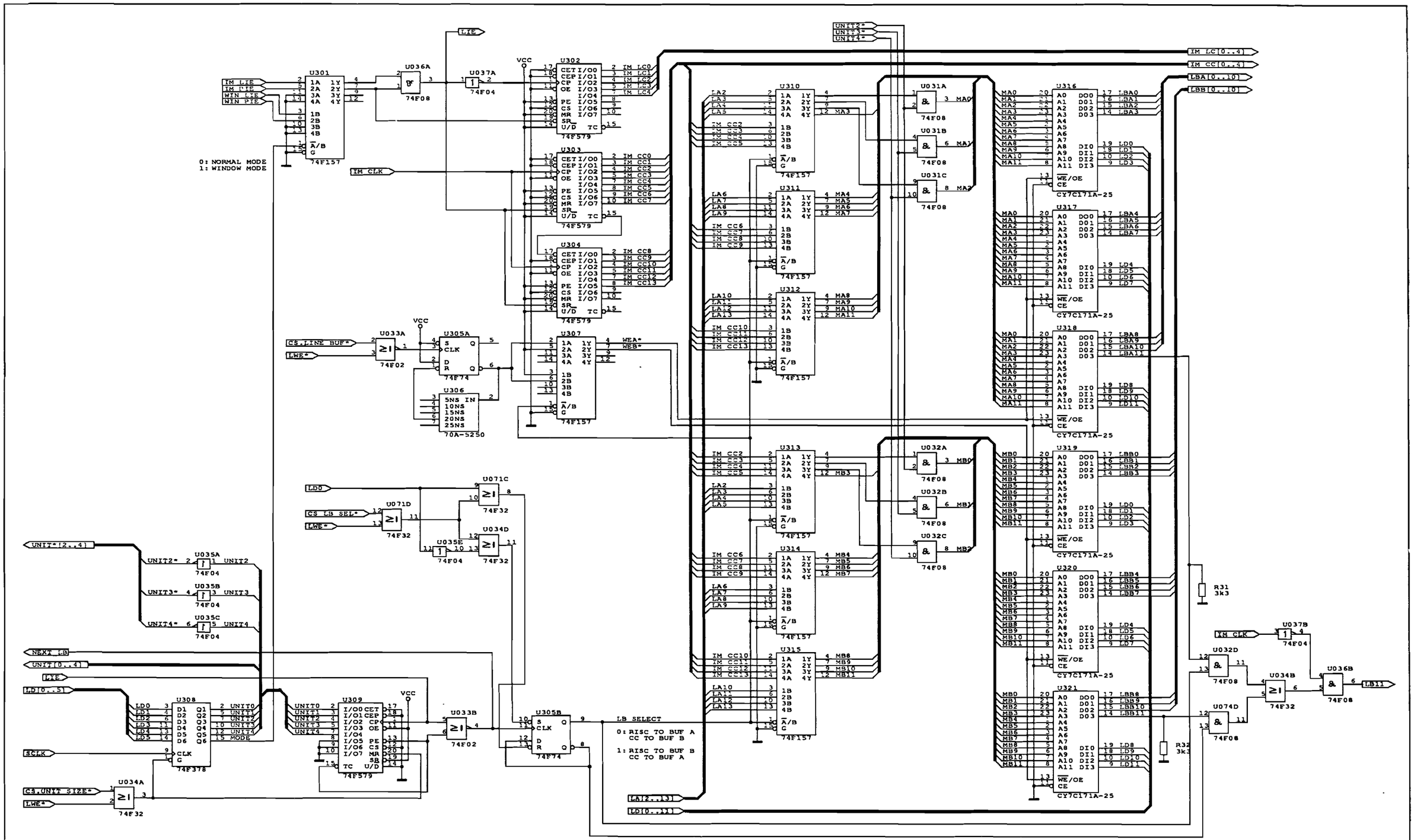
Title
AREA SEGMENTATION PROCESSOR - CONTROLLER

Size	Document Number	REV
A2	CONTROL.SCH	

Date: May 27, 1993 Sheet 1 of 1



TITLE	INTERRUPT HANDLER		
COMPANY	PHILIPS - CFT		
DESIGNER	P.T.M. de WINNE		
SIZE D	EPLD EPM5016-1	NUMBER 1.00	REV A
DATE 11:00a 1-11-1993		SHEET 2 OF 10	
TURBO ON		SECURITY OFF	



IM_LC : image line counter
IM_CC : image column counter

LA : latched address (processor)
LD : latched data (processor)

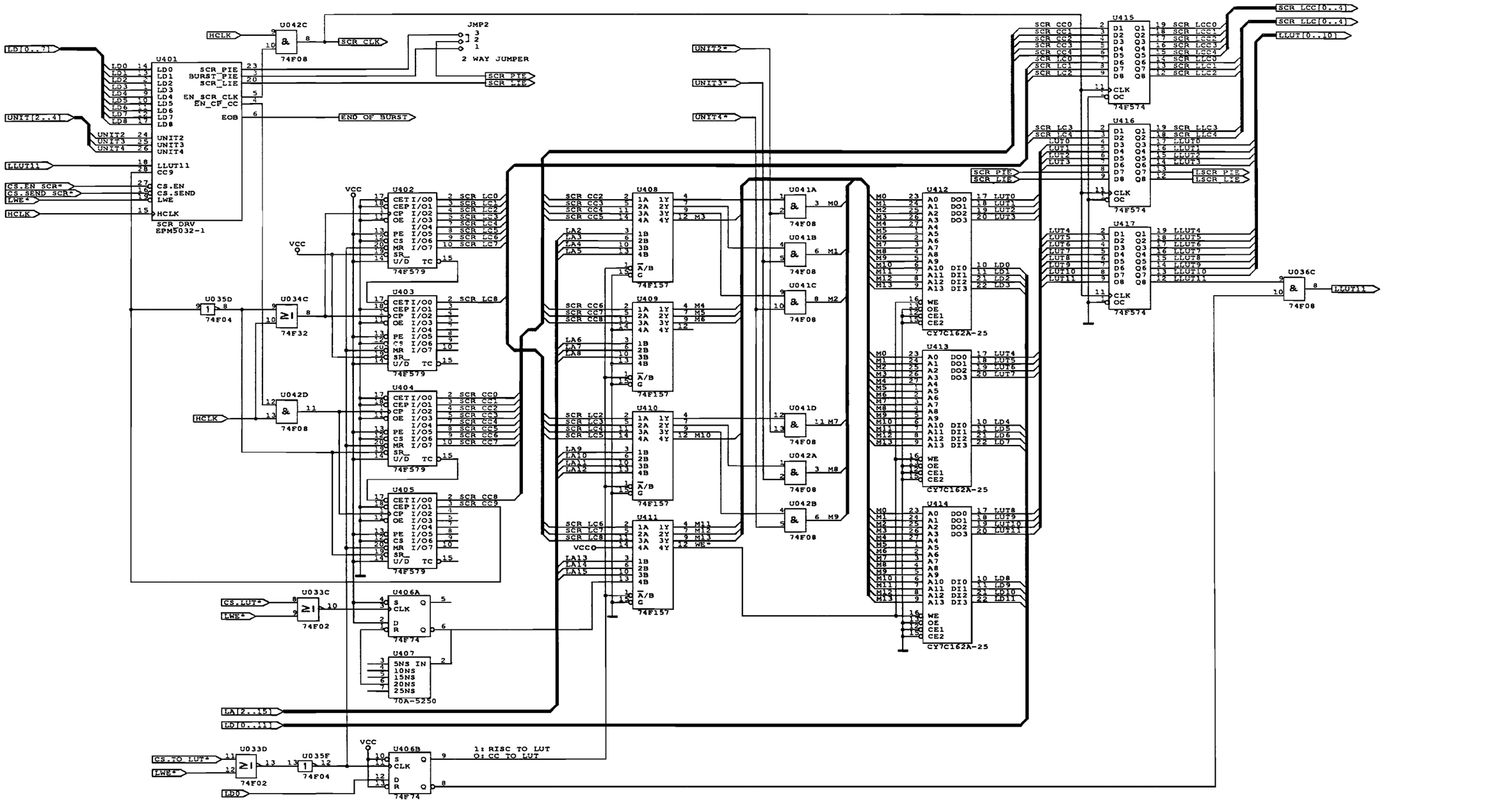
MA : multiplexer A
MB : multiplexer B

LBA : line buffer A
LBB : line buffer B

PHILIPS CFT F.G.M. Smeets & P.T.M. de WINNE

Title
AREA SEGMENTATION PROCESSOR - IMAGE HANDLER

Size	Document Number	REV
A2	IMAGE.SCH	
Date: May 27, 1993	Sheet 3 of 14	



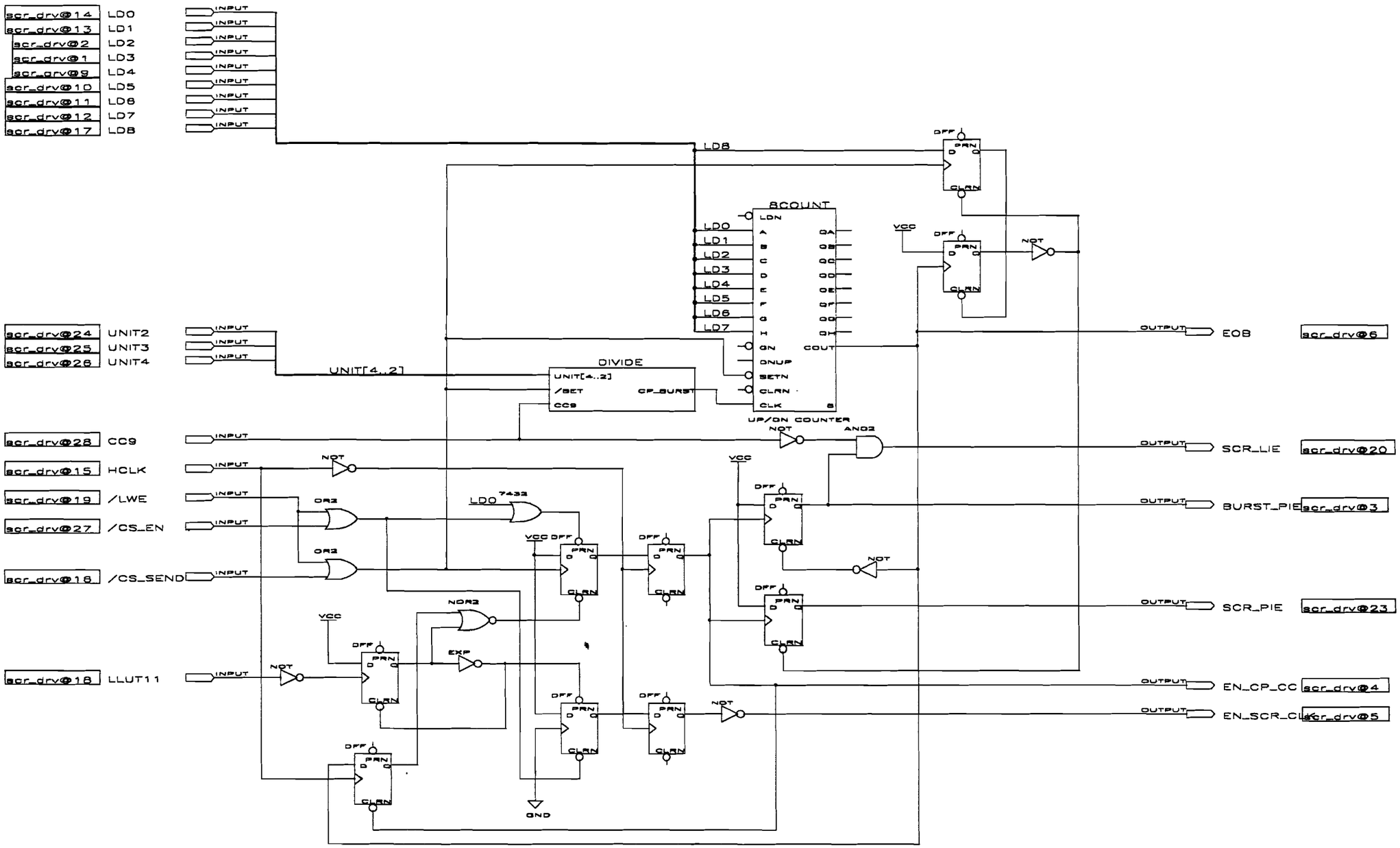
SCR_LC : screen line counter
SCR_CC : screen column counter

LA : latched address (processor)
LD : latched data (processor)

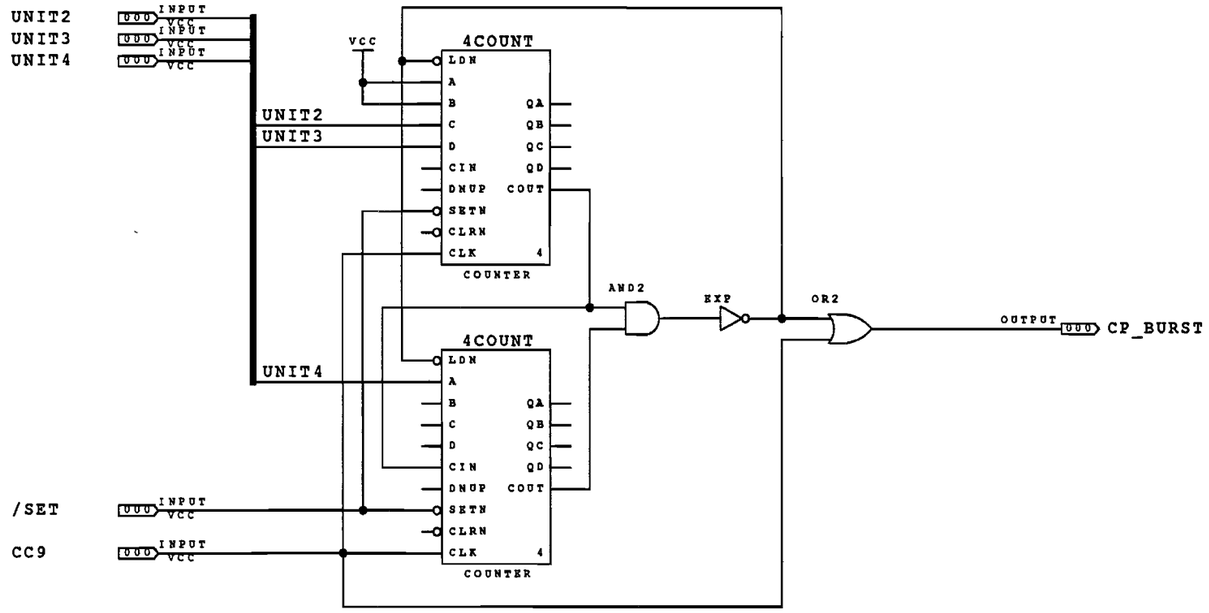
M : multiplexer

LUT : look up table

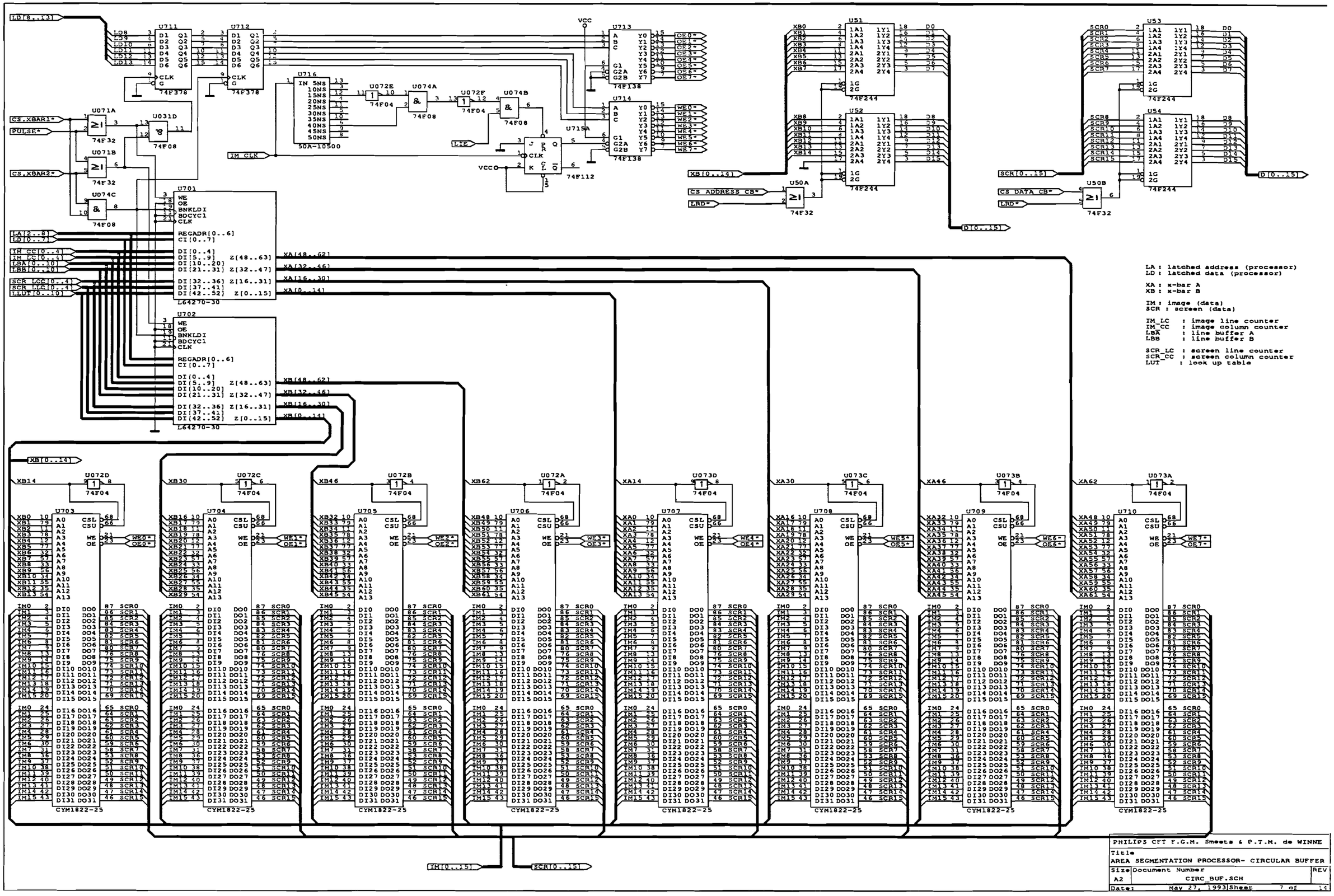
PHILIPS CFT F.G.M. Smeets & P.T.M. de WINNE		
Title: AREA SEGMENTATION PROCESSOR - SCREEN HANDLER		
Size: A2	Document Number: SCREEN.SCH	REV:
Date: May 27, 1993	Sheet: 4 of 14	

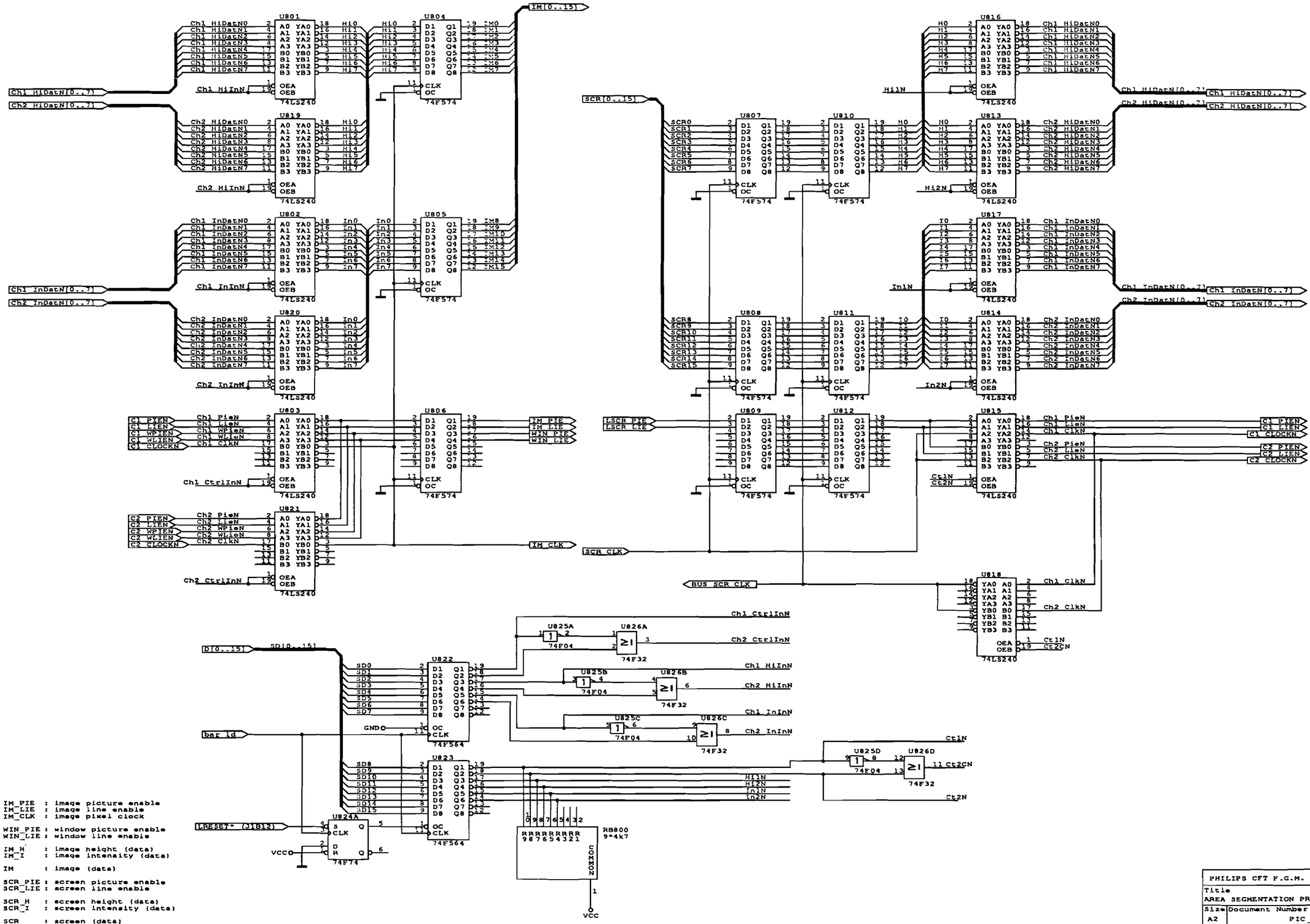


TITLE	SCREEN DRIVER		
COMPANY	PHILIPS – CFT		
DESIGNER	F.G.M. Smeets		
SIZE D	NUMBER 1.00	REV A	
DATE 2:58p 4–16–1993	SHEET 5	OF 10	



TITLE	DIVIDER		
COMPANY	PHILIPS - CFT		
DESIGNER	F.G.M. Smeets		
SIZE D	EPLD EPM5032-1	NUMBER 1.00	REV A
DATE 2:11p 10-26-1992		SHEET 6	OF 10
TURBO ON		SECURITY OFF	


LA : latched address (processor)
LD : latched data (processor)
XA : x-bar A
XB : x-bar B
IM : image (data)
SCR : screen (data)
IM_LC : image line counter
IM_CC : image column counter
LBA : line buffer A
LBB : line buffer B
SCR_LC : screen line counter
SCR_CC : screen column counter
LUT : look up table
PHILIPS CFT F.G.M. Smeets & P.T.M. de WINNE
Title
AREA SEGMENTATION PROCESSOR- CIRCULAR BUFFER
Size A2
Document Number CIRC_BUF.SCH
REV
Date: May 27, 1993 Sheet 7 of 14



IM_PIE : image picture enable
IM_LIE : image line enable
IM_CLK : image pixel clock

WIN_PIE : window picture enable
WIN_LIE : window line enable

IM_H : image height (data)
IM_I : image intensity (data)

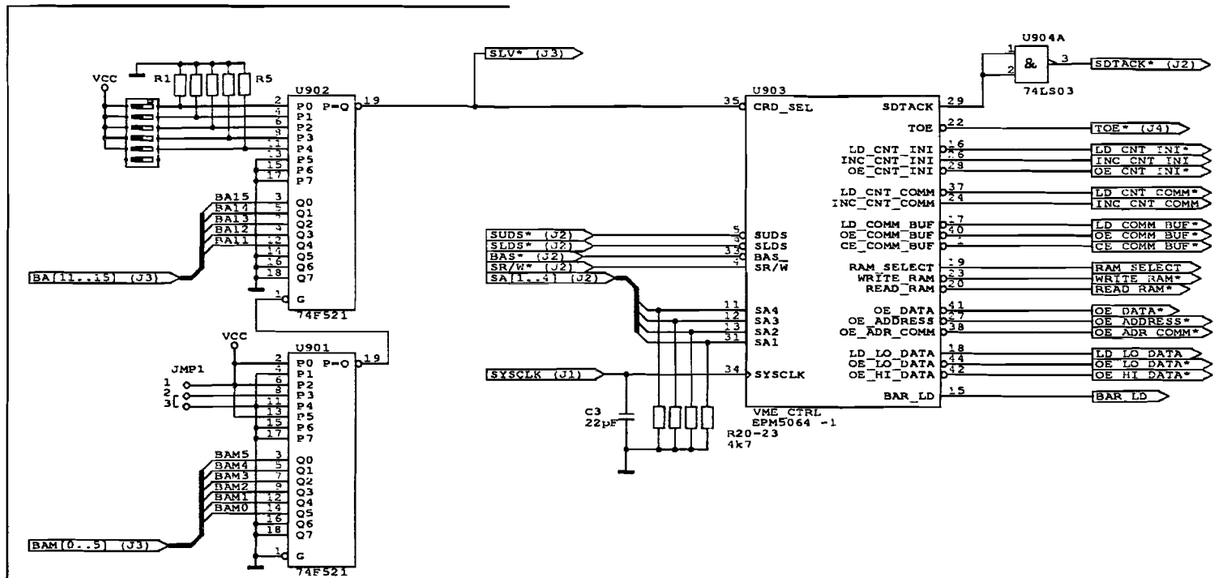
IM : image (data)

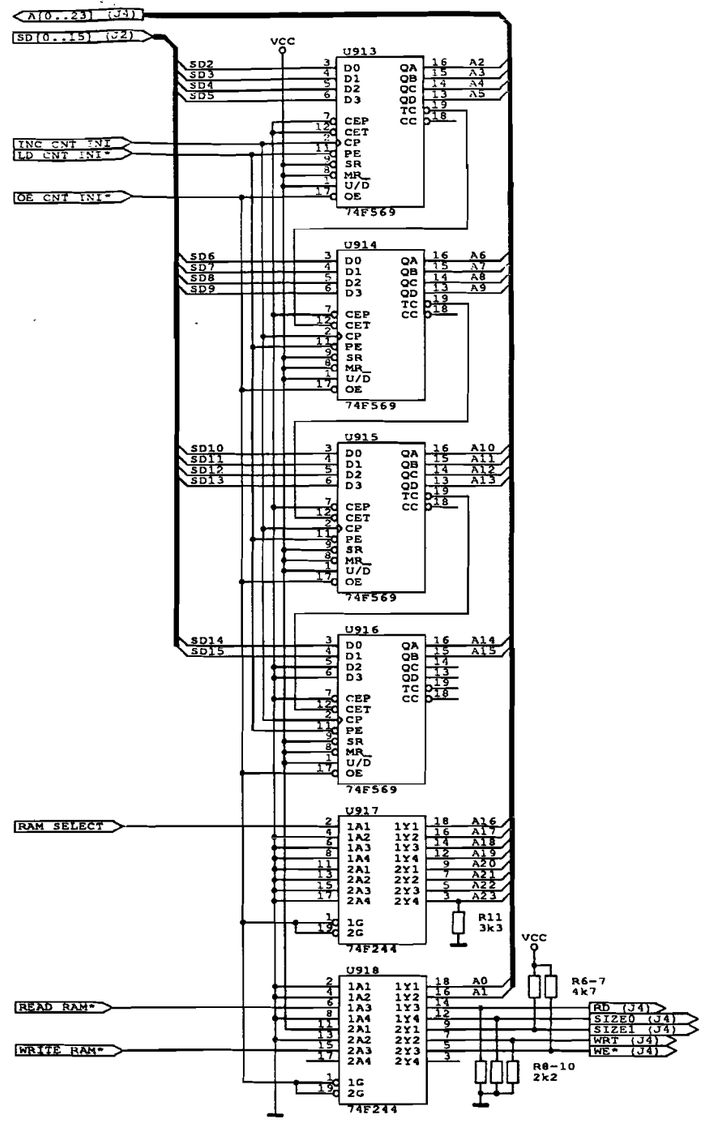
SCR_PIE : screen picture enable
SCR_LIE : screen line enable

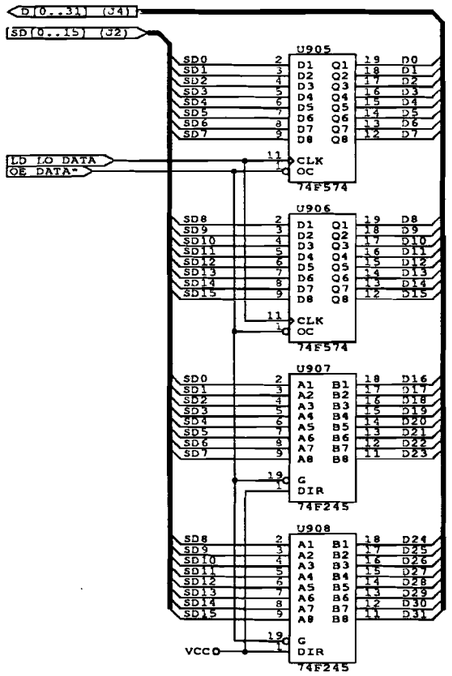
SCR_H : screen height (data)
SCR_I : screen intensity (data)

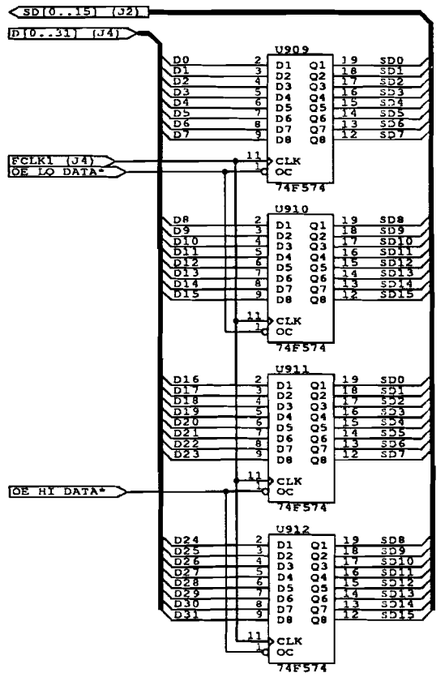
SCR : screen (data)

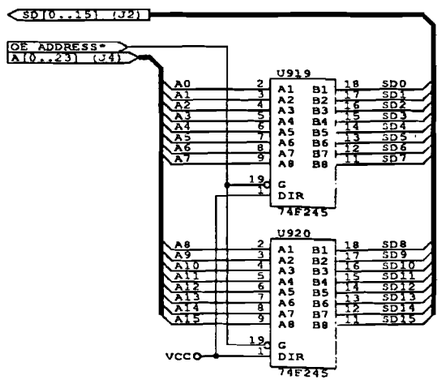
PHILIPS CFT F.G.M. SMEETS & P.T.M de WINNE		
Title: AREA SEGMENTATION PROCESSOR - PI-BUS INTERF.		
Size: A2	Document Number: PIC_INT.SCH	REV:
Date: February 19, 1993	Sheet: 8 of 14	



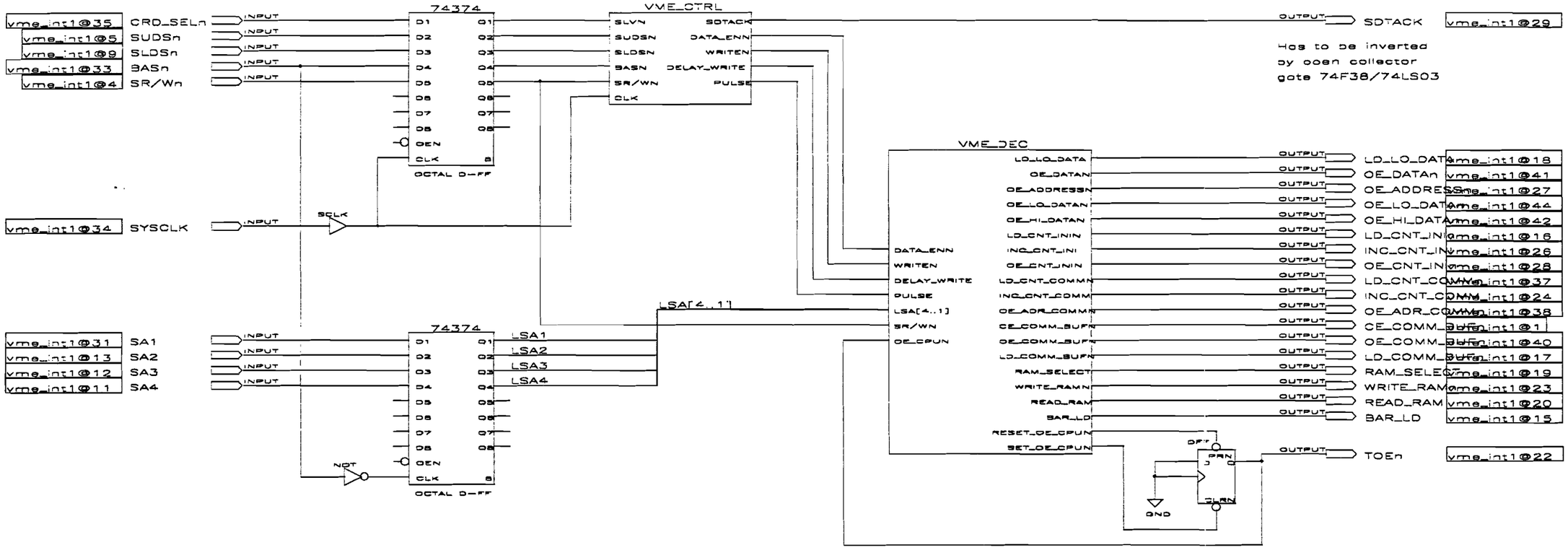




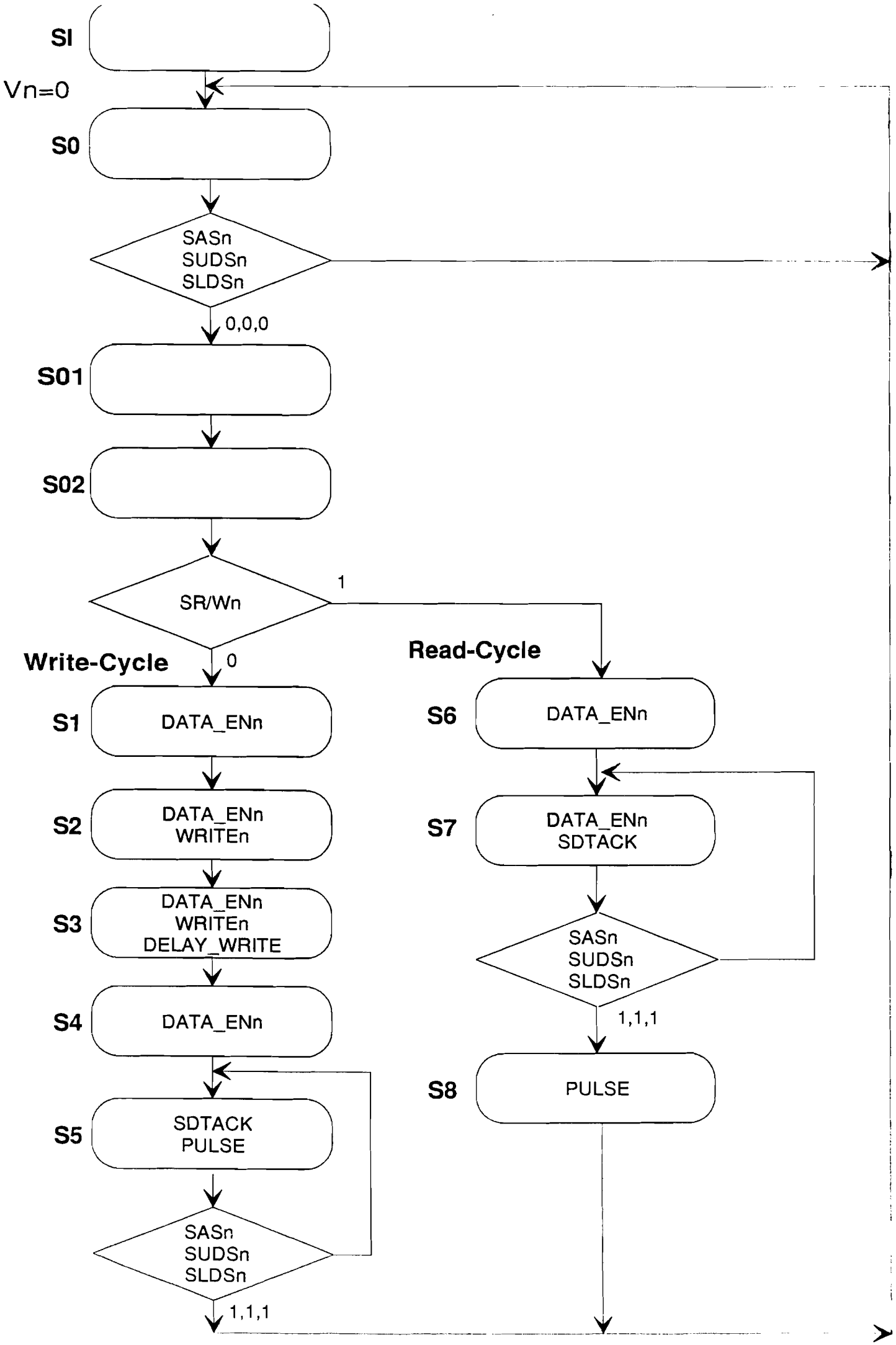




PHILIPS CFT F.G.M. Smeets & P.T.M. de WINNE		
Title		
AREA SEGMENTATION PROCESSOR - VME-BUS INTERF		
Size A2	Document Number VME_INT.SCH	REV
Date: April 27, 1993	Sheet 9 of 14	



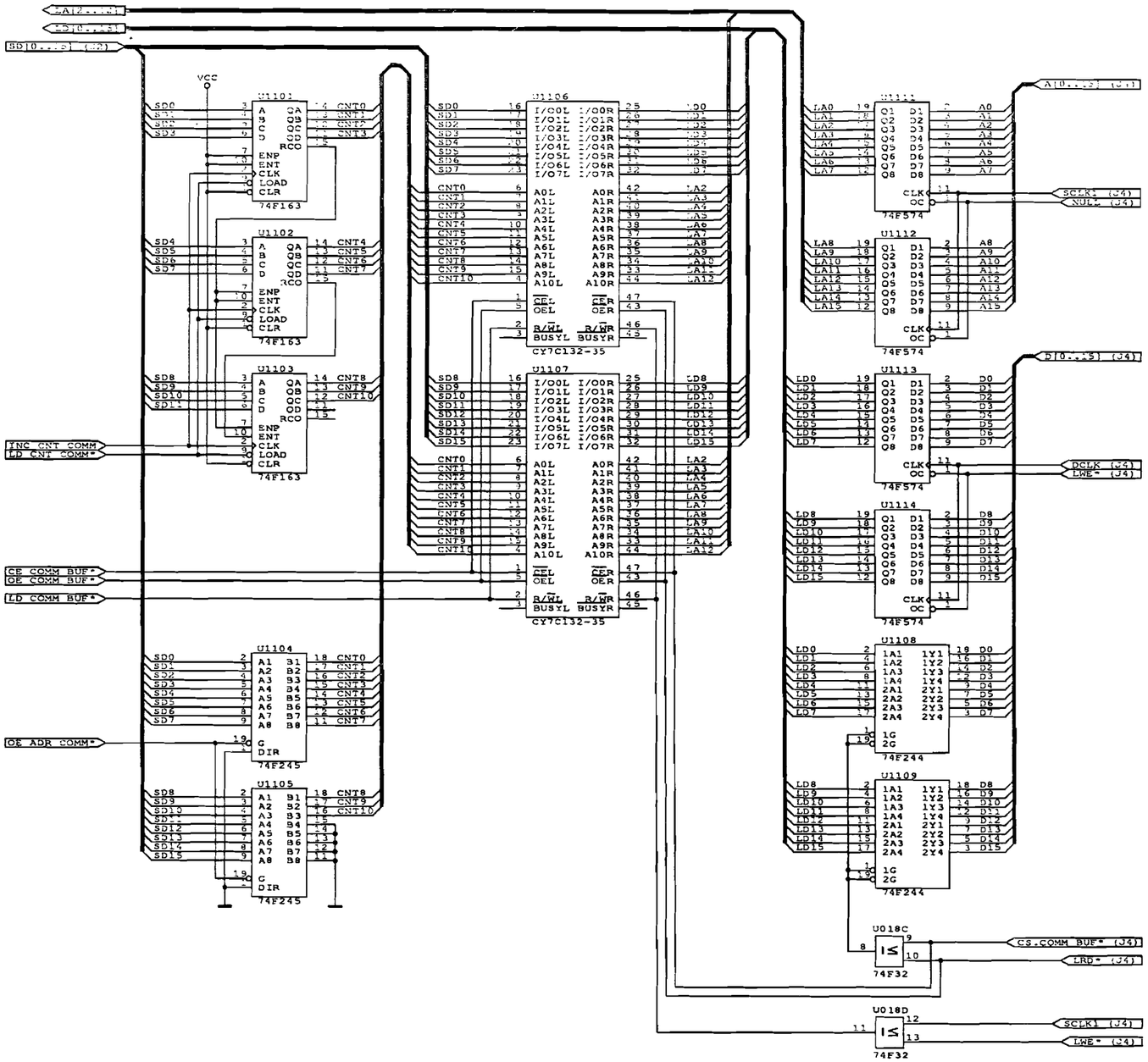
TITLE	VME-INTERFACE
COMPANY	PHILIPS - CFT
DESIGNER	F.G.V. Smeets
SIZE D	NUMBER 1.00 REV A
DATE 8:00o 1-21-1993	SHEET 1 OF 1


SI
Vn=0
S0
SASn
SUDSn
SLDSn
0,0,0
S01
S02
SR/Wn
1
0
Write-Cycle
Read-Cycle
S1
DATA_ENn
S2
DATA_ENn
WRITEn
S3
DATA_ENn
WRITEn
DELAY_WRITE
S4
DATA_ENn
S5
SDTACK
PULSE
SASn
SUDSn
SLDSn
1,1,1
S6
DATA_ENn
S7
DATA_ENn
SDTACK
SASn
SUDSn
SLDSn
1,1,1
S8
PULSE

```
%************************************************************
*        DESIGN OF CONTROLLER TO INTERFACE TO THE VME       *
*                                                           *
*        Version : 1.1                                      *
*        Date    : 09/04/93                                 *
*        By      : P.T.M. DE WINNE / F.G.M. SMEETS          *
************************************************************%

TITLE "vme_ctrl";
SUBDESIGN vme_ctrl
(
  SLVn         : INPUT;
  SUDSn        : INPUT;
  SLDSn        : INPUT;
  BASn         : INPUT;
  SR/Wn        : INPUT;
  CLK          : INPUT;
  SDTACK       : OUTPUT;
  DATA_ENn     : OUTPUT;
  WRITEn       : OUTPUT;
  DELAY_WRITE  : OUTPUT;
  PULSE        : OUTPUT;
)

VARIABLE
  ss : MACHINE WITH STATES
(si,s0,s01,s02,s1,s2,s3,s4,s5,s6,s61,s62,s7,s8);
  DEN: NODE;
  WRN: NODE;
  DWR: NODE;
  PUL: NODE;
  ACK: NODE;

BEGIN
  DEFAULTS
    DEN = VCC;
    WRN = VCC;
    DWR = GND;
    PUL = GND;
    ACK = GND;
  END DEFAULTS;

  ss.clk       = clk;
  DATA_ENn     = DFF(DEN,clk,VCC,VCC);
  WRITEn       = DFF(WRN,clk,VCC,VCC);
  DELAY_WRITE  = DFF(DWR,clk,VCC,VCC);
  PULSE        = DFF(PUL,clk,VCC,VCC);
  SDTACK       = DFF(ACK,clk,VCC,VCC);

  TABLE
     %present present                          next  next                %
     % state  input                            state outputs             %
        ss,SLVn,SUDSn,SLDSn,BASn,SR/Wn => ss,DEN,WRN,DWR,PUL,ACK;
     %-------------------------------------------------------------------%
%POWER UP%
        si,   x,    x,    x,    x,    x => s0,  1,  1,  0,  0,  0;
%WAIT STATE FOR NEXT CYCLE%
        s0,   1,    x,    x,    x,    x => s0,  1,  1,  0,  0,  0;
        s0,   0,    1,    x,    x,    x => s0,  1,  1,  0,  0,  0;
        s0,   0,    x,    1,    x,    x => s0,  1,  1,  0,  0,  0;
        s0,   0,    x,    x,    1,    x => s0,  1,  1,  0,  0,  0;
        s0,   0,    0,    0,    0,    x => s01, 1,  1,  0,  0,  0;
% WAIT FOR VALID SR/W-SIGNAL%
        s01,  x,    x,    x,    x,    x => s02, 1,  1,  0,  0,  0;
        s02,  0,    0,    0,    0,    0 => s1,  0,  1,  0,  0,  0;
        s02,  0,    0,    0,    0,    1 => s6,  0,  1,  0,  0,  0;

%WRITE-CYCLE%
        s1,   x,    x,    x,    x,    x => s2,  0,  0,  0,  0,  0;
        s2,   x,    x,    x,    x,    x => s3,  0,  0,  1,  0,  0;
        s3,   x,    x,    x,    x,    x => s4,  0,  1,  0,  0,  0;
        s4,   x,    x,    x,    x,    x => s5,  1,  1,  0,  1,  1;
        s5,   x,    0,    x,    x,    x => s5,  1,  1,  0,  1,  1;
        s5,   x,    x,    0,    x,    x => s5,  1,  1,  0,  1,  1;
        s5,   x,    x,    x,    0,    x => s5,  1,  1,  0,  1,  1;
        s5,   x,    1,    1,    1,    x => s0,  1,  1,  0,  0,  0;

%READ-CYCLE%
        s6,   x,    x,    x,    x,    x => s61, 0,  1,  0,  0,  0;
        s61,  x,    x,    x,    x,    x => s62, 0,  1,  0,  0,  0;
        s62,  x,    x,    x,    x,    x => s7,  0,  1,  0,  0,  1;
        s7,   x,    0,    x,    x,    x => s7,  0,  1,  0,  0,  1;
        s7,   x,    x,    0,    x,    x => s7,  0,  1,  0,  0,  1;
        s7,   x,    x,    x,    0,    x => s7,  0,  1,  0,  0,  1;
        s7,   x,    1,    1,    1,    x => s8,  1,  1,  0,  1,  0;
        s8,   x,    x,    x,    x,    x => s0,  1,  1,  0,  0,  0;

  END TABLE;
END;
```

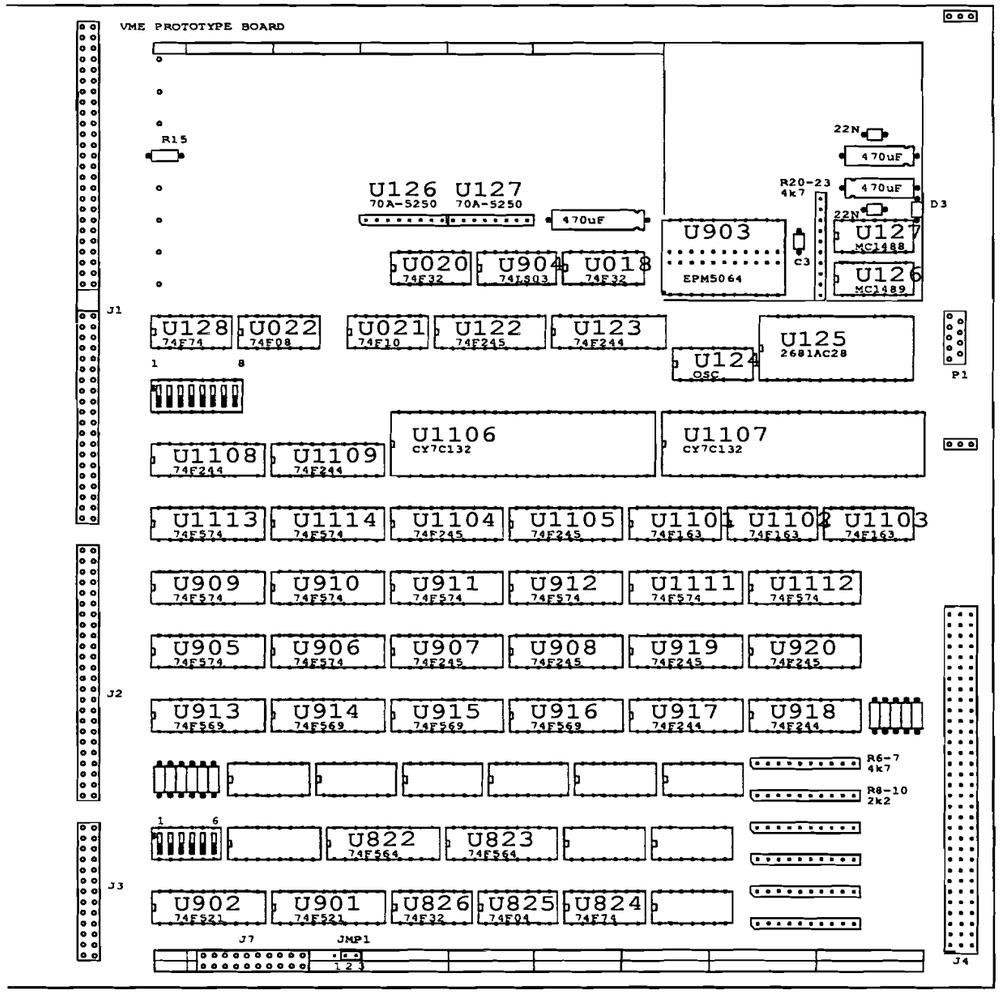
```
%************************************************************
*                                                           *
*        DECODER FOR VME-SIGNALS                            *
*                                                           *
*        Version : 1.1                                      *
*        Date    : 21/01/1993                               *
*        By      : P.T.M. DE WINNE / F.G.M. SMEETS          *
*                                                           *
************************************************************%

TITLE "signal decoder";

CONSTANT DIS_CPU  = B"0001";
CONSTANT EN_CPU   = B"0010";
CONSTANT CNT_INI  = B"0011";
CONSTANT CNT_COMM = B"0100";
CONSTANT COMM_BUF = B"0101";
CONSTANT LO_PRG   = B"0110";
CONSTANT LO_CAD   = B"0111";
CONSTANT HI_PRG   = B"1000";
CONSTANT HI_CAD   = B"1001";
CONSTANT BAR      = B"1010";

DESIGN IS "vme_dec";

SUBDESIGN vme_dec
(
  DATA_ENn         : INPUT;
  WRITEn           : INPUT;
  DELAY_WRITE      : INPUT;
  PULSE            : INPUT;
  LSA[4..1]        : INPUT;
  SR/Wn            : INPUT;
  OE_CPUn          : INPUT;
  LD_LO_DATA       : OUTPUT;
  OE_DATAn         : OUTPUT;
  OE_ADDRESSn      : OUTPUT;
  OE_LO_DATAn      : OUTPUT;
  OE_HI_DATAn      : OUTPUT;
  LD_CNT_ININ      : OUTPUT;
  INC_CNT_INI      : OUTPUT;
  OE_CNT_ININ      : OUTPUT;
  LD_CNT_COMMn     : OUTPUT;
  INC_CNT_COMM     : OUTPUT;
  OE_ADR_COMMn     : OUTPUT;
  CE_COMM_BUFn     : OUTPUT;
  OE_COMM_BUFn     : OUTPUT;
  LD_COMM_BUFn     : OUTPUT;
  RAM_SELECT       : OUTPUT;
  WRITE_RAMn       : OUTPUT;
  READ_RAM         : OUTPUT;
  BAR_LD           : OUTPUT;
  RESET_OE_CPUn    : OUTPUT;
  SET_OE_CPUn      : OUTPUT;
)

BEGIN

  DEFAULTS
    RESET_OE_CPUn = VCC;
    SET_OE_CPUn   = VCC;
    LD_LO_DATA    = GND;
    OE_DATAn      = VCC;
    OE_ADDRESSn   = VCC;
    OE_LO_DATAn   = VCC;
    OE_HI_DATAn   = VCC;
    LD_CNT_ININ   = VCC;
    INC_CNT_INI   = GND;
    OE_CNT_ININ   = VCC;
    LD_CNT_COMMn  = VCC;
    INC_CNT_COMM  = GND;
    OE_ADR_COMMn  = VCC;
    CE_COMM_BUFn  = VCC;
    OE_COMM_BUFn  = VCC;
    LD_COMM_BUFn  = VCC;
    RAM_SELECT    = GND;
    WRITE_RAMn    = VCC;
    READ_RAM      = GND;
    BAR_LD        = GND;
  END DEFAULTS;

  IF ( LSA[4..1]==DIS_CPU ) THEN
    RESET_OE_CPUn = WRITEn;
  END IF;

  IF ( LSA[4..1]==EN_CPU ) THEN
    SET_OE_CPUn = WRITEn;
  END IF;

  IF ( LSA[4..1]==CNT_INI AND SR/Wn==0 ) THEN
    LD_CNT_ININ = WRITEn;
    INC_CNT_INI = DELAY_WRITE;
  END IF;
```

```
IF ( LSA[4..1]==CNT_INI AND SR/Wn==1 AND OE_CPUn==1 ) THEN
  OE_CNT_ININ = DATA_ENn;
  OE_ADDRESSn = DATA_ENn;
END IF;

IF ( LSA[4..1]==CNT_COMM AND SR/Wn==0 ) THEN
  LD_CNT_COMMn = WRITEn;
  INC_CNT_COMM = DELAY_WRITE;
END IF;

IF ( LSA[4..1]==CNT_COMM AND SR/Wn==1 ) THEN
  OE_ADR_COMMn = DATA_ENn;
END IF;

IF ( LSA[4..1]==COMM_BUF AND SR/Wn==0 ) THEN
  CE_COMM_BUFn = DATA_ENn;
  LD_COMM_BUFn = WRITEn;
  INC_CNT_COMM = PULSE;
END IF;

IF ( LSA[4..1]==COMM_BUF AND SR/Wn==1 ) THEN
  CE_COMM_BUFn = DATA_ENn;
  OE_COMM_BUFn = DATA_ENn;
  INC_CNT_COMM = PULSE;
END IF;

IF ( LSA[4..1]==LO_PRG AND SR/Wn==0 ) THEN
  LD_LO_DATA = DELAY_WRITE;
END IF;

IF ( LSA[4..1]==LO_PRG AND SR/Wn==1 AND OE_CPUn==1 ) THEN
  RAM_SELECT  = GND;
  READ_RAM    = VCC;
  OE_CNT_ININ = DATA_ENn;
  OE_LO_DATAn = DATA_ENn;
END IF;

IF ( LSA[4..1]==LO_CAD AND SR/Wn==0 ) THEN
  LD_LO_DATA = DELAY_WRITE;
END IF;

IF ( LSA[4..1]==LO_CAD AND SR/Wn==1 AND OE_CPUn==1 ) THEN
  RAM_SELECT  = VCC;
  READ_RAM    = VCC;
  OE_CNT_ININ = DATA_ENn;
  OE_LO_DATAn = DATA_ENn;
END IF;

IF ( LSA[4..1]==HI_PRG AND SR/Wn==0 AND OE_CPUn==1 ) THEN
  RAM_SELECT  = GND;
  OE_DATAn    = DATA_ENn;
  OE_CNT_ININ = DATA_ENn;
  WRITE_RAMn  = WRITEn;
  INC_CNT_INI = PULSE;
END IF;

IF ( LSA[4..1]==HI_PRG AND SR/Wn==1 AND OE_CPUn==1 ) THEN
  RAM_SELECT  = GND;
  READ_RAM    = VCC;
  OE_CNT_ININ = DATA_ENn;
  OE_HI_DATAn = DATA_ENn;
  INC_CNT_INI = PULSE;
END IF;

IF ( LSA[4..1]==HI_CAD AND SR/Wn==0 AND OE_CPUn==1 ) THEN
  RAM_SELECT  = VCC;
  OE_DATAn    = DATA_ENn;
  OE_CNT_ININ = DATA_ENn;
  WRITE_RAMn  = WRITEn;
  INC_CNT_INI = PULSE;
END IF;

IF ( LSA[4..1]==HI_CAD AND SR/Wn==1 AND OE_CPUn==1 ) THEN
  RAM_SELECT  = VCC;
  READ_RAM    = VCC;
  OE_CNT_ININ = DATA_ENn;
  OE_HI_DATAn = DATA_ENn;
  INC_CNT_INI = PULSE;
END IF;

IF ( LSA[4..1]==BAR AND SR/Wn==0 ) THEN
  BAR_LD = DELAY_WRITE;
END IF;

END;
```



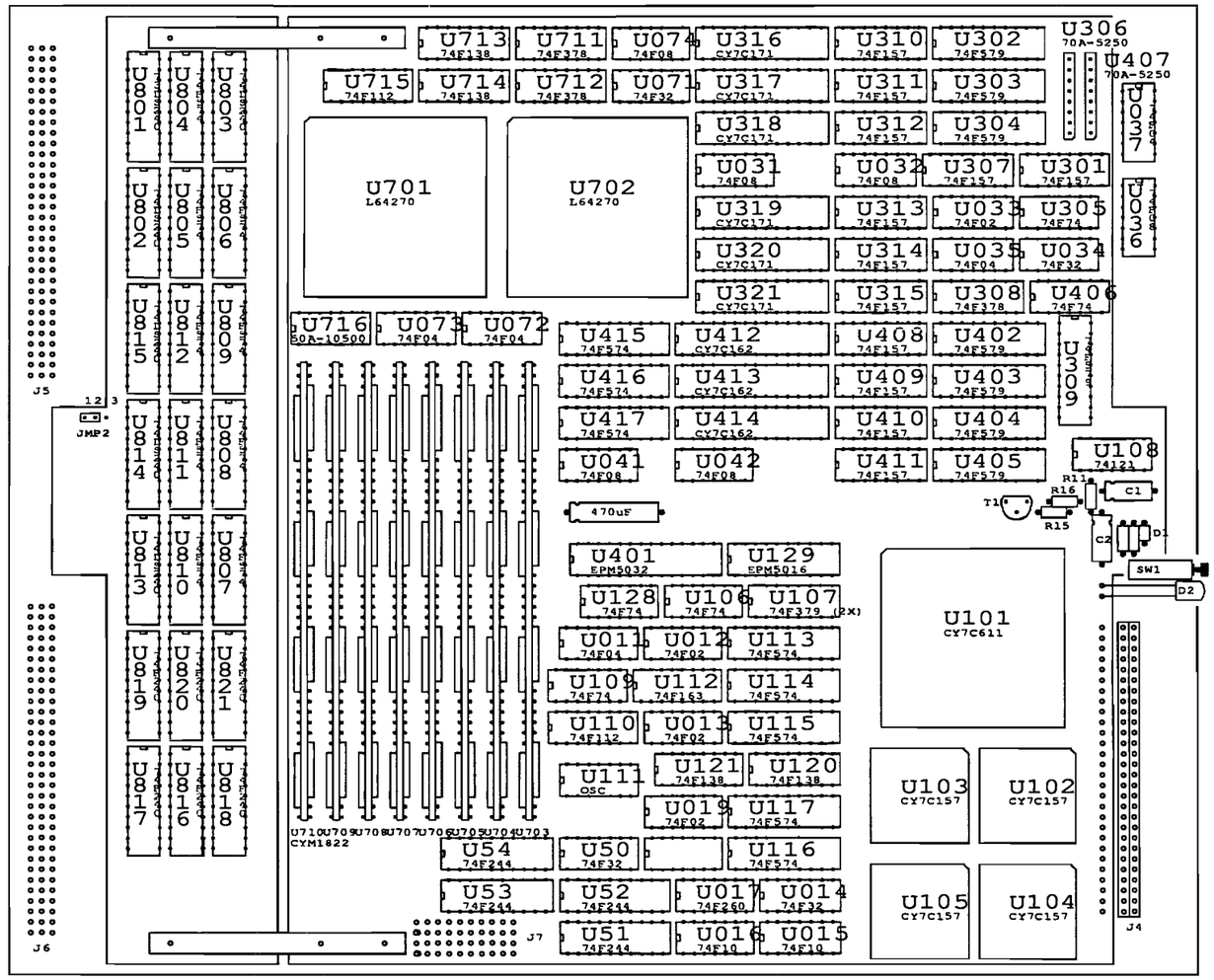
PHILIPS CFT F.G.M. Smeets & P.T.M. de WINNE		
Title: AREA SEGMENTATION PROCESSOR - COMM BUFFER		
Size: A2	Document Number: COMM_BUF.SCH	REV:
Date: February 24, 1993	Sheet	of

TOP VIEW
BOARD 1
ASP



PHILIPS CFT F.G.M. Smeets & P.T.M. de WINNE		
Title: AREA SEGMENTATION PROCESSOR - BOARD 1		
Size: A2	Document Number: BOARD1.SCH	REV:
Date: May 27, 1993	Sheet 12 of 14	

TOP VIEW
BOARD 2
ASP



PHILIPS CFT F.G.M. Smeets & P.T.M. de WINNE

Title
AREA SEGMENTATION PROCESSOR - BOARD 2

Size	Document Number	REV
A2	BOARD2.SCH	
Date: May 27, 1993	Sheet 3 of 14	

BOTTOM VIEW
CONNECTOR J4
CONNECTOR J7
ASP

BOTTOM VIEW
CONNECTOR J4

	C	B	A
1	5 V	5 V	5 V
2	5 V	5 V	5 V
3	D29	D30	D31
4	D26	D27	D28
5	D23	D24	D25
6	D20	D21	D22
7	D17	D18	D19
8	D14	D15	D16
9	D11	D12	D13
10	D8	D9	D10
11	D5	D6	D7
12	D2	D3	D4
13	SIZE1	D0	D1
14	SIZE0	FCLK1	SCLK1
15	A21	A22	A23
16	A18	A19	A20
17	A15	A16	A17
18	A12	A13	A14
19	A9	A10	A11
20	A6	A7	A8
21	A3	A4	A5
22	A0	A1	A2
23	WRT	WE*	RD
24	LWE*	TOE*	LRD*
25	WIQ	RESET	NULL
26	CS_DUART*	CS_COMBUF*	DCLK
27	RCO	INT_HOST	ACK_HOST
28			
29			
30			
31	GND	GND	GND
32	GND	GND	GND

BOTTOM VIEW
CONNECTOR J7

20		Ch1CtrlInN	1
19		Ch2CtrlInN	2
18		Ch1HiInN	3
17		Ch2HiInN	4
16		Ch1InInN	5
15		Ch2InINN	6
14		Ct1N	7
13	Ct2CN	Ct2N	8
12	In2N	HI1N	9
11	In1N	Hi2N	10

PHILIPS CFT F.G.M. Smeets & P.T.M. de WINNE

Title
AREA SEGMENTATION PROCESSOR - CONNECTOR J4

Size	Document Number	REV
A2	CON_J4.SHT	

Date: February 10, 1993 Sheet 14 of 14

Appendix C Program 512x512 image

```
/*****************************************************************************/
/*          file    : tes512.c                                               */
/*          language: C                                                      */
/*          designer: P.T.M. de Winne                                        */
/*          date    : 24/02/93                                               */
/*          routines: main ()                                                */
/*                    void set_interrupt_level ( int )                       */
/*                      sets the interrupt level in PSR register             */
/*                    void ask_unitsize_mode ( )                             */
/*                      asks user the unitsize and working mode              */
/*                    trap hardw_trap ()                                     */
/*                      handles hardware trap                                */
/*                    void init_xbar()                                       */
/*                      Sets the buswidth of the two crossbars.              */
/*                    void screen()                                          */
/*                      sends an screen of 512*512 pixels                    */
/*                                                                           */
/*****************************************************************************/
#define LASER
#if LASER
    #include "e:\peter\ermsw\sc\inc\asp.h_c"
#else
    #include "\sc\inc\asp.h_c"
#endif

#define LINE_LENGTH         512
#define LINES_PER_IMAGE     512

/* global variables */
int unitsize;    /* contains the unit size */
int unitsize_p;  /* contains the unit size exponent
                            example: unitsize=8 ==> unitsize_p=3*/
int mode;        /* contains the working mode (normal or windowed) */
int read_block;  /* contains the current read block */
int write_block; /* contains the current write block */
int rw_block;    /* contains (write_block<<11) + (read_block<<8) */
int line_buffer; /* contains the selected line buffer to RISC */
int lc_image_filled;    /* line counter image */
int lc_screen;   /* line counter screen */
int lc_image_read;

enum mode_t {normal, windowed};
enum to_lut_t {counters, risc};
enum to_lineb_t {A, B};

void main()
    {
    int i,j;

    init_processor();
    init_io();
    CLS();
    print("TEST OF THE AREA SEGMENTATION PROCESSOR\r\n\n");
    init_xbar();
    clear_line_buffers();
    clear_lut();
    clear_comm_buf();
    ask_unitsize_mode();

    set_interrupt_level(0);
    enable_screen(1);

    lc_image_filled = 0;
    lc_screen = 0;
    lc_image_read = 0;

    print("Processor State Register %x.\r\n", REGISTER_PSR);
    print("The unitsize is %d and the mode is %d.\r\n", unitsize, mode);
    print("The image line counter is %d\r\n/
            The screen line counter is %d.\r\n\n\n", lc_image_read, lc_screen);

    image();
    screen();

    print("\r\nThe program is finished!\r\n");
    delay(2);

    exit();
    }

void image()
    /*  function: recieves an image of 512*512 pixels
        pre     :
        post    :
    */
    {
    int j;

    CLS();
    print("Receiving an image\r\n\n\n");
    lb_select(A);
    fill_lb();
    lb_select(B);
    fill_lb();
    set_write_block(0);
    config_xbar_write(1-line_buffer);
    print("The current write block is: %d.\r\n", write_block);
    print("The selected linebuffer for the processor is: ");
    putch(line_buffer + 'A');
    print(".\r\nThe lc_image_read is %d.\r\n", lc_image_read);
    print("The lc_image_filled is %d.\r\n", lc_image_filled);
    print("\r\nReady to recieve an image\r\n");
    while ( lc_image_read < (LINES_PER_IMAGE/unitsize) )
        {
        setXY(0,10);
        DELEOL();
```

```
        print("The line counter image read is: %d.",lc_image_read);
      }
    print("\r\n\nThe line counter read is %d. \r\nThe image is read!. \r\n",
lc_image_read);
    print("Image read!\r\n");

    }   /* end of image */

void screen()
    /*  function: sends an screen of 512*512 pixels
        pre     :
        post    :
    */
    {
    int i, j, lines_pb, lines_send;

    CLS();
    print("Sending a screen\r\n\n");
    fill_lut();
    print("Look up table is filled\r\n");
    lines_send = 0;
    lines_pb = LINES_PER_IMAGE/(unitsize*8);
    i = 0;
    do
        {
        set_read_block(i);
        config_xbar_read();
        if ( i == 7 )
            send_burst( lines_pb, 1 );
        else
            send_burst( lines_pb, 0 );
        j=0;
        lines_send = lines_send + lines_pb;
        while (lc_screen < lines_send )
            {
            j++;
            }
        print("Block %d is send. Waited %d clockcycles.\r\n",i,j);
        i++;
        }
    while (i < 8);

    }   /* end of screen */

void ask_unitsize_mode ( )
    /*  function: Asks the user on the terminal which unitsize must be
                  used and in which mode must be used.
        pre     : <none>
        post    : unitsize is one of (4,8,16,32) ^ unitsize_p = (2, 3, 4, 5) ^
                  unitsize = 2exp(unitsize_p) ^ mode is normal or windowed ^

                  unitsize-1+(mode<<5) is places into register U308
    */
    {
    int keyb;
    char *unitsize_reg;

    do
        {
        print("\n\rGive the new unit size 4, 8, 16, 32 (end with <rt>): ");
        keyb = 4 /*get_dec()*/;
        }
    while ( !( (keyb == 4) || (keyb == 8) || (keyb == 16) || (keyb == 32) ) );
    unitsize = keyb;
    for (unitsize_p = 0; keyb > 1; keyb = keyb >> 1)
        unitsize_p ++;
    print("\r\nGive the new mode (NORMAL = n, WINDOW = w): ");
    do
        {
        keyb = 'w' /*getch()*/;
        }
    while( !( (keyb=='n') || (keyb=='N') || (keyb=='w') || (keyb=='W') ));
    putch(keyb);
    print("\r\n");
    if ( (keyb =='n') || (keyb == 'N') )
        mode = normal;
    else
        mode = windowed;
    unitsize_reg = UNIT_SIZE_REGISTER;
    *unitsize_reg = mode*32 + unitsize-1;
    }   /* end of _ask_unitsize_mode */

void delay (int del)
    {
    int i;
    for( ;del >= 0; del--)
        {
        for (i=0; i < 1000000; i++);
        }
    } /* end of delay */

trap hardw_trap()
    /*  function: handles a hardware trap
        pre     :
        post    :
    */
    {
    int numb;
    int psr;

    psr = REGISTER_PSR;
    if ( (numb != 3) || ( (numb==3) && ((psr&0x40) != 0) ) )
        {  /* 3 = privileged instruction */
        print("\r\nHardware trap %d occurred at pc: %x , npc: %x.\r\n",numb,/
                                                    REGISTER_L1, REGISTER_L2);
        while (1) {};
```

```
    }
    else        /* switch to supervisor mode */
        REGISTER_PSR = psr|0x40;
    } /* end of hardw_trap */

void init_xbar()
    /*  function: Sets the buswidth of the two crossbars.
        pre     : <none>
        post    : buswidth of crossbar A is 1 ^ buswidth of crossbar A is 1 ^
                  read_block=7 ^ write_block = 0 ^ U711 is loaded with 7 ^
                  rw_block = 0x700
    */
    {
    short *xbar

    xbar = CROSS1_BASE + 0x100;
    *xbar = 0x700;
    xbar = CROSS2_BASE + 0x100;
    *xbar = 0x700;
    read_block = 7;
    write_block = 0;
    rw_block = 0x700;
    } /* end of init_xbar */

void pause()
    {
    print("Press a key to continue!");
    getch();
    print("\r\n");
    }

int read_control(int offset)
    {
    short *read_ctrl;

    read_ctrl = CS_CTRL+offset;
    return *read_ctrl;
    }

void set_interrupt_level (int level)
    /*  function: Sets the interrupt level of the SPARC RISC processor. An
                  interrupt greater then level can interrupt the processor
                  when the traps are enabled.
        pre     : level = 0..15 ^ processor mode is supervisor
        post    : Processor interrupt level = level
    */
    {
    int psr;
    psr = REGISTER_PSR;
    psr = psr&0xfffff0ff;
    psr = psr|(level<<8);
    REGISTER_PSR = psr;
    }   /* end of set_interrupt_level */
```

```
/*****************************************************************************/
/*          file    : imag_rou.c                                             */
/*          language: C                                                      */
/*          designer: P.T.M. de Winne                                        */
/*          date    : 25/02/93                                               */
/*          routines: void clear_line_buffers()                              */
/*                      clears linebuffer A and B                            */
/*                    void config_xbar_write(int lb )                        */
/*                      connects the image counters and linebuffer lb to     */
/*                      the current write block by switching the crossbars   */
/*                    void lb_select( int )                                  */
/*                      The path for the RISC is opened to linebuffer lb     */
/*                    trap lbll ()                                           */
/*                          handles interrupt 1                              */
/*                    trap next_lb ()                                        */
/*                          handles interrupt 4                              */
/*                    void set_write_block(int block)                        */
/*                          fills write_block with the value of block        */
/*                          and fills U711                                   */
/*                                                                           */
/*****************************************************************************/
#define LASER
#ifdef LASER
  #include "e:\peter\ermsw\sc\inc\asp.h_c"
#else
  #include "\sc\inc\asp.h_c"
#endif

#define LINEB_MAX          4096
#define LINEB_MAX_4       16384
#define LINE_LENGTH         512
#define LINE_LENGTH_4      2048
#define LINES_PER_IMAGE     512
#define MAX_BLOCK         32768


#define IM_CC_FIRST           0
#define IM_LC_FIRST           5
#define LBA_FIRST            10
#define LBB_FIRST            21

enum line_t {A, B};

extern int unitsize;
extern int unitsize_p;
extern int write_block;
extern int read_block;
extern int rw_block;
extern int line_buffer;
extern int lc_image_read;
extern int lc_image_filled;

void clear_line_buffers()
```

```
    /*  function: Clears linebuffer A and linebuffer B.
        pre     : <none>
        post    : Linebuffer A and B are filled with zeros. RISC
                  is connected to linebuffer B; counters to linebuffer A.
    */
    {
    int i;
    short *lineb;

    lineb = LINE_BUFFER_BASE;
    lb_select(A);
    for (i = 0; i < LINEB_MAX_4; i = i + 4 )
        *(lineb+i) = 0;
    lb_select(B);
    for (i = 0; i < LINEB_MAX_4; i = i + 4 )
        *(lineb+i) = 0;
    }

void config_xbar_write( int lb )
    /*  function: connects the image counters and linebuffer lb to the current
                  write block by switching the crossbars
        pre     : unitsize_p = {2, 3, 4, 5} ^ write_block = {0,1,2,3,4,5,6,7}
                  ^ lb = {A, B}
        post    : image counter and linebuffer lb are connected to write_block
    */
    {
    int i, unit, rw;
    short *xbar_out;

    /* calculate the address of the first output line */
    /* remind addressline 0,1 are not used so factor 64 is used instead of
16*/
    if (write_block >= 4)
        xbar_out = CROSS1_BASE + 64*(write_block-4);
    else
        xbar_out = CROSS2_BASE + 64*write_block;

    unit = unitsize_p;
    rw = rw_block;
    rw = rw + IM_CC_FIRST;
    i = 0;
    do  {   /* connect the image column counter LSB*/
        *xbar_out = rw + i;
        xbar_out = xbar_out + 4;
        i++;
        }
    while ( i < unit );

    rw = rw_block;
    rw = rw + IM_LC_FIRST;
    i = 0;
    do  {   /* connect the image line counter LSB*/
        *xbar_out = rw + i;
        xbar_out = xbar_out + 4;
        i++;
        }
    while ( i < unit );

    rw = rw_block;
    if (lb == A)
        rw = rw + LBA_FIRST;
    else
        rw = rw + LBB_FIRST;

    i = 2*unit - 4;
    do  {   /* connect the linebuffer MSB */
        *xbar_out = rw + i;
        xbar_out = xbar_out + 4;
        i++;
        }
    while (i < 11);
    }    /* end of config_xbar_read */


void fill_lb()
    /*  function: fills the line buffer
        pre     :  unitsize ^ lc_image_filled are vallid
        post    : lc_image_filled = lc_image_filled + 1 ^ linebuffer is filled
    */
    {
    short *lb;
    int i, j, max_block;

    lb = LINE_BUFFER_BASE;
    i=0;
    max_block = MAX_BLOCK/(unitsize*unitsize);
    j = (lc_image_filled*LINE_LENGTH/unitsize)%max_block;
    do
        {
        *lb = j;
        j++;
        i = i + unitsize;
        lb = lb + unitsize;
        }
    while (i < LINE_LENGTH);

    lc_image_filled = lc_image_filled + 1;
    }

void lb_select(int lb)
    /*  function: The path for the RISC is opened to linebuffer lb
        pre     : lb one of { A, B}
        post    : RISC is connected to linebuffer lb ^ line_buffer = lb
    */
    {
    char *lbs;
```

```
    lbs = LINE_BUFFER_SEL;
    *lbs = lb;
    line_buffer = lb;
    }

trap lbll()
    /*  function: interrupt service routine for lbll (line buffer bit 11)
        pre     :   not used
        post    :
    */
    {
    print("\r\n interrupt line buffer 11.\r\n");
    }

trap next_lb()
    /*  function: interrupt service routine after the line buffers swapped
                  their function
        pre     : line_buffer is one of (0, 1)
        post    : line_buffer is swapped ^ xbar is reprogrammed ^ linebuffer
                  is filled
    */
    {

    lc_image_read++;
    if ( (lc_image_read % (LINES_PER_IMAGE/(unitsize*8))) == 0)
        {
        set_write_block((write_block+1)&0x7);
        set_read_block((read_block+1)&0x7);
        };
    config_xbar_write(line_buffer);
    line_buffer = 1 - line_buffer;
    fill_lb();

    }   /* end of next_lb */

void set_write_block(int block)
    /*  function: fills write_block with the value of block and fills U711
        pre     : block = (0, .. ,7)
        post    : write_block = block ^ U711 is filled with write_block
                  and read_block
    */
    {
    short *xbar;

    write_block = block;

    xbar = CROSS1_BASE + 0x100;
    rw_block = (write_block<<11) + (read_block<<8);
    *xbar = rw_block;
    }   /* end of set_write_block */
```

```
/*****************************************************************************/
/*          file    : scre_rou.c                                             */
/*          language: C                                                      */
/*          designer: P.T.M. de Winne                                        */
/*          date    : 25/02/93                                               */
/*          routines: void clear_lut()                                       */
/*                        clears the look up table                           */
/*                    void config_xbar_read( )                               */
/*                        connects the screen counters and the LUT to the    */
/*                        current read block by switch the crossbars         */
/*                    void enable_screen()                                   */
/*                        restart sending a screen (burst) after llutll      */
/*                        interrupt                                          */
/*                    trap end_of_burst ()                                   */
/*                        handles interrupt 3                                */
/*                    void fill_lut()                                        */
/*                        fills the look up table                            */
/*                    trap llutll ()                                         */
/*                        handles interrupt 2                                */
/*                    void path_to_lut( int )                                */
/*                        Connects the counters or the RISC to the look      */
/*                        up table. The counters are reset.                  */
/*                    void send_burst(int units, int last)                   */
/*                        sends a burst with units*unitsize lines and        */
/*                        reset scr_pie when last = 1                        */
/*                    void send_screen( )                                    */
/*                        sends a screen                                     */
/*                    void set_read_block(int block)                         */
/*                        fills read_block with the value of block           */
/*                        and fills U711                                     */
/*****************************************************************************/
#define LASER
#ifdef LASER
  #include "e:\peter\ermsw\sc\inc\asp.h_c"
#else
  #include "\sc\inc\asp.h_c"
#endif

#define LUT_MAX           16384
#define LUT_MAX_4         65536
#define LINE_LENGTH         512
#define LINES_PER_SCREEN    512
#define MAX_BLOCK         32768

#define SCR_CC_FIRST         32
#define SCR_LC_FIRST         37
#define LLUT_FIRST           42

enum to_lut_t {counters, risc};

extern int unitsize;
extern int unitsize_p;
```

```
extern int read_block;
extern int write_block;
extern int rw_block;
extern int lc_screen;


void clear_lut()
    /*  function: clears the look up table
        pre     : <none>
        post    : The lut is filled with zeros.
                  The RISC is connected to the lut.
    */
    {
    short *lut;
    int i;

    lut = LUT_BASE;
    path_to_lut(risc);
    for (i = 0; i < LUT_MAX_4; i=i+4)
        *(lut+i) = 0;
    }   /* end of clear_lut */

void config_xbar_read( )
    /*  function: connects the screen counters and the LUT to the current
                  read block by switch the crossbars
        pre     : unitsize_p = {2, 3, 4, 5} ^ read_block = {0,1,2,3,4,5,6,7}
        post    : screen counter and LUT are connected to the read_block
    */
    {
    int i, unit, rw;
    short *xbar_out;

    /* calculate the address of the first output line */
    /* remind addressline 0,1 are not used so factor 64 is used instead of
16*/
    if (read_block < 4)
        xbar_out = CROSS2_BASE + (read_block<<6);
    else
        xbar_out = CROSS1_BASE + ((read_block-4)<<6);

    unit = unitsize_p;
    rw = rw_block;
    rw = rw + SCR_CC_FIRST;
    i = 0;
    do      /* connect the screen column counter LSB*/
        {
        *xbar_out = rw + i;
        xbar_out = xbar_out + 4;
        i++;
        }
    while (i < unit);
    rw = rw_block;
    rw = rw + SCR_LC_FIRST;
    i = 0;
    do  {   /* connect the screen line counter LSB*/
        *xbar_out = rw + i;
        xbar_out = xbar_out + 4;
        i++;
        }
    while (i < unit) ;
    rw = rw_block;
    rw = rw + LLUT_FIRST;
    i = (unit<<1) - 4;
    do  {   /* connect the LUT MSB */
        *xbar_out = rw + i;
        xbar_out = xbar_out + 4;
        i++;
        }
    while (i < 11);
    }    /* end of config_xbar_read */

void enable_screen(data)
int data;
    /*  function: restart sending a screen (burst) after interrupt llutl1 or
                  starts only the screen clock
        pre     : data = { 0, 1}
        post    : sending a screen (burst) is restarted if data = 0
                  only the clock is enabled if data = 1
    */
    {
    char *en;

    en = ENABLE_SCREEN_BASE;
    *en = data;
    }   /* end of enable_screen */

trap end_of_burst()
    /*  interrupt service routine after end of burst occured
        function:
        pre     :
        post    :
    */
    {
    lc_screen = lc_screen + LINES_PER_SCREEN/(unitsize*8);
    } /* end of end of burst */

void fill_lut()
    /*  function: Fills the look up table
        pre     : <none>
        post    : look up table is filled ^ counters are connected to lut
    */
    {
    int i,j, max_block;
    short *lut;

    path_to_lut(risc);
```

```
    lut = LUT_BASE;
    i=0;
    max_block = MAX_BLOCK/(unitsize*unitsize);
    j=0;
    do
        {
        *lut = j;
        j++;
        if (j >= max_block) j=0;
        i = i + unitsize;
        lut = lut + unitsize;
        }
    while (i < LUT_MAX_4);
    path_to_lut(counters);

    } /* end of fill_lut */


trap llut11()
    /*  interrupt service routine form llut11 (latched look up table bit 11)

        function: not used
        pre     :
        post    :
    */
    {
    print("\r\ninterrupt llut 11.\r\n");
    } /* end of llut11 */

void path_to_lut(int select)
    /*  function: Connects the counters or the RISC to the look up table.
                  The counters are reset.
        pre     : select is one of (counters, risc)
        post    : The lut is connected to select. The counters are reset.
    */
    {
    char *lut;

    lut = TO_LUT_BASE;
    *lut = select;
    }   /* end of path_to_lut */

void send_burst(int lines, int last)
    /*  function: sends a burst with lines*unitsize lines and
                                reset scr_pie when last = 1
        pre     : lines = (1, .. , 128) ^ last = (0, 1)
        post    : units*unitsize lines are send
    */
    {
    short *send_burst;

    send_burst = SEND_SCREEN_BASE;
    if (last == 1)
        lines = lines + 256;
    *send_burst = lines;
    }   /* end of send_burst */

void send_screen( )
    /*  function: sends a screen
        pre     : <none>
        post    : a screen is send
    */
    {
    short *send_scr;

    path_to_lut(counters);      /* reset counters */
    send_scr = SEND_SCREEN_BASE;
    *send_scr = 384;
    }   /* end of send_screen */

void set_read_block(int block)
    /*  function: fills read_block with the value of block and fills U711
                  and rw_block
        pre     : block = (0, .. ,7)
        post    : read_block = block ^ U711 is filled with write_block
                  and read_block ^ rw_block is filled
    */
    {
    short *xbar;

    read_block = block;

    xbar = CROSS2_BASE + 0x100; /* write to buswith address */
    rw_block = (write_block<<11) + (read_block<<8);
    *xbar = rw_block;
    } /* end of set_read_block */
```

```
/*****************************************************************************/
/*          file    : comm_rou.c                                             */
/*          language: C                                                      */
/*          designer: P.T.M. de Winne                                        */
/*          date    : 25/02/93                                               */
/*          routines: trap host_int ()                                       */
/*                          handles interrupt 5                              */
/*                    void clear_comm_buf()                                  */
/*                          clears the communication buffer                  */
/*                                                                           */
/*****************************************************************************/
#define LASER
#ifdef LASER
  #include "e:\peter\ermsw\sc\inc\asp.h_c"
#else
  #include "\sc\inc\asp.h_c"
#endif

#define COMM_MAX    2048
#define COMM_MAX_4  8192


trap host_int()
    /*  function: handles the interrput from the host.
        pre     :
        post    :
    */
    {
    print("\r\nInterrupt from the host recieved\r\n");
    }

void clear_comm_buf()
    /*  function: clears the communication buffer
        pre     : <none>
        post    : the communication buffer (U1106, U1107) is filled with zeros
    */
    {
    int i;
    short *commb;

    commb = COMM_BUFF_BASE;
    for( i=0; i < COMM_MAX_4; i= i+4)
        *(commb+i) = 0;
    }
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!       file    : init2.a
!       lanuage : assembly
!       designed: P. de Winne
!       date    : 24/02/93
!       routine : _init_processor (initialisation of the SPARC RISC processor)
!                 _exit (leaves the program)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!#include c:\sc\inc\asp.h
#include e:\peter\ermsw\sc\inc\asp.h

.tabsize  8
.seg text

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!       void init_processor ()
!           function:   initialises the processor
!           pre:    processor mode is supervisor
!           post:   WIM-register = 0, Y-register = 0, TBA-register = 0
!                   current window = 7, co-processor disabled, floating
!                   point processor disabled, mode = supervisor, traps
enabled,
!                   Processor Interrupt Level = 15, stack pointer loaded.
!
.global _init_processor
.extrn  __stack
_init_processor:
        wr      %g0, %g0, %wim          ! init window invalid mask
        mov     %o7, %g7                ! save the return address
        wr      %g0, 0x400fe7, %psr     ! init processor state register
        mov     %g7, %o7                ! put back the return address
        wr      %g0, %g0, %y            ! init y register
        sethi   %hi(_TRAP_BASE), %l0
        or      %l0, %lo(_TRAP_BASE), %l0
        wr      %l0, %g0, %tbr          ! init trap base register
        sethi   %hi(__stack), %o6       ! init the stack pointer
        or      %o6, %lo(__stack), %o6
        retl
        nop
! end of _init_processor

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!       void exit ()
!           function:   leaves the program
!           pre:    <none>
!           post:   program counter = 0
!
.global _exit
_exit:
```

```
        jmpl    %g0, %g0, %g0
        nop
! end of _exit

.end
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!       file    : trap_jum.a
!       language: assembly

!       designer: P de Winne
!       date    : 24/02/93
!                 trap jumps
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
.extrn _main
.extrn _hardw_trap
.extrn _lbil
.extrn _llutil
.extrn _end_of_burst
.extrn _next_lb
.extrn _host_int

.seg trap

!!!!!!!!!!!!!!
!
!      code for reset (should be placed at trapbase)
!
reset_jmp:
    nop
    nop
    ba    _main
    nop
!
!      code for instruction access (should be placed at trapbase + 0x10)
!
instr_acc_jmp:
    sethi   %hi(_hardw_trap), %l0
    or      %l0, %lo(_hardw_trap), %l0
    jmpl    %l0, %g0, %g0
    mov     1, %l3
!
!      code for illegal instruction (should be placed at trapbase + 0x20)
!
ill_instr_jmp:
    sethi   %hi(_hardw_trap), %l0
    or      %l0, %lo(_hardw_trap), %l0
    jmpl    %l0, %g0, %g0
    mov     2, %l3
!
!      code for privileged instruction (should be placed at trapbase + 0x30)
!
prev_instr_jmp:
    sethi   %hi(_hardw_trap), %l0
    or      %l0, %lo(_hardw_trap), %l0
    jmpl    %l0, %g0, %g0
    mov     3, %l3
!
```

```
!      code for floating point disabled (should be placed at trapbase + 0x40)
!
float_dis_jmp:
    sethi   %hi(_hardw_trap), %l0
    or      %l0, %lo(_hardw_trap), %l0
    jmpl    %l0, %g0, %g0
    mov     4, %l3
!!!!!!!!!!!!!
!!!!!!!!!!!!!
!
!      code for window overflow (should be placed at trapbase + 0x50)
!
win_over_jmp:
    sethi   %hi(_hardw_trap), %l0
    or      %l0, %lo(_hardw_trap), %l0
    jmpl    %l0, %g0, %g0
    mov     5, %l3
!
!      code for window underflow (should be placed at trapbase + 0x60)
!
win_under_jmp:
    sethi   %hi(_hardw_trap), %l0
    or      %l0, %lo(_hardw_trap), %l0
    jmpl    %l0, %g0, %g0
    mov     6, %l3
!
!      code for address not aligned (should be placed at trapbase + 0x70)
!
addr_not_jmp:
    sethi   %hi(_hardw_trap), %l0
    or      %l0, %lo(_hardw_trap), %l0
    jmpl    %l0, %g0, %g0
    mov     7, %l3
!
!      code for floating point execption (should be placed at trapbase + 0x80)
!
float_e_jmp:
    sethi   %hi(_hardw_trap), %l0
    or      %l0, %lo(_hardw_trap), %l0
    jmpl    %l0, %g0, %g0
    mov     8, %l3
!
!      code for data access exeption (should be placed at trapbase + 0x90)
!
data_a_jmp:
    sethi   %hi(_hardw_trap), %l0
    or      %l0, %lo(_hardw_trap), %l0
    jmpl    %l0, %g0, %g0
    mov     9, %l3
!
!      code for data tag overflow (should be placed at trapbase + 0xa0)
!
tag_over_jmp:
    sethi   %hi(_hardw_trap), %l0
    or      %l0, %lo(_hardw_trap), %l0
    jmpl    %l0, %g0, %g0
    mov     10, %l3

.org 0x110

!
!      jump for interrupt 1: lbll (should be placed at trapbase + 0x110)
!
inter1:
    sethi   %hi(_lbll), %l0
    or      %l0, %lo(_lbll), %l0
    jmpl    %l0, %g0, %g0
    nop
!
!      jump for interrupt 2: lutll (should be placed at trapbase + 0x120)
!
inter2:
    sethi   %hi(_llutll), %l0
    or      %l0, %lo(_llutll), %l0
    jmpl    %l0, %g0, %g0
    nop
!
!      jump for interrupt 3: end_of_burst (should be placed at trapbase +
0x130)
!
inter3:
    sethi   %hi(_end_of_burst), %l0
    or      %l0, %lo(_end_of_burst), %l0
    jmpl    %l0, %g0, %g0
    nop
!
!      jump for interrupt 4: next_lb (should be placed at trapbase + 0x140)
!
inter4:
    sethi   %hi(_next_lb), %l0
    or      %l0, %lo(_next_lb), %l0
    jmpl    %l0, %g0, %g0
    nop
!
!      jump for interrupt 5: host_int (should be placed at trapbase + 0x150)
!
inter5:
    sethi   %hi(_host_int), %l0
    or      %l0, %lo(_host_int), %l0
    jmpl    %l0, %g0, %g0
    nop

.end
```

Appendix D Program address generation

```
/*
 *  Acronym             :   IMAGSCR2.C
 *  Name of the module  :   IMAGSCR2.C
 *  Product/Project     :   Area Segmentation Processor LINE BUFFER AND LUT
 *                              content calculation
 *  Group number        :
 *  Creation date       :   1993-03-25
 *  Modification date   :   ....-..-..
 *  Document language   :   010 (English)
 *  Program language    :   C
 *  Status              :   Preliminary
 *  Name author         :   P.T.M. de WINNE
 *
 *                  Nederlandse Philips Bedrijven B.V.
 *              CFT   Centre For manufacturing Technology
 *                      CAM-Centre _ MMSP Dept.
 *                    Eindhoven - The Netherlands
 *
 *          Copyright N.V. PHILIPS' Gloeilampenfabriek 1987
 *     All rights are reserved. Reproduction in whole or in part is
 *     prohibited without the written consent of the copyright owner.
 *
*/

/*
 * This program is a test program for routines which can be used on the
 * Area Segmentation Processor (ASP). The ASP segments images with control of
 * CAD data, stores the AOIs (Area Of Interest specified in the CAD data) in a
 * circular buffer and composes these stored AOIs into screens which are send
 * away.
 *
 * An image is divided into units. A unit is a square of 4x4 (8x8, 16x16 or
 * 32x32) pixels. The size of a unit is specified by the unitsize (4, 8, 16 or
 * 32).
 *
 * Writing the AOIs into the circular buffer is controlled by the image
 * handler. Every unit in the image gets an address. When the unit is in an
 * AOI it is an unique address else it becomes address zero (the dummy
 * address). The addresses of one line are placed into a linebuffer. A counter
 * counts the units. The value of the counter is used as an address for the
 * linebuffer. The data of the linebuffer is than the address for the unit in
 * the circular buffer.
 *
 * The AOIs are composed the screens. Sending a screen is controlled by the
 * screen handler. The screen handler contains a look up table in which the
 * address of the units are placed.
 *
 * Calculating the content of a screen is done on the host computer (not on
 * the ASP). The information is send to the ASP as CAD data. The ASP
 * calculates with this information the content of the linebuffer and the look
 * up table.
 *
 * The calculation of the linebuffer and the look up table is time consuming.
 * Therefor some linebuffers and a part of the LUT is calculated and kept in
 * arrays in the memory. When an interrupt NEXT_LB or END_OF_BURST occurs a
 * linebuffer or a part of the LUT is copied to the appropriate RAM. These
 * copy routines are in the program interchanged by print routines to files.
 *
 * The calculation of the linebuffers and LUT is done in "compute", printing
 * linebuffers is done by "show_lines" and printing the LUT is done by
 * "out_screen".
*/
/*
 *  ------------------------ Include Specification ------------------------
*/
#include <d:\msc5\include\stdio.h>
#include <d:\msc5\include\stdlib.h>
#include <d:\msc5\tmp\x1032p2s.c>    /* The file with the array of aois */
/*
 *  ------------------------ Define Specification -------------------------
*/
#define BUF_NUM 5   /* the number of linebuffers stored into memory */
#define SCR_BUF 32  /* the number of screen lines stored into memory */
#define SCR_WIDTH 128 /* the screen width the minimum unitsize of 4 is used */
/*
 *  -------------------- Gloabal variable Specification --------------------
*/
int *lines[BUF_NUM];
int inl;
int outl;
int *scr_fill[SCR_WIDTH];
int used[SCR_BUF];
int screen[SCR_BUF][SCR_WIDTH];
int lc_image;
int lc_out;
int address;
int screen_nr;
int lc_screen;
int block;
int last_block;
int last_pos;
int last_val;
int switch_buf[32];
int switch_index;
/*---------------------- Explanation of the global variables
-------------------
 *
 * The calculated line buffers are stored into the elements of "lines". The
 * elements of "lines" are here dynamically allocated as arrays of length 128.
 * "inl" is an index to the first free array in "lines" and "outl" is an index
 * to the first array to be printed.
 *
 * The routine compute calculate as long there is space in "lines", then
 * show_lines outputs linebuffers so that one linebuffer is left in "lines".
 * This is necessary because the circular buffer is divided into 8 eight
 * blocks. Switching a block is done a the end of a line. When compute
```

```
 * experience that the units of one line doesn't fit into the block, bit 11 of
 * the last unit in the previous linebuffer is set to generate an interrupt
 * (lbll) to the processor. It must be possible to set the interrupt bit so
 * one linebuffer must stay in "lines" except for the last of course.
 *
 * The variable that counts the lines of an image is "lc_image". This variable
 * counts the computed linebuffers placed in "lines", not the number of
 * linebuffer put in the RAM. Because the routine compute is left the variable
 * "address" (that kept the next unique address for a unit) is also global.
 * The current block in which the units are placed is kept in "block".  The
 * value of the block number (0..7) is placed into bit 12, 13 and 14. This is
 * to the simplify the calculation. The "lc_out" is used in show_lines. It
 * counts the lines which are printed. Because the routine show_lines is left
 * this variable must be global.
 *
 * The lines for the LUT are stored into "screen". "screen" must be seen as an
 * array of which the element also consists of an array (the x direction).
 * Here it can contain 32 lines of the LUT. Because placing the addresses in
 * "screen" is not done sequential in y direction the arrays in "screen"
 * doesn't have to be sequential. At most 32 elements of "fill_scr" point to
 * an array of "screen". An element in "used" keeps the screen line numbers.
 * When the array in "screen" is not used the element contains -1. An element
 * in "screen" contains an address (bit 10..0), an interrupt bit (bit 11) to
 * interrupt the processor when a block has to be switched and the block
 * number in which the units is placed (bit 14..12).
 *
 * The variable "lc_screen" counts the lines of the LUT which are printed.
 * "screen_nr" counts the number of screens. Both variables are used in
 * out_screen.
 *
 * The lines for the LUT are printed in fill_lut_line. There by it is
 * necessary to set the interrupt bit and calculate a list which contains the
 * order in which the block has to be switched. This switch list for one line
 * is kept in "switch_buf" with the index pointing to the next free position
 * "switch_index". The "switch_buf" is printed when the next linebuffer
 * arrives and is therefor kept global. The variable "last_block" keeps the
 * block number of the previous pixel(s). "last_pos" and "last_val" contain
 * the position and value where the interrupt bit must be set when the block
 * changes.
*/

/*
 *  ------------------------ Structure definions -------------------------
*/
struct aoi_type  {
  int ix,     /* x-coord aoi in the image, */
      iy,     /* y-coord aoi in the image, */
      w,      /* width aoi,                */
      l,      /* length aoi,               */
      snr,    /* screen number,            */
      sx,     /* x-coord in the screen,    */
      sy;     /* y-coord in the screen     */
              };
```

```
/*
 *  -------------------- Forward Procedure Declaration --------------------
*/
void compute( FILE * , FILE * , int, int, int);
void show_lines( FILE * , int, int );
void out_line( FILE * , int );
void out_screen( FILE * , FILE *, int, int);
void fill_lut_zero( FILE * );
void fill_lut_line( FILE * , FILE * , int);

/*
 *  ------------------------- The main function ----------------------------
*/
main()
  {
  int i, j, k;
  FILE *fpolb, *fposcr, *fposw;
  int unitsize, units_per_line, lines_per_image, units_per_block;

  unitsize = 4;
  units_per_line = 1200/unitsize;              /* 12000 pixels per line */
  lines_per_image = 2000/unitsize;             /* 20000 lines per image */
  units_per_block = (unsigned)32768/(unitsize*unitsize);
                          /* 32768 halfwords (2bytes) per block; unitsize=4 */
  in1 = 0;
  out1 = 0;
  address = 1;
  lc_image = 0;
  lc_out = 0;
  screen_nr = 1;
  lc_screen = 127;
  block = 0;
  switch_index = 1;
  switch_buf[0] = 0;
  last_block = 0;
  last_pos = 1;

  for( k = 0; k < SCR_BUF; k++ )
    used[k] = -1;

  for ( i=0; i < SCR_BUF ; i++ )
    for( j = 0; j < 512/unitsize ; j++ )
      screen[i][j] = 0;

  for( i = 0; i < BUF_NUM; i++ )
    lines[i] = calloc(units_per_line, sizeof(int) );

  fpolb = fopen("d:\\temp\\lb.txt", "w"); /* file pointer output line buffer
*/
  if (fpolb == NULL )
    {
    printf("Can't open line buffer file.\r\n");
    exit(0);
```

```
    };

  fposcr = fopen("d:\\temp\\scr.txt", "w");
  if (fposcr == NULL )
    {
    printf("Can't open screen file.\r\n");
    exit(0);
    };

  fposw = fopen("d:\\temp\\switch.txt", "w");
  if (fposw == NULL )
    {
    printf("Can't open switch file.\r\n");
    exit(0);
    };

  compute(fposcr, fposw, unitsize, units_per_line, units_per_block);
  while( lc_image < lines_per_image )
    {
    show_lines(fpolb, lines_per_image, units_per_line);
    compute(fposcr, fposw, unitsize, units_per_line, units_per_block);
    }

  show_lines(fpolb,lines_per_image,units_per_line);/*print the last linebufs*/
  out_screen(fposcr, fposw, 1, unitsize);/*print the last content of the LUT*/
  if ( switch_index == 1)                    /* complete the switch list */
    fprintf(fposw, "%d, \r\n",switch_buf[0]);
  else
    {
    for ( j = 0; j < unitsize; j++ )
      for ( k = 0; k < switch_index; k++ )
        fprintf(fposw, "%d, ", switch_buf[k]);
    fprintf(fposw, "\r\n");
    }

  fclose(fpolb);
  fclose(fposcr);
  fclose(fposw);

  for( i = 0; i < BUF_NUM; i++)
    free( lines[i] );

  exit(0);
  }

void compute(fpout, fpouts, unitsize, units_per_line, units_per_block)
  FILE *fpout, *fpouts;
  int unitsize, units_per_line, units_per_block;
/*
 * "compute" calculates the line buffer content and the LUT content. The
 * results are placed into "lines" and "scr_fill". The function runs until all
 * the elements of "lines" are filled. When the routine is entered with a
 * lc_image it places "lowest" to the aoi in CAD_DATA so that (lc_image >=
 * lowest->iy && lc_image < lowest->iy + lowest->l) || lowest is the first ||
 * lowest is the last. When one of the two last state the "lines[lc_image]" is
 * filled with zero's. When the first state "highest" is searched as the first
 * aoi so that lc_image < highest->iy. For every x on the y line "lc_image",
 * every aoi between [lowest .. highest> is tested. If the coordinate is in an
aoi the
 * address of "address" is assigned to the correct position in the line buffer
 * and to the correct position in scr_fill. Problems are:
 * - the block is full and the line is not finished. Search in the previous
 * line buffer the last not zero address and set the interrupt bit. Renumber
 * the current line buffer so that the new block starts at address 1 an do the
 * same for the scr_fill.
 * - the position in which the address must be places in scr_fill is not free.
 * Print the oldest line in scr_fill and assign the emptied line to
 * the needed line of scr_fill.
*/
  {
  struct aoi_type *lowest, *highest, *loop, *loop2;
  int i, x, k, l, m, n, prev, placed;

  lowest = (struct aoi_type *)CAD_DATA; /*start at beginning of the cad data*/
  while ( (out1!=in1)||(lc_image==0) ) /*run when there is place in buffer*/
    {
    /* search lowest */
    for ( ; (lc_image > (lowest->iy+lowest->l-1)) && \
                              (lowest < &CAD_DATA[CAD_LENGTH]); lowest++ );

    if ( (lc_image < lowest->iy) || (lc_image > (lowest->iy + lowest->l-1)) )
                                                  /* start up or end*/
      { /* send a linebuffer with zeros when the first aoi starts later or
                                                      when the highest aoi is past */
      for ( i=0 ; i < units_per_line; i++)
        lines[in1][i] = 0;
      }
    else
      {
      /* search highest */
      for (highest = lowest+1; (lc_image >= highest->iy) && \
                          (highest < &(CAD_DATA[CAD_LENGTH+1])) ; highest++);

      placed = 0;
      for ( x=0 ; x < units_per_line; x++)
        {                 /* for every x on y line "lc_image" */
        loop = lowest;
        do      /* search in all aoi between [lowest..highest> if */
          {                                   /*   the coordinate is in a aoi */
          if ( (lc_image >= loop->iy) && (lc_image < (loop->iy+loop->l)) && \
                                (x >= loop->ix) && (x < (loop->ix+loop->w)) )
            {  /* place number */
            if (address == units_per_block)
              {   /* block is full !!! */
              if (in1 == 0) prev = BUF_NUM - 1;/* take previous linebuffer */
              else prev = in1 - 1;
```

```
            /* search last unit in previous array or first element
                                                    for the interrupt bit*/
            for (k=units_per_line-1;(lines[prev][k] == 0) && (k!=0); k--);
            lines[prev][k]=lines[prev][k]+2048;/*set bit11 the interrupt
bit*/
            address = 1;                        /* reset address */
            block = (block+0x1000) & 0x7000; /*increment block number mod
8*/
            for ( l=0; l < x; l++)
                /* renumber the placed units in lines[inl] and in scr_fill*/
              if ( lines[inl][l] != 0 )
                {    /* there is something to renumber */
                lines[inl][l] = address;    /* renumber the linebuffer */
                for ( loop2 = lowest; loop2 < highest; loop2++)
                  {
                  if ( (l >= loop2->ix) && (l < (loop2->ix + loop2->w)) &&\
                      (lc_image >= loop2->iy) && (lc_image < (loop2->iy + \
                                                          loop2->l)) )
                        /* renumber scr_fill */
                        scr_fill[ loop2->sy + (lc_image - loop2->iy) ]\
                            [ loop2->sx + l - loop2->ix ] = address + block;
                  else if ((loop2->iy == (highest-1)->iy) && ((l<loop2->ix)\

                           || ( l >= ((highest-1)->ix + (highest-1)->w) ) )
)
/* The aoi list is sorted first on the iy coordinate then on the ix
coordinate.
   So when the loop2->iy lies on the same height as the one before highest the

   ix coordinate of the one between them lies between the ix of those two. It
   is not necessary to continue searching when l (the x coordinate) is smaller

   then loop2->ix or when l is to the right of the last aoi on that line. */
                    break;
                  }
                address++;
                };
            };

          lines[inl][x] = address;    /* place the address */
          placed = 1;
          if (scr_fill[loop->sy+lc_image-loop->iy] == NULL)
            {       /* a buffer must be reserved */
            for(m=0; (m<SCR_BUF) && (used[m] != -1) ;m++);/*search a
buffer*/
            if (m == SCR_BUF)
              { /* all buffer used */
              out_screen(fpout, fpouts, 0, unitsize);
              for( n = 0; (n < SCR_BUF) && (used[n] != lc_screen); n++);
                              /* search which array of screen was used */
              used[n] = loop->sy+lc_image-loop->iy;
              scr_fill[loop->sy+lc_image-loop->iy] = scr_fill[lc_screen];
              scr_fill[lc_screen] = NULL;
                }
              else
                { /* a free buffer found */
                used[m]=loop->sy+lc_image-loop->iy;
                scr_fill[ used[m] ] = screen[m];
                }
              };
            scr_fill[ loop->sy+lc_image-loop->iy ][ loop->sx+x-loop->ix ] =\
                                                        address+block;
            }
          else if ( (loop->iy == (highest-1)->iy) && ( (x < loop->ix) || \
                                      ( x >= ((highest-1)->ix + (highest-1)->w))) )
/* The aoi list is sorted first on the iy coordinate then on the ix
coordinate.
   So when the loop->iy lies on the same height as the one before highest the
   ix coordinate of the one between them lies between the x of those two. It
   is not necessary to continue searching when x (the x coordinate) is smaller

   then loop->ix or when x is to the right of the last aoi on that line. */
            break;
          loop++;
          }
        while (loop < highest);
        if (placed)
          {
          address++;
          placed = 0;
          }
        else
          lines[inl][x] = 0;
        }
      }
     inl = (inl+1)%BUF_NUM;
     lc_image++;
     }
  }

void show_lines(fpo, lpi, upl)
  FILE *fpo;
  int lpi, upl; /* lines per image,  units per line */
/*
 * "show_lines" outputs linebuffers from "lines" so that on line is left
 * except when the is nothing to compute.
*/
  {

  while ( ((out1+1)%BUF_NUM) != inl )
    out_line( fpo , upl);

  if ( lc_image == lpi )
    { /* print the last two lines */
    out_line( fpo, upl);
    out_line( fpo, upl );
```

```
    };
  }

void out_line( fpo, upl )
  FILE *fpo;
  int upl; /* units per line */
/*
 * "out_line" prints one line of the array "lines".
*/
  {
  int i;

  fprintf(fpo, "line %4d: ",lc_out);
  for ( i=100; i< 260 /*upl*/; i++)
     fprintf(fpo,"%4d", lines[out1][i]);
  fprintf(fpo, "\n");
  lc_out++;
  out1= (out1+1)%BUF_NUM;
  }

void out_screen( fpout, fpout2, last, unitsize)
  FILE *fpout, *fpout2;
  int last, unitsize;
/*
 * out_screen outputs one or more lines until one line is cleared in "screen"
 * when "last" is zero. When "last" is not zero all lines are printed (the
 * rest of the actual screen and the next screen when there is one).
*/
  {
  while ( ( (scr_fill[lc_screen] == NULL) && !last) ||
          ( (lc_screen < 127) && last) )
    {
    lc_screen++;
    if ( (lc_screen >= SCR_WIDTH) && !last )
      {
      lc_screen = 0;
      fprintf(fpout, "\r\nScreen %d.\r\n", screen_nr);
      screen_nr++;
      };
    if (scr_fill[lc_screen] != NULL)
      fill_lut_line(fpout, fpout2, unitsize);
    else
      fill_lut_zero(fpout);
    fprintf(fpout, "\r\n");
    }

  if (last && (scr_fill[0] != NULL) )
    {
    fprintf(fpout, "\r\nScreen %d.\r\n", screen_nr);
    screen_nr++;
    for (lc_screen = 0; lc_screen < SCR_WIDTH; lc_screen++)
      {
      if (scr_fill[lc_screen] == NULL)
        fill_lut_zero(fpout);
      else
        fill_lut_line(fpout, fpout2, unitsize);
      fprintf(fpout, "\r\n");
      }
    };
  }

void fill_lut_zero( fpout )
  FILE *fpout;
/*
 * The line "lc_screen" in the LUT is filled with zeros.
*/
  {
  int i;

  fprintf(fpout, "line %3d: ", lc_screen);
  for ( i = 0; i < SCR_WIDTH ; i++)
        fprintf(fpout, "0");
  };

void fill_lut_line( fpout, fpouts , us)
  FILE *fpout, *fpouts;
  int us; /* unitsize */
/*
 * The line "lc_screen" of the LUT is filled with "scr_fill[lc_screen]".
 * The array scr_fill[lc_screen] is also cleared. The switch list is
 * calculated and the switch bit is set when necessary. The switch bit is set
 * when the block number of the actual unit is different than that of the
 * "last_block". The interrupt bit is then set on position "last_pos". Because
 * the LUT is write only the value of that last position is kept in
 * "last_val". The switch list "switch_buf" is initially filled with zero on
 * the first position which means that the first block for the units is the
 * zero block. When in the same line one or more block switches occur the new
 * block number are put in the switch array. When the next line is entered the
 * switch array is printed unitsize times and the new block number is put in
 * the array. The array is also printed unitsize times when there is only one
 * element in the array but the block changes when the next lines comes.
*/
  {
  int i, j, k;

  fprintf(fpout, "line %3d: ", lc_screen);
  for ( i = 0; i < SCR_WIDTH ; i++)
    {
    if ( scr_fill[lc_screen][i] != 0)
      {  /* check for block difference */
      fprintf(fpout, "%4d", /*((scr_fill[lc_screen][i]&0x7000)>>12)+'a',*/\
                                          scr_fill[lc_screen][i]&0xfff);
      if ( ( (((scr_fill[lc_screen][i]&0x7000)>>12) != last_block) || \
                 switch_index>1) ) && (lc_screen != (last_pos&0xff80)>>7 ) )
        {
        fprintf(fpouts, "llutll bit is set. Position lc_screen %d, x %d,\
```

```
            alue %d.\r\n", (last_pos&0xff80)>>7, last_pos&0x7f, last_val);
        for ( j = 0; j < us; j++)
          for ( k = 0; k < switch_index; k++)
            fprintf(fpouts, "%d, ", switch_buf[k]);
        fprintf(fpouts, "\r\n");
        switch_buf[0] = ((scr_fill[lc_screen][i]&0x7000)>>12);
        switch_index = 1;
        last_block =  switch_buf[0];
        }
      else if ( (((scr_fill[lc_screen][i]&0x7000)>>12) != last_block) && \
                                      (lc_screen == (last_pos&0xff80)>>7) )
        {
        fprintf(fpouts, "llutli bit is set. Position lc_screen %d, x %d, \
              value %d.\r\n", (last_pos&0xff80)>>7, last_pos&0x7f, last_val);
        switch_buf[switch_index] = ((scr_fill[lc_screen][i]&0x7000)>>12);
        last_block =  switch_buf[switch_index];
        switch_index++;
        };

      last_pos = lc_screen*SCR_WIDTH+i;
      last_val = scr_fill[lc_screen][i]&0xfff;
      scr_fill[lc_screen][i] = 0;
      }
    else
      fprintf(fpout, "   0" );
    }
}
```

Appendix E Assembly listings fill_ua_buffer

E.1 Original version

```
!Line#  Source Line   Flame Computer Corp. Sparc-C Compiler Version 3.6.11  May 24 09:00:17 1993

        TITLE   C:\SC\SRC\C\fill.c
        NAME    fill

!*** 1  #define LINE_LENGTH                     3000
!*** 2  #define BUFFER_SIZE                        4
!*** 3  #define LINE_BUFFER_BASE_ADDRESS    0x090002
!*** 4  extern int current_image_line;
!*** 5  extern int image_unit_addresses[BUFFER_SIZE][LINE_LENGTH];
!*** 6
!*** 7  void fill_ua_buffer ( )
!*** 8      {
!*** 9      int i;
_fill_ua_buffer:
00000000  9DE3BFF0      save    %o6, -16, %o6
!*** 10      short *linebuffer;
!*** 11
!*** 12      linebuffer = LINE_BUFFER_BASE_ADDRESS;
00000004  37000240      sethi   %hi(589826), %i3
00000008  B616E002      or      %i3, 2, %i3
0000000C  B210001B      mov     %i3, %i1
!*** 13      for( i = 0; i < LINE_LENGTH; i++ )
00000010  B0102000      mov     0, %i0
00000014  80A62BB8      cmp     %i0, 3000
00000018  36800000      bge,a   IB3
0000001C  01000000      nop
!*** 14          *(linebuffer + 4*i)= image_unit_addresses[current_image_line][i];
00000020  B6102004      mov     4, %i3
                    IB1:
00000024  84100018      mov     %i0, %g2
00000028  40000000      call    __mul
0000002C  8210001B      mov     %i3, %g1
00000030  B6100001      mov     %g1, %i3
00000034  B406401B      add     %i1, %i3, %i2
00000038  37000000      sethi   %hi(_image_unit_addresses), %i3
0000003C  B616E000      or      %i3, %lo(_image_unit_addresses), %i3
00000040  1F000000      sethi   %hi(_current_image_line), %o7
00000044  D003E000      ld      %o7, %lo(_current_image_line), %o0
00000048  40000000      call    __mul
0000004C  92102EE0      mov     12000, %o1
00000050  B8100008      mov     %o0, %i4
00000054  B606C01C      add     %i3, %i4, %i3
00000058  B92E2002      sll     %i0, 2, %i4
0000005C  F606C01C      ld      %i3, %i4, %i3
00000060  F6368000      sth     %i3, %i2, %g0
!*** 15      }
00000064  B0062001      inc     1, %i0
                    IB2:
00000068  80A62BB8      cmp     %i0, 3000
0000006C  26800000      bl,a    IB1
00000070  B6102004      mov     4, %i3
                    IB3:
00000074  81C7E008      ret
00000078  81E80000      restore %g0, %g0, %g0

_fill_ua_buffer  Local Symbols

Name                    Class   Type            Size   Offset  Location

i. . . . . .  reg     int                                    i0
linebuffer . . . . . .  reg    *short                          i1
```

E.2 Improved version

```
!Line#  Source Line   Flame Computer Corp. Sparc-C Compiler Version 3.6.11  May 24 10:05:23 1993

        TITLE   C:\SC\SRC\C\fill2.c
        NAME    fill2

!*** 1  #define LINE_LENGTH                     3000
!*** 2  #define BUFFER_SIZE                       4
!*** 3  #define LINE_BUFFER_BASE_ADDRESS    0x090002
!*** 4  extern int current_image_line;
!*** 5  extern int image_unit_addresses[BUFFER_SIZE][LINE_LENGTH];
!*** 6
!*** 7  void fill()
!*** 8      {
!*** 9      full_ua_buffer(&(image_unit_addresses[current_image_line][0]));
00000000  9DE3BFF0      save    %o6, -16, %o6
00000004  31000000      sethi   %hi(_image_unit_addresses), %i0
00000008  B0162000      or      %i0, %lo(_image_unit_addresses), %i0
0000000C  1F000000      sethi   %hi(_current_image_line), %o7
00000010  D003E000      ld      %o7, %lo(_current_image_line), %o0
00000014  40000000      call    __mul
00000018  92102EE0      mov     12000, %o1
0000001C  B2100008      mov     %o0, %i1
00000020  40000000      call    _full_ua_buffer
00000024  90060008      add     %i0, %o0, %o0
!*** 10     }
!*** 11
!*** 12 void fill_ua_buffer ( image_1 )
00000028  81C7E008      ret
0000002C  81E80000      restore %g0, %g0, %g0
!*** 13 int image_1[LINE_LENGTH];
!*** 14     {
_fill_ua_buffer:
00000030  9DE3BFF0      save    %o6, -16, %o6
!*** 15     int i;
!*** 16     short *linebuffer;
!*** 17
!*** 18     linebuffer = LINE_BUFFER_BASE_ADDRESS;
00000034  39000240      sethi   %hi(589826), %i4
00000038  B8172002      or      %i4, 2, %i4
0000003C  B410001C      mov     %i4, %i2
!*** 19     for( i = 0; i < LINE_LENGTH; i++ )
00000040  80A66BB8      cmp     %i1, 3000
00000044  36800000      bge,a   IB3
00000048  01000000      nop
!*** 20         {
0000004C  BB2E6002      sll     %i1, 2, %i5
                  IB1:
!*** 21         *(linebuffer) = image_1[i];
00000050  F806001D      ld      %i0, %i5, %i4
00000054  F8368000      sth     %i4, %i2, %g0
!*** 22         linebuffer += 4;
00000058  B406A004      add     %i2, 4, %i2
!*** 23         }
!*** 24     }
0000005C  B2066001      inc     1, %i1
                  IB2:
00000060  80A66BB8      cmp     %i1, 3000
00000064  26800000      bl,a    IB1
00000068  BB2E6002      sll     %i1, 2, %i5
                  IB3:
0000006C  81C7E008      ret
00000070  81E80000      restore %g0, %g0, %g0

_fill_ua_buffer  Local Symbols

Name                    Class   Type            Size   Offset  Location

image_1. . . . . .      reg     *int/array      3000           i0
i. . . . . .            reg      int                           i1
linebuffer . . . . . .  reg     *short                         i2
```

Annelies Meerbach	EH 1.10
Anne-Marie v. Helvoort	EH 1.07
Yvonne v. Bokhoven	EH 1.09
Leni v.d. Zanden	EH 1.06
Marja de Mol	EH 1.26
Joke Verhoef	EMV - EL 1.15
Majoke Velberg	EEG 1.13
Mariet van Rixtel	EEG 2.06
Els Gerritsen	EH 2.28
Joyce Bagchus	EH 2.06
Peter v.d. Ven	EH 2.06
Piet de Greef	EH 1.26
Tiny Verhoeven	EH 6.35
Tiny Bijl	EH 8.34
Linda Balvers	EH 9.34
Lia de Jong	EH 7.33
Rian v. Gaalen	EH 10.28
Doret Pellegrino	EH 12.28
Tanja de Haan	BG 2.14
Linda v. Loon	
Yvonne Broers	EH 3.05
Paul v. Eck	EH 1,21
Piet Bell	EH loge
Jan Bell	"
John Snoeijs	"
Louise Schavet	HG 0.08
Cliff Hasham	CTD - RF HG -1.11
Albert Nelissen	CTD - RF hg -1.11
Niels Olthuis	WFW 2.136
Arthur Thepass	Magazijn elektro EH 0.01
C.H.J.D. v. Rijsewijk	Magazijn elektro EH 0.01
A.F. Chamboné	E - EB EH 10.15
ing. Alex Wijffels	E - EB EH 10.08
dr.ir. T. Kwaaitaal	EH 2.27
dr.ir. E. Wezenbeek	IPL - TNO Kamer 53
Ton v.d. Boom	Vakgroep Regeltechniek, Gebouw Elektrotechniek, Kamer 12.05, TU Delft, Postbus 5031, 2600 GA Delft
Yucai & Ge	Hageheldlaan 62, 5641 GP Eindhoven
Prof. C. Mulders	Herman Gorterlaan 319, 5644 SN Eindhoven
Dhr. C. Huber	Sleedoorn 9, 5666 AT Geldrop
Alberto Martis	Prof. v.d. Grintenlaan 32, 5652 NB Eindhoven
Arendsz, A.	Mathildelaan 75, 5611, BE Eindhoven (PTH-er)
dr.ir. B. den Brinker	ESP - EH 5.27
Peter Janssen	Furkabaan 26, 3524 ZJ Utrecht
Roy Simon	Voltairestraat 60, 3076 TP Rotterdam (broer)
Rudolph	Gentiaanstraat 592, 7322 CL Apeldoorn
Alex Riley	
Julio de la Court	
Margarita	
Bode	
Dhr. H. v.d. Heiden	BDK Pav. F.05
Centrale Stud. Administratie	

Annelies Meerbach	EH 1.10
Anne-Marie v. Helvoort	EH 1.07
Yvonne v. Bokhoven	EH 1.09
Leni v.d. Zanden	EH 1.06
Marja de Mol	EH 1.26
Joke Verhoef	EMV - EL 1.15
Majoke Velberg	EEG 1.13
Mariet van Rixtel	EEG 2.06
Els Gerritsen	EH 2.28
Joyce Bagchus	EH 2.06
Peter v.d. Ven	EH 2.06
Piet de Greef	EH 1.26
Tiny Verhoeven	EH 6.35
Tiny Bijl	EH 8.34
Linda Balvers	EH 9.34
Lia de Jong	EH 7.33
Rian v. Gaalen	EH 10.28
Doret Pellegrino	EH 12.28
Tanja de Haan	BG 2.14
Linda v. Loon	
Yvonne Broers	EH 3.05
Paul v. Eck	EH 1,21
Piet Bell	EH loge
Jan Bell	"
John Snoeijs	"
Louise Schavet	HG 0.08
Cliff Hasham	CTD - RF HG -1.11
Albert Nelissen	CTD - RF hg -1.11
Niels Olthuis	WFW 2.136
Arthur Thepass	Magazijn elektro EH 0.01
C.H.J.D. v. Rijsewijk	Magazijn elektro EH 0.01
A.F. Chamboné	E - EB EH 10.15
ing. Alex Wijffels	E - EB EH 10.08
dr.ir. T. Kwaaitaal	EH 2.27
dr.ir. E. Wezenbeek	IPL - TNO Kamer 53
Ton v.d. Boom	Vakgroep Regeltechniek, Gebouw Elektrotechniek, Kamer 12.05, TU Delft, Postbus 5031, 2600 GA Delft
Yucai & Ge	Hageheldlaan 62, 5641 GP Eindhoven
Prof. C. Mulders	Herman Gorterlaan 319, 5644 SN Eindhoven
Dhr. C. Huber	Sleedoorn 9, 5666 AT Geldrop
Alberto Martis	Prof. v.d. Grintenlaan 32, 5652 NB Eindhoven
Arendsz, A.	Mathildelaan 75, 5611, BE Eindhoven (PTH-er)
dr.ir. B. den Brinker	ESP - EH 5.27
Peter Janssen	Furkabaan 26, 3524 ZJ Utrecht
Roy Simon	Voltairestraat 60, 3076 TP Rotterdam (broer)
Rudolph	Gentiaanstraat 592, 7322 CL Apeldoorn
Alex Riley	
Julio de la Court	
Margarita	
Bode	
Dhr. H. v.d. Heiden	BDK Pav. F.05
Centrale Stud. Administratie	