5534

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
DESIGN AUTOMATION GROUP

THE BLITTER PROJECT,

A TEST CASE.

W.J.M. Philipsen

Master thesis
reporting on graduation work
performed from 01.12.87 to 24.08.88
by order of prof. dr. ing. J.A.G. Jess
and supervised by ir. G.L.J.M. Janssen

ABSTRACT

This report present a test-case for the design tools present at the Design Automation Group. We designed a blitter, graphic processor, with these tools, and evaluated the use and the results obtained with them. Blitter stands for Bit BLock Image TransfERrer. It can move large blocks of bitplane image data from one location in memory to an other. We give a description of the blitter, and a description of the design process. The blitter is not yet finished, but most parts are ready.

In the second part, there is a description of each tool and a report of the problems we encountered, designing the blitter. It appeared that parts of the blitter were hard examples. We discovered some bugs in the tools. But most of the problems have been solved.

CONTENTS

LIST OF FIGURES

1. Introduction

Vast developments in the Integrated Circuit technology, increased the need for Computer Aided Design tools. In the Design Automation Group of the department of Electrical Engineering of Eindhoven University of Technology several tools have been developed. The goal of this project was to test these tools, by designing a chip with them. The chip to be designed was a blitter, a graphics processor. The specifications of an existing blitter have been used to design our own chip.

This report consists of two parts. In the first there is a description of the blitter, and its design. In the second part. We alloted each tool a section

2. The Blitter

In this chapter we discuss the blitter. After a short description of a blitter, and a bird's-eye view of the tools used for the design, there is a description of the design process of the blitter.

2.1 A short description of the bit blitter.

Blitter is an abbreviation for *b*lock *i*mage *t*ransfer*r*er. It is a graphics processor whose main application is movement of large blocks of bitplane data. It can perform such operations, after a set-up of its registers, considerably faster than an ordinary processor. It includes features to facilitate copying and processing of "rectangular" regions of memory. Typically, these regions are areas within graphics images. The process of performing a blitter operation is also called a blit.

The blitter uses up to four DMA (Direct Memory Access) channels. Of the four DMA channels, three are dedicated to retrieving data from memory to the blitter. These are known as source A, source B and source C. The one destination DMA channel is designated source D. It is not always necessary to use all the channels. Each channel may independently be enabled. All three sources are fetched from memory in a pipelined fashion and held in registers for logic combination before being send to destination.

Figure 1 shows the DMA-system. The DMA-controller distributes the memory cycles between the blitter and the processor. If the blitter uses the memory while the processor works up it is last instruction, it doesn't hold up the processor. The 68000 processor has been the example when processor dependent features had to be defined.

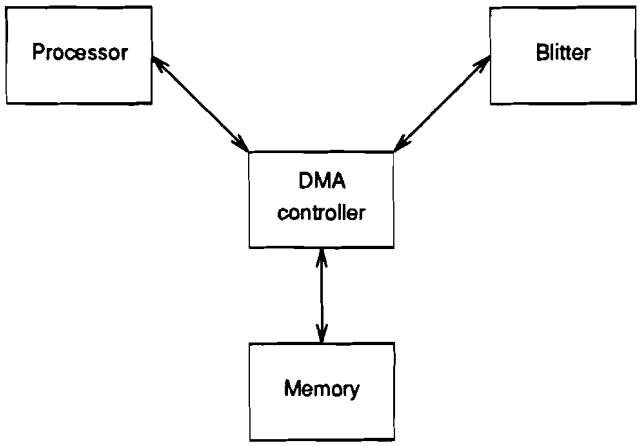

Figure 1. The Direct Memory Access system.

A summary of the blitter features and operations:

- *Data copying*. The blitter can copy bit-plane image data, from one location to an other.
- *Multiple pointers and modulos*. Each channel has it is own pointer and modulo registers. This allows the blitter to operate upon identical windows within larger images, with different sizes.
- *Ascending and descending addressing*. The blitter can address in two directions, it can either start at the bottom or at the top of the window.

- *Logic Operations*. The blitter can perform a logic operation upon the data of the three sources before transferring the result to the target. Before a blit is started, the blitter is set up to perform one of the logic operations on the three data sources when preparing the output.
- *Shifting*. The blitter can shift one or two of its data sources up to 15 bits before applying the logic operation. This is necessary when you want to move images across word boundaries.
- *Masking*. The blitter can mask the leftmost and the rightmost data word from each horizontal line of a window.
- *Area-filling*. The blitter can perform hardware-assisted area fill between predrawn lines.

2.2 The design path.

In this section we will give an overview of the design system. To comprehend the discussion about the blitter, it is necessary to know the design system in general. A thorough description of each tool can be found in part two of this report.

This chip has been designed for the nmos process available at EFFIC (Eindhoven Fabrication Facility for Integrated Circuits), using the 6μ design rules.

We started with a thorough description of the functionality of the blitter in plain English, with the description of the blitter in the Commodore Amiga personal computer[1] as an example. This description in words was refined by writing a computer program, with the same functionality as the blitter, but that writes it output to a terminal. The program was written in "C". We tried to make the C description as close as possible to the functionality of the blitter. Non blitter functions, as writing to the screen, were placed in separate functions.

Using this formal description of the blitter, it was divided into several high level blocks. For drawing the pictures we used ESCHER (Eindhoven SCHematic EntRy). With the ESCHER+ simulator, an extension of ESCHER, it is possible to add to each module a description of its behaviour, and to simulate the resulting design. With these high level simulations one already encounters specific hardware problems, as for example the communication between the different parts.

Stepwise refinement of the schematic with ESCHER, dividing large modules into smaller ones, finally leads to two basic kind of modules: controllers and others. For the controller we made state machine descriptions, using the behaviour descriptions from the ESCHER+ simulation. With ESKISS the controllers have been converted into modules of the second kind. ESKISS computes a state encoding for the state machine. The output is a boolean specification of the controller. For the other modules we manually made boolean descriptions.

These logic specifications were optimized with EUCLID. The optimized logic specifications were prepared with the technology mapper for the pluri-cell generator. The output of the technology mapper is a gate file. The cell generator generates layout for these cells. and a netlist with the connections between the cells.

From the gatefile a SLS description can be made with the tool "log_sls". SLS is an abbreviation of Switched Level Simulator, a logic simulator. These simulations proved to be useful, design errors could be found in a very early stage.

The cells obtained with the cell generator were placed and routed with MACPLACE and ROCOCO. The result is a pluri-cell layout. The cells are placed in columns, and between the columns there are the routing channels. MACPLACE computes a placement for the cells, and the connections were made by ROCOCO.

With the floorplanner, we made a floorplan for these modules, using the network we made with ESCHER. The floorplan also was routed with ROCOCO.

From this final layout the circuit could be extracted again, and simulated with SLS.

2.3 The Blitter Features.

2.3.1 Data copying.

The most important function of a blitter is copying large blocks of image data from one location in memory to an other. Bitplane images are usually stored in a linear way in memory. Each word is stored at the address of it is left neighbour plus one. And the first word of a line is stored at the address of the last word of the previous line plus one.

Figure 2 shows an example of a representation of a bit-plane. Each address accesses one 16 bit word. The blitter needs only to know the starting point, the width and the height (in the example 10, 7 and 5). After the processor has loaded the registers of the blitter, the blitter performs the transfer independently of the processor. To get access to the memory it claims DMA cycles from the DMA manager. When it has finished the blit operation the blitter signals this to the processor by setting an interrupt flag.

10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	32	33	34	35	36	37
38	39	40	41	42	43	44

Figure 2. The addresses of an image in memory.

2.3.2 Pointers and modulos.

In a pointer register the blitter stores the address of the next data word to fetch from memory. The 19-bit addresses are divided in two parts. The upper 3 bits are stored in the PTH register, and the lower 16 in the PTL register. In most systems, the memory will be divided in bytes, although the processor uses word (one word is two bytes), addressing. For this reason the least significant bit will always be zero, and is in general not implemented.

Our 19 bit address bus enables our blitter to address the lower 512k word (= 1024 kbyte) of memory, twice as much as the amount of the Amiga blitter.

Because each channel has its own pointer and modulo registers, each channel can address a bitplane with different sizes, and at different locations.

When the blitter has to perform an operation on a part of an image, the blitter uses the modulo registers. The modulo is the difference of the width of the larger image and the smaller window, that the blitter should operate upon. The modulo is added to address, at the end of each line. Because each channel has its own modulo register, each channel may address a window within a larger bit-plane, with different sizes.

10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	32	33	34	35	36	37
38	39	40	41	42	43	44

Figure 3. A window within a larger image.

Figure 3 shows an example of an image larger as the window used. To operate upon the smaller window only, the address sequence must be as follows:

19, 20, 21, 26, 27, 28, 33, 34 ,35

This requires a normal increment of two each time, and at the end of each window line the additional jump value, the modulo, to bring the pointer to the start of the next window-line. The module is 8 in this case, so the original width of the image was 7 words.

2.3.3 Ascending and descending addressing.

It is important to be able to control the direction of addressing, when source and destination areas overlap. When you want to move the data to a lower address in memory, you use ascending, and and when you want to move the data to a higher address you use descending addressing. Otherwise it is possible that the blitter writes to an address that is not yet read. With the fill operation only descending addressing is available.

Also with shifting the direction of addressing is important, because a certain direction of addressing implies a certain direction for shifting. The addressing direction is controlled by the bit desc in the CON1 register.

2.3.4 Shifting.

In order to be able to shift an image any number of bits, and not just multiples of 16, there is a separate shift facility, to shift words across word boundaries.

The shifter has the two last read words for one channel as input. These two words are put in one 32 bit wide bit vector. For ascending addressing the oldest word is put at the most significant places. The output will be bits (16-sh) through (31-sh). Sh is the number of bit to be shifted.

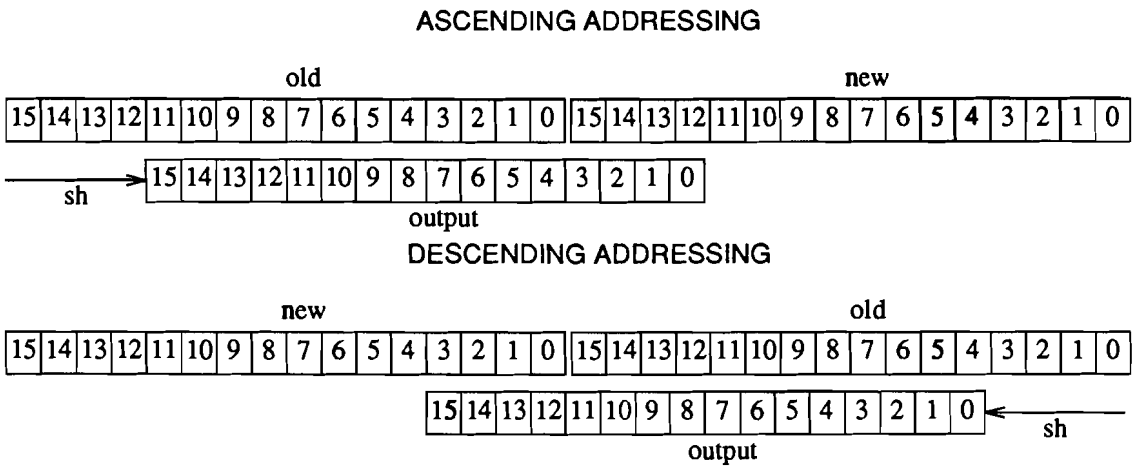

Figure 4. example of shifting.

When descending addressing is used, the words are placed the other way around in the vector. In this way we will always have the word with the lowest address at the most significant place. In this case the output is bits sh through (15+sh). Figure 4 shows a 4 bit shifting operation for both descending and ascending addressing.

2.3.5 Logic operations.

Three sources are available to the blitter logic unit. These sources are usually one bit-plane from each of three separate graphics images. While each of these sources is a rectangular region composed of many points, the same logic function is performed to each point throughout the rectangular region.

The logic function performed on each point is chosen by the LF control byte in the BLTCON0 register. For each bit all possible minterms (8) are constructed. Each bit in the LF control byte enables one minterm. Table 1 shows them. This gives 256 logic functions.

enable bits	7	6	5	4	3	2	1	0
minterms	ABC	$AB\bar{C}$	$A\bar{B}C$	$A\bar{B}\bar{C}$	$\bar{A}BC$	$\bar{A}B\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}\bar{B}\bar{C}$

Figure 5. The minterms selected by LF control.

As an example I will derive the value for LF for the "cookie-cut" operator. The formula for the function is:

$D=AB+\bar{A}C$

This is equal to:

$D=ABC+AB\bar{C}+\bar{A}BC+\bar{A}\bar{B}C$

Thus bits 7, 6, 3 and 2 should be high in LF to select the "cookie-cut". In [1] an other method can be found to calculate the value for the logic function, a more confusing method.

2.3.6 Masking.

All blitter operations are done upon words. To do operations on windows with a boundarys within a word, the masking facility is available. Two masks can be defined. One the FWM (First Word Mask) will be laid on the first word of each line. If a mask is laid on a word, all bits of the word, whose corresponding bit in the data word is zero, will be treated as is they were zero's during further processing. The other mask, LWM the Last Word Mask, will be apllied on the last word. Figure 6 gives an example.

input word	1110011110000111
mask	0000000111111111
result	0000000110000111

Figure 6. An example of masking.

2.3.7 Area filling.

The blitter can perform a hardware assisted area fill, between predrawn lines. It scans each word from right to left for bit that are one. If it finds a one, it inverts its fill state. The output bit is always the fill state. There are two filling modes:

- Inclusive fill
- Exclusive fill

If the input bit is one and the fill state changes from zero to one, the output bit will be zero in the exclusive fill mode, and one in the inclusive fill mode. In other cases, the output will always be equal to the filling state. The filling modes are enabled respectively with the bits IFE and EFE in the register CON1. The user has to take care that only one of the modes is enabled.

The initial fill state, at the beginning of each line is equal to the bit FCI in CON1. Within a line the fill state is passed as a kind of carry bit from one word to an other. The filling is done after the logic operation has been applied upon the data.

Because words are scanned from right to left, filling can only be used in the descending addressing mode.

Figure 7. An example of the filling facility.

2.4 A formal description of the blitter.

For further design we need a more precise description of the blitter. To get a formal description of the circuit we wrote a C program with same functionality as the blitter. I this way we got an exact description of what the blitter should do.

We will discuss only the major procedures of the program. The complete source text can be found in Appendix A.

2.4.1 The main loop.

After a reset the blitter waits until it is addressed to load one of its registers. This is the case if the pin ce (chip select) becomes high. The addressed register is loaded with the data on the databus, and the blitter starts scanning ce again. This cycle is repeated until the blitter register BLTSIZE is loaded. This register contains the size of the window, the blit has to be performed upon. If this register has been loaded the procedure blit is started to perform the blitter operation.

```
main ()
{
  while (TRUE)
    {
      reg_address = get( reg_addr_port );
      if ( get( chip_select )) /* wait until ce high */
       {
         data = get( data_bus );
         write_register( reg_address, data );
         if (reg_address == BLTSIZE)
           {
              blit();          /* start processing */
              put( interrupt , 1);
           }
       }
    }
}
```

Figure 8. The main loop.

After the blit has finished the blitter will start loading its registers again. Most of the registers will still contain their old value. It is possible and allowed to use them again, without reloading them.

2.4.2 The blitting part.

There are two loops. The first is done for each line, and the second for each word within a line. At the start of each line the data registers and the fill carry bit are reset. The first data words are read, and masking is applied upon the A source. The shifter has to read two words read from each channel, before the first output word can be calculated. The second for-loop is limited to have always two words in the pipeline.

After the last word has been read, there is one word left in the pipeline. It is shifted with the other word put to zero. The if statement is necessary to be able to handle windows that have a width of only one word.

Finally the pointers are adjusted to point to the next addresses. If the bit DESC in the register CON1 is high, the modulos are added, else they are subtracted. The procedures "get_data" and "put_data" take care of the additional decrease or increase after each word.

```
blit()
{
  WORD logop();
  int i, j, fco;

  for ( i = HEIGHT( bltsize ); i>0 ; i-- )
    {
      /* the start of a new line */
      /* reset registers         */
      bltAdat = 0;
      bltBdat = 0;
      bltCdat = 0;

      /* reset the fill carry in bit */
      fco = BIT( FCI, bltcon1 );

      /* get the first data word of this line */
      get_data();

      /* apply the mask on the A and B channel */
      bltAdat = bltAdat & bltAfwm;

      for ( j = WIDTH( bltsize ) - 2; j>0 ; j-- )
       {
         get_data();
         bltDdat = logop( SHIFT( bltAold, bltAdat,
                                 SH( bltcon0),
                                 BIT( DESC, bltcon1 )),
                          SHIFT( bltBold, bltBdat,
                                 SH( bltcon1 ),
                                 BIT( DESC, bltcon1 )),
                          bltCold, fco );
         put_data();
       }
```

```
        if (WIDTH( bltsize ) > 1)
         {
           get_data();
           bltAdat = bltAdat & bltAlwm;
           bltDdat = logop( SHIFT( bltAold, bltAdat,
                                   SH( bltcon0 ),
                                   BIT( DESC, bltcon1 )),
                            SHIFT( bltBold, bltBdat,
                                   SH( bltcon1 ),
                                   BIT( DESC, bltcon1 )),
                            bltCold, fco );
           put_data();
         }
        else
         bltAdat = bltAdat & bltAlwm;
        bltDdat = logop( SHIFT( bltAdat, 0,
                                SH( bltcon0 ),
                                BIT( DESC, bltcon1 )),
                         SHIFT( bltBdat, 0,
                                SH( bltcon1 ),
                                BIT( DESC, bltcon1 )),
                         bltCdat, fco );
        put_data();
        if (BIT( DESC, bltcon1 ))
         {
           bltApt -= bltAmod;
           bltBpt -= bltBmod;
           bltCpt -= bltCmod;
           bltDpt -= bltDmod;
         }
        else
         {
           bltApt += bltAmod;
           bltBpt += bltBmod;
           bltCpt += bltCmod;
           bltDpt += bltDmod;
         }
      }
}
```

Figure 9. The blitting part.

2.4.3 Logic operations

This procedure calculates all minterms, and uses them if the corresponding bit in the CON1 register is one. After that one of the two filling modes is applied, if necessary. Note that the words are scanned from right to left, thus filling only makes sense if using descending addressing.

```
  WORD logop( Adata, Bdata, Cdata, fco )
WORD Adata, Bdata, Cdata;
{
  WORD ddat;
  int  j;
  ddat = ( (BIT( 7, bltcon0 ) *  Adata &  Bdata &  Cdata ) |
         (BIT( 6, bltcon0 ) *  Adata &  Bdata & ~Cdata ) |
         (BIT( 5, bltcon0 ) *  Adata & ~Bdata &  Cdata ) |
         (BIT( 4, bltcon0 ) *  Adata & ~Bdata & ~Cdata ) |
         (BIT( 3, bltcon0 ) * ~Adata &  Bdata &  Cdata ) |
         (BIT( 2, bltcon0 ) * ~Adata &  Bdata & ~Cdata ) |
         (BIT( 1, bltcon0 ) * ~Adata & ~Bdata &  Cdata ) |
         (BIT( 0, bltcon0 ) * ~Adata & ~Bdata & ~Cdata ));
  if( BIT( IFE, bltcon1 ) | ( BIT( EFE, bltcon1 )))
    for( j=0; j<16; j++ )
      {
       fco = fco * BIT( j, ddat );
       if BIT( EFE, bltcon1 )
         {
           if ( fco == 1 )
             ddat = ddat | MASK( j );
           else
             ddat = ddat & ~MASK( j );
         }
       if BIT( IFE, bltcon1 )
         {
           if ( fco | BIT( j, ddat ))
             ddat = ddat | MASK( j );
           else
             ddat = ddat & ~MASK( j );
         }

      }
  return( ddat );
}
```

Figure 10. logic operations.

2.4.4 The Communication protocol.

There are three types of communication. First in the set-up phase the blitter reads the contents of the register bus and the databus, to get the register address and the value to be stored in the register. During the

blit, the blitter can read data from memory, or write data to memory.

Because there was no clear description of the protocols used by the Amiga blitter, we made some for our selves. We used for the program a protocol that was easy to implement in C. It is always possible to alter afterwards the protocol, because it is not a real part of the blitter. Any other protocol can be implemented without dramatic changes of the blitter.

2.4.4.1 Loading the registers.

The code for the first kind is included in the main loop, see section 2.4.1. The blitter keeps reading the reg_address port until it is chip select is high. At that moment it reads the word from the data bus, and transfers that word into the addressed register. It repeats this cycle until the register BLTSIZE is loaded with a value. When the processor has written a value to the BLTSIZE register the blitter starts the blit defined by the contents of it is registers. There is no acknowledgement from the blitter to the processor, but because the blitter is at that moment only reading, it will not be difficult to define a certain data valid period. If this solution is impossible it is always possible to use the request lines for validation.

2.4.4.2 Reading from memory.

This is the piece of code in the program that takes care of reading one word from the address stored in 'addr' in memory.

```
/* reading from memory */
{
  WORD getword( addr )
    ADDRESS addr;
  {
    put( data_bus_req, 1 );          /* request for cycle*/
    while( get( dma_req ) == 0 );  /* cycle available? */
    put( ram_write, 0 );           /* select read      */
    put( reg_addr_port, REG_ADDR( addr )); /* write address */
    put( ram_addr_port, RAM_ADDR( addr ));
    return( get( data_bus ));          /* read data       */
  }
}
```

Figure 11. Reading from memory.

When the blitter wants to read from memory, it puts the data bus request line high. After the dma_req line is pulled down, it puts the ram_write to 0 and loads the address into reg_addr_port and ram_addr_port. Then it reads the data_bus.Put and get do nothing but reading the value of the port addressed. So there is no check if the data available at the data bus is good or not, nor is there a signal to the processor to validate the address. This means that there has to be an exact timing schedule for reading and writing. Information concerning the speed of the memory has To be available when designing the timing for the blitter.

It is possible to use the dma request line and data bus request line for the validation. In this case the only time that the data is valid at the data bus should be defined. Then getword would become:

```
/* reading from memory with validation of the data */
{
WORD getword( addr )
ADDRESS addr;
{
    put( data_bus_req, 1 );          /* request for cycle*/
    while( get( dma_req ) == 0 );    /* cycle available? */
    put( ram_write, 0 );             /* select read      */
    put( reg_addr_port, REG_ADDR( addr ) ); /* write address    */
    put( ram_addr_port, RAM_ADDR( addr ) );
    put( data_bus_req, 0);                  /* addr valid       */
    while( get( dma_req ) == 1);            /* data valid?      */
    return( get( data_bus ));        /* read data        */
  }
```

Figure 12. Reading with validation.

2.4.4.3 Writing to memory.

There is only a small difference in the signals, ram_write becomes 1 and the data is put on the data bus, instead of read from it. Timing is the same, with the same remarks. Also for writing it is possible to use the existing request lines for validation.

```
/* writing to memory */
{
  putword( addr, data )
    ADDRESS addr;
  WORD    data;
  {
    put( data_bus_req, 1 );          /* request for cycle*/
    while( get( dma_req ) == 0 );    /* cycle available? */
    put( ram_write, 1 );                    /* select write     */
    put( reg_addr_port, REG_ADDR( addr ));/* write address    */
    put( ram_addr_port, RAM_ADDR( addr ));
    put( data_bus, data );                  /* write data       */
  }
```

Figure 13. Writing to memory.

2.5 Escher+ simulation.

2.5.1 The Escher+ scheme

For a part of the C-description an Escher+ simulation was made. At the time the simulations were done, it was not yet possible the use multiple levels. Each instance in the current template had to have its own behaviour description. Simulating the complete blitter would have led to very complicated behaviour descriptions. Therefore we simulated only a part of the blitter. Loading of the registers was left out in this simulation.

The display replaces the memory. All blitter operations are done upon a memory, with start address $00, and that contains 32 words. It represents a 8 lines high image, with 4 4-bit words in a line.

The blitter consists of two shifters, the masking hardware, logop (the logic operations block), some registers and a control unit. The next sections will give an explanation of the behaviour of the different blocks.

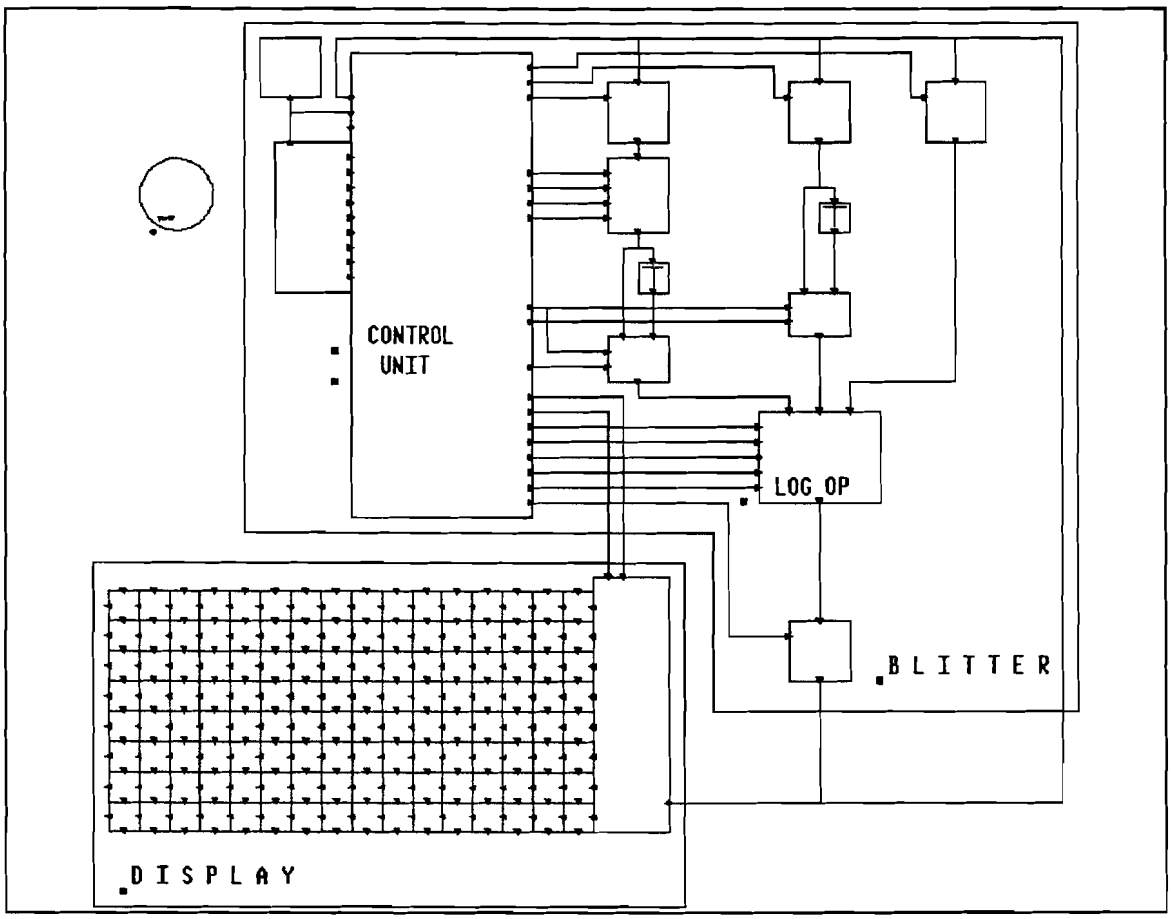

Figure 14. Escher+ simulation scheme.

2.5.2 The control unit.

2.5.2.1 The behaviour of cntr.

Most of the registers have been included in the control unit, and not been implemented as real registers, to simplify the design.

During the first simulation at simtime is 0, the initializations in lines 53 through 63 are evaluated. These are resets of control lines, the state register is set to zero and the interrupt line ready is set to one. The ready line has two functions. First it is connected to the interrupt block, it signals when the blitter operation is finished. Second, it is used to start the blitter again. By putting a one on this line during the initialization the blitter is prevented from starting a random blitter operation.

When all control inputs have their values the blitter can be started by setting the ready line to 0. Then the block of line 63 is evaluated. The control inputs are passed to their corresponding output lines. That are ife, efe, logfun, sha, shb, lwm and fwm. This block will be evaluated each new blitter operation. At the end of the block the local wait is set to zero, and trigger is triggered.

The local wait insures that only one block is evaluated each time the control unit is evaluated. With the delay of the trigger we can evaluate the unit, immediately or at a time in future, again, when it is ready for the next step. All statements are almost a one to one projection of the C-document in the Lisp-like code, with a state added when a delay was necessary, and at destinations of jump statements. Whenever possible we put the corresponding C statements as comment in the behaviour description.

The difference between byte_cntr and byte_cntr_desc is the last can also use descending addressing. When using this possibility the shift_desc and reg_load_desc should be used.

2.5.2.2 The behaviour of load_big.

This template just loads the control unit with new blitter operations, and starts it with setting the ready line to 0. The blitter should not be started at simtime is 0, because of the initialization in the control unit at that time. The user should use this block to edit the blitter operations.

2.5.3 Registers.

2.5.3.1 The behaviour of delay.

The delay templates are used to offer the shift templates both the old and the new data word of the A or B source. The contents of reg_new is transferred to reg_old, and reg_new is loaded within. Then the reg_old is transferred to the output.

2.5.3.2 The behaviour of dhold.

This is a special register that transfers the output of the logic unit to the data bus. It converts the 4-bit input bus into an output integer.

2.5.3.3 The behaviour of big_hold.

The input value is stored in reg. Then the right bit is delayed to the corresponding output line, and reg is shifted one bit right by dividing it by 2. This is done for every output line.

2.5.4 The behaviour of mask.

If the first word timer (fwt) is false, the registers are loaded with the input values. Otherwise the first word mask (fwm) is put on the input lines first. If the last word timer (lwt) is false, the registers are transferred to the out lines, when not the the last word mask (lwm) is put on it first.

2.5.5 Logop.

In lines 28 through 62, the value of each bit is computed from the three sources, and the logfun input lines. The results are stored in reg. The values stored in reg are transferred, direct or after further processing, depending on if one of the fill enable lines is high. For efe the carry, computed by taking the exclusive or of the bit and the old carry, taken as output. For inclusive fill the result of an or operation on the carry and bit. When filling the last carry is transferred to fco. It's the users responsibility that only one fill enable is high.

2.5.6 Shift.

The shift template gets two words as input, old and new, and the shift value sh. These two words are put into a bit-vector that is twice as long as a word, with the old word left. E.g. for a 4 bit word:

```
   old      new
3 2 1 0  3 2 1 0  <-- the words
7 6 5 4  3 2 1 0  <-- the bit-vector
```

When the addressing is ascending, thus desc = 0, the output has to be bit (7 - sh) through (4 - sh). This is done by setting offset to 4-sh, en the using bits offset through offset+3.

But when addressing is descending the words should be put the other way around. E.g.:

```
   new      old
3 2 1 0  3 2 1 0  <-- the words
7 6 5 4  3 2 1 0  <-- the bit-vector, the other way around.
```

We now have to take bits (3+sh) through sh. This means bits sh through 0 from new, and 3 through sh from old. That are bits sh through 0 and 7 through 4+sh from the old bit-vector. That is the same as 8+sh mod 8 through 4+sh mod 8. In the behaviour description this means setting the offset to 4+sh, and using bit i+offset mod 8. The modulo 8 has no influence when addressing the other way. When using an other word-length as 4, modulo (2*width) should be used, instead of the modulo 8.

2.5.7 Display.

2.5.7.1 The behaviour of mem_big.

The memory is divided in 8 rows of 4 words, that are 4 bits wide. R_w is the read/write control line. It should be 0 for reading, and 1 for writing. This line is also used to trigger this block. The control unit first set the right values for address and data, when writing, on the input lines, and then writes on the r_w line. Row0 through row7 are the rows. Each row is a four words bus. When reading, the wanted value is put on the data line. When writing, the value of the data line is stored in the right word. The right word is selected by a block of if else statements.

The behaviour of first_ and other_pix.

In other_pix in_0 through in_2 are put to the output. In_3 is divided by 2 and put to the output, after color is set to first bit of in_3.

In first_pix, the pixel at the right edge of a word, in_0 is passed to out_1, and so on, to have the right word in_3 in the next word.

It would be much nicer to be able to use a part of the pixels of the screen directly, but it is possible to work with this solution.

2.6 Final design.

For further design we will not use the same scheme as we used for the Escher+ simulations. In this case the blitter has been split up in 9 blocks. The address generator calculates at witch address the next data word is located for each channel. The logic unit takes care of preparing the data. It includes the masking hardware, shifting and the logic operations. Size control keeps track of the number of lines and words, and signal at the end of a line, and at the end of the window. In the set-up phase the decoder choses the destination for the data. And finally, four of the blitter registers, and the main controller for all other jobs.

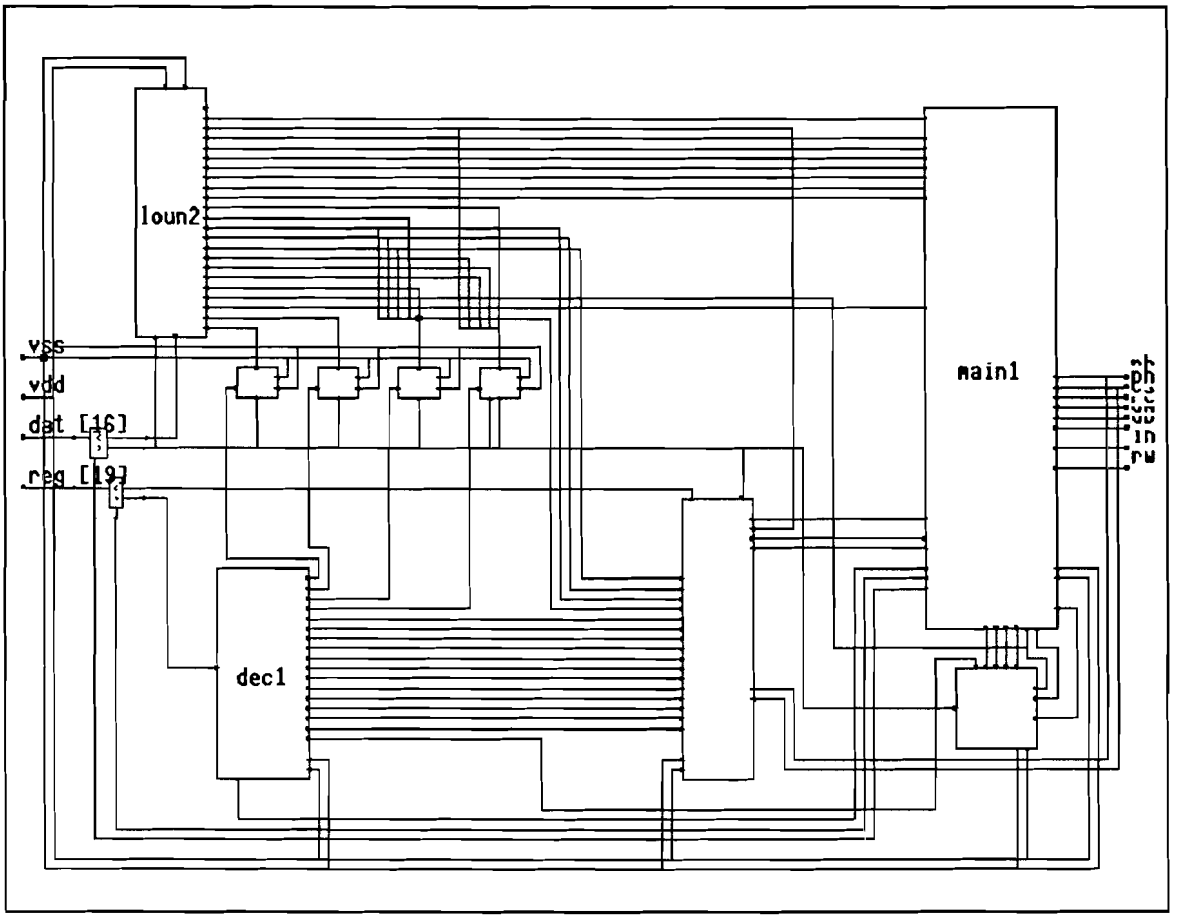

Figure 15. The blitter circuit.

2.6.1 The main controller.

The main controller has been constructed using the behaviour description of the controller from the Escher+ simulation. This behaviour description proved to be very useful for constructing the controller.

```
From the behaviour of byte_cntr:

        (setq wait 1))
    (if (and (= state 4) (= wait 0))
        (progn
          (setq x (+ x 1))                ;for ..; ..; x++
          (if (< x size_1)                ;for..; x < SIZE_W; ..
              (setq state 2)
            (setq state 5))
          (delay 0 trigger state 2)       ;next eval this cycle
          (setq wait 1))
      (if (and (= state 5) (= wait 0))
          (progn
            (delay 0 data 0 1)
            (delay 1 en_ahold 1 2)
            (delay 1 en_bhold 1 2)
            (setq return 6)
            (setq state putdata)
            (delay 2 trigger state 2)       ;next eval after 2 cycles
            (setq wait 1))

The lines from the main controller ESKISS description.


0--------    tr_x        other       0000000000000000 # get next data
1--------    tr_x        tr_y        0001000000000000 # trigger y count
---------    tr_y        l_put       0000000000000000 # put data

# save last data word of this line
---------    l_put       l_wt_pt_d   0000000000000000 # wait logop finished
-------0-    l_wt_pt     l_wt_pt_d   0000000000000000 # wait until pt
-------1-    l_wt_pt     l_en_d      0000000000000110 # enable ddat and d-address
---------    l_en_d      l_wr_d      0000000000000111 # start write cycle
--------0    l_wr_d      l_wr_d      0000000000000110 # wait until ready
--------1    l_wr_d      tr_y        0001000000000000 # trigger y count
0--------    tr_y        ld_x        0100000000000000 # put last word
1--------    tr_y        wait0       0000000000000000 # ready
```

Figure 16. Behaviour to ESKISS translation example.

The controller has been implemented using ESKISS. First an ESKISS description of the controller has been made. ESKISS generated automatically a boolean description for the controller.

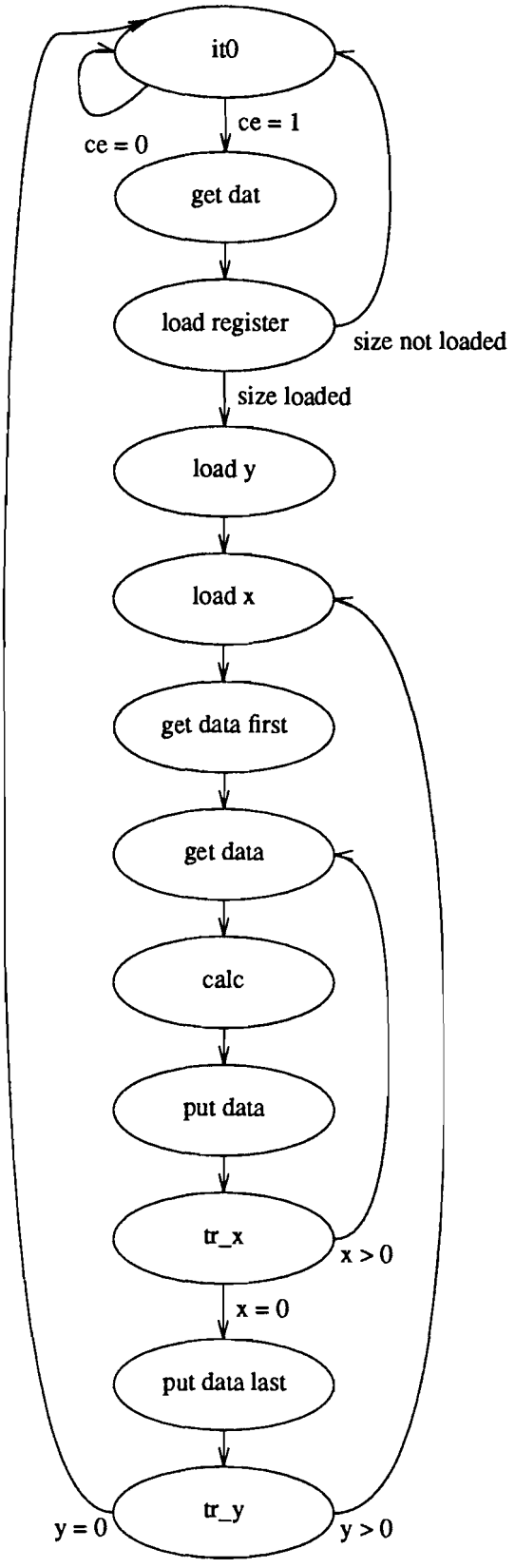

Figure 17. State machine for the main controller.

Because I had already introduced states in the ESCHER+ behaviour of my controller, it was easy to translate this behaviour into an ESKISS description. Here are some lines of the behaviour description, and their corresponding lines in the ESKISS description. For example the command (setq x (+ 1 x)) will be implemented by setting the output pin tr_x high and low again. And the following if statement, by jumping to 2 different states depending on the in the value on the input line zero_x.

The communication with the outside has been described poor in the behaviour description. That is why this part a completely different form the Escher+ behaviours. For the communication in the set-up phase, loading the blitter registers, we now use the protocol described in the C-document, in section 2.4.4. Figure 17, 18 and 19 show the state machine, resulting from this approach. The parts to get and put data have been put in separate figures, because the total figure would have become to big.

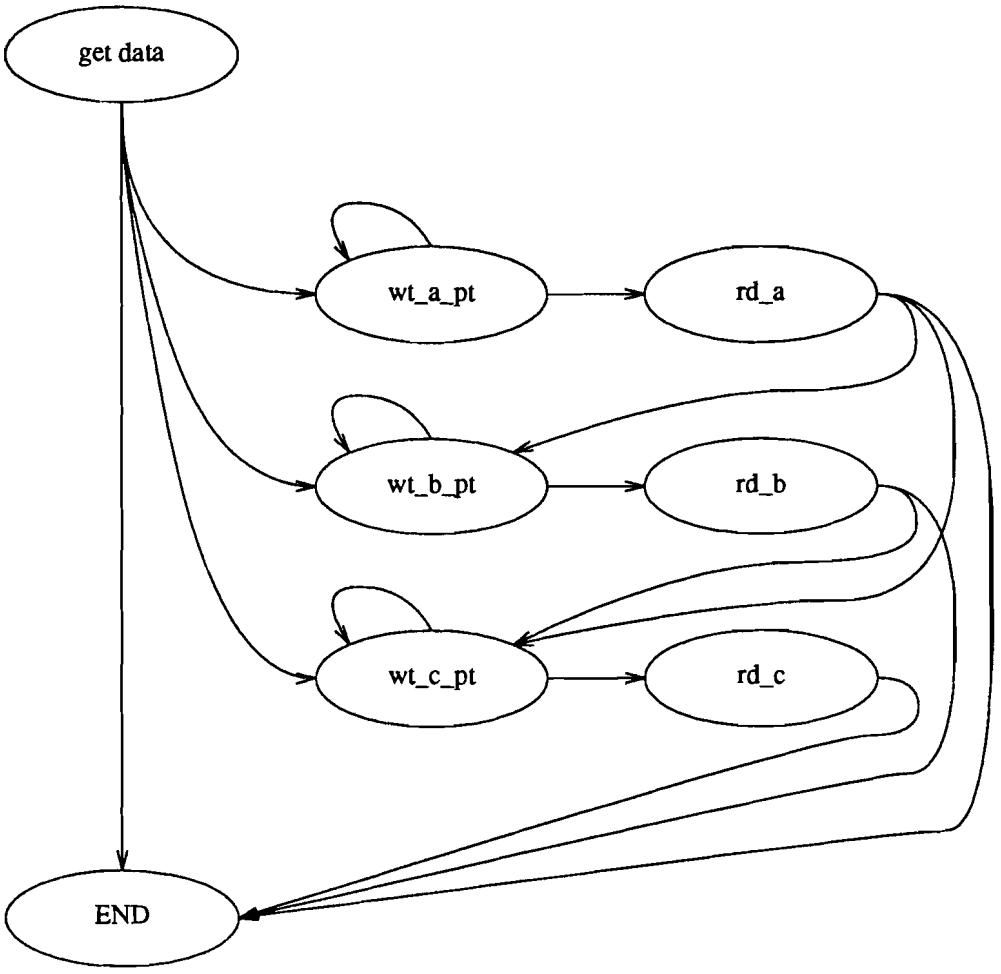

Figure 18. Get data state machine.

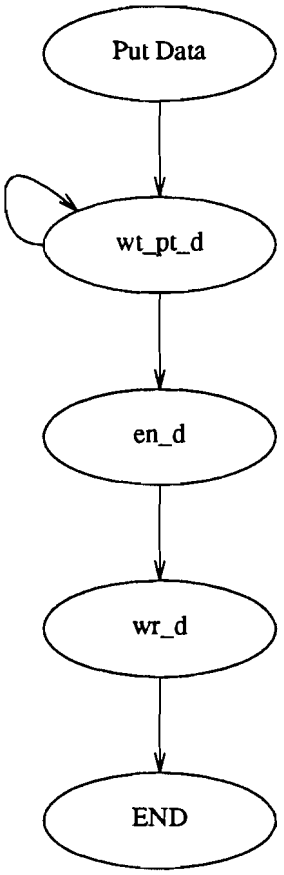

Figure 19. put_data state machine.

The way the blitter reads and writes to memory during the blit is the same as the 68000[2] processor. This will make it easier to find a suitable DMA-manager for the system. If a particular DMA manager is to be used, the blitter can always be changed. The communication protocols are completely enclosed in the main controller, and can be changed by changing its ESKISS description.

To simplify adjusting the controller, not everything has been "squeezed" out of the controller. It will possible to make the controller more compact, but this will be bad for the readability of the description.

The controller has not been tested yet, due to time limitations. Testing may show the need for "wait" states. If modules connected to the controller have very long delay times, it may be necessary to stop the controller for some clock cycles, to allow the module to finish its operations. For example logop, with a fill operation is in use, will take a lot of time.

The complete ESKISS input file is to be found in appendix B.

2.6.2 Address generator.

Each channel has its own pointer and modulo. During the blit, we always have to add 1 in the ascending mode and subtract 1 in descending mode. At the end of a line we have an additional increase or decrease of the modulo. We wanted to use only one ALU to calculate the addresses. We also wanted the addresses to

calculated, while the rest of the blitter was reading the data from memory, or processing it. Therefore this module has it own controller. It enabled the right registers for calculations, and stores the new calculated values.

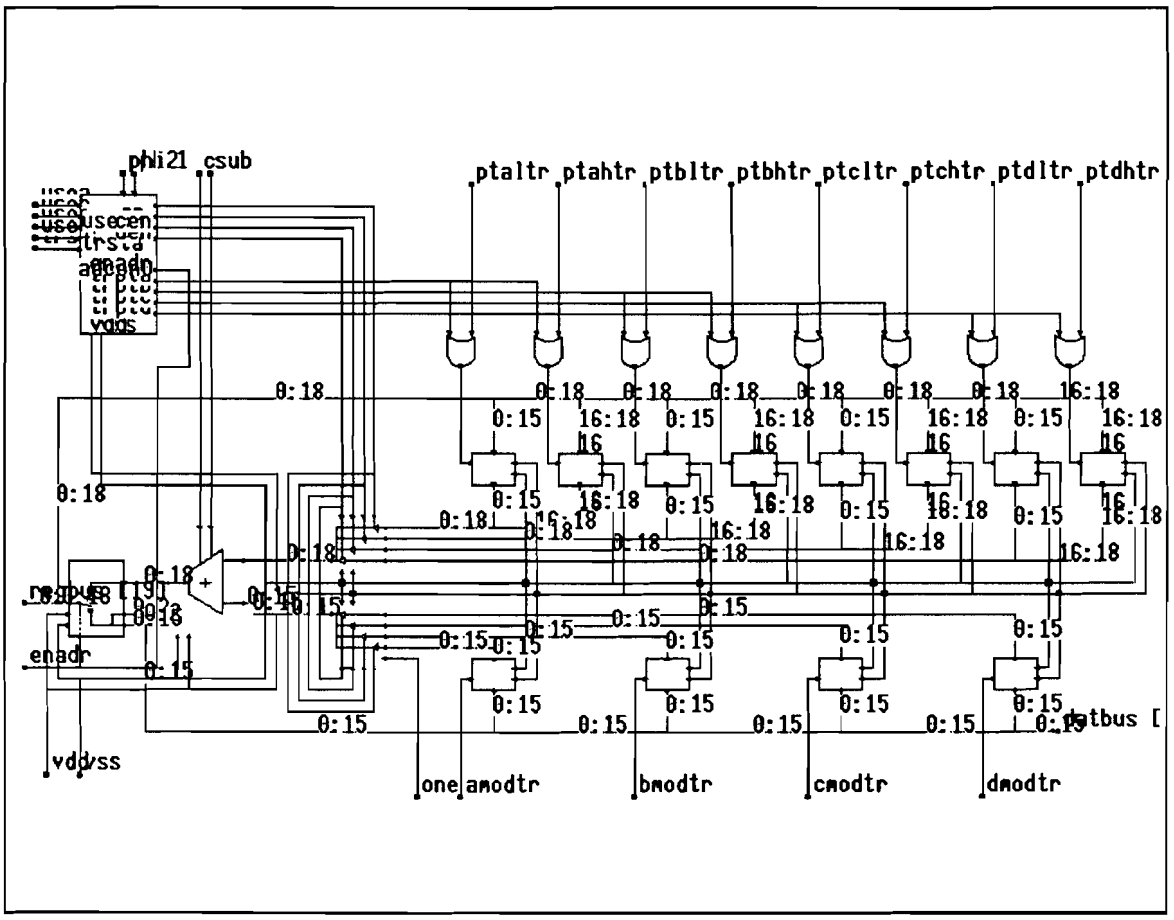

Figure 20. address generator

The modulos can be stored in a 16 bit register. The data input is connected to the data bus. The trigger is available for the main controller. With this line it can load the modulo registers in the set-up phase.

2.6.2.1 The Adder.

The adder takes care of the calculation of the address pointers. It has to add one to the address to get the next address within a line, or the contents of the modulo register at the and of a line. We will use the "last word timer" line to indicate the end of a line. Depending on the ascending or descending mode it has to subtract or add. If the DESC bit is high it has to subtract. Thus the adder can be in 4 different states. Table 1 gives the function for each state.

Because overflows of the adder can only occur in the case of errors, we can can neglect their effects. Then subtracting is the same as adding the 2-complement. The 2-complement of a value is equal to the inverted value plus one. This gives to the 2-complement functions for the adder (where mod' is the bitwise inverted of mod).

The third function reduces to pt + mod'. We can now implement the adder using full-adders. The carry-in bit for the first adder can be used to add one. One input channel for the adder is always the pointer. We get the input for the other and the state of the carry-in bit of table 1.

desc	one	function	2-complement	input	carry in
0	0	pt + mod + 1	pt + mod + 1	mod	1
0	1	pt + 1	pt + 1	0+	1
1	0	pt - mod - 1	pt + (mod' +1) - 1	mod'	0
1	1	pt - 1	pt - 1	1+	1

TABLE 1. Adder states and functions.

In [3] we found this circuit for a full adder, rewritten the "logic_syntax"[4].

```
s : a' b ci' + a' b' ci + a b' ci' + a b ci;
count : a ci + b ci a' + ci' a b;
```

Figure 21. boolean description of a full adder.

Using this description we get the boolean specification for our adder by making a chain of full adders, and adding the special features we wanted. In the first line, the carry in for the first full adder is defined. The "x" intermediate represents the function stated if table 1. For the upper 3 bits the "b" value is always zero, because the modulos are only 16 bit. Thus we can suffice with a simplified version of a full adder.

```
ci : one' desc ;

x0 : b0' sub one'  + b0 sub' one' + desc one;
s0 : a0' x0 ci' + a0' x0' ci + a0 x0' ci' + a0 x0 ci;
c0 : a0 ci + x0 ci a0' + ci' a0 x0;

x1 : b1' sub one'  + b1 sub' one' + one desc ;
s1 : a1' x1 c0' + a1' x1' c0 + a1 x1' c0' + a1 x1 c0;
c1 : a1 c0 + x1 c0 a1' + c0' a1 x1;

   .
   .
   .

x15 : b15' sub one'  + b15 sub' one' + one desc ;
s15 : a15' x15 c14' + a15' x15' c14 + a15 x15' c14' + a15 x15 c14;
c15 : a15 c14 + x15 c14 a15' + c14' a15 x15;

x16 : one desc ;
s16 : a16' x16 c15' + a16' x16' c15 + a16 x16' c15' + a16 x16 c15;
c16 : a16 c15 + x16 c15 a16' + c15' a16 x16;

x17 : one desc ;
s17 : a17' x17 c16' + a17' x17' c16 + a17 x17' c16' + a17 x17 c16;
c17 : a17 c16 + x17 c16 a17' + c16' a17 x17;

x18 : one desc ;
s18 : a18' x18 c17' + a18' x18' c17 + a18 x18' c17' + a18 x18 c17;
```

Figure 22. Boolean description for the adder.

2.6.2.2 Registers

The pointers are 19 bit wide. The least significant 16 bit are stored in a 16 bit register, and the others in a 3 bit register. Their input is connected to the internal register bus. In the set-up phase the low and the high part can be controlled separately. The data bus will be connected to the register bus and the controller can load the registers with the value. During the blit the address generator controller can load the pointer with a newly calculated address.

We used 2-phase non-overlapping clock flip-flops for the registers. The registers were made by writing a gate description. There is a separate tool that generates a layout description for registers. But these static registers are easier in the simulations. For the dynamic registers a certain clock speed is needed, and they don't work without their gate capacities. So SLS simulations, without taking the delays in account, would be much more difficult.

These registers don't provided a scan path, like those generated by the register generator do. This scan path should be implemented in a real design.

```
q0' : ~(q0 + xx0);
q0  : ~(q0' + yy0);
xx0 : ~(d0 + cl);
yy0 : ~(d0' + cl);
d0' : ~(d0);
```

Figure 23. Gate description for one d flip flop.

2.6.2.3 The multiplexers.

The boolean description for the multiplexers has been made, and layout was generated for them. In figure 24 there is a listing of the boolean description of the small multiplexer.

```
# 2-channel multiplexer 4-bit wide

# version 1

out0 : a0 aen + b0 aen';
out1 : a1 aen + b1 aen';
out2 : a2 aen + b2 aen';
out3 : a3 aen + b3 aen';
```

Figure 24. Boolean description for a 2-channel 4-bit multiplexer.

Figure 25 shows the pluri-cell layout for this multiplexer.

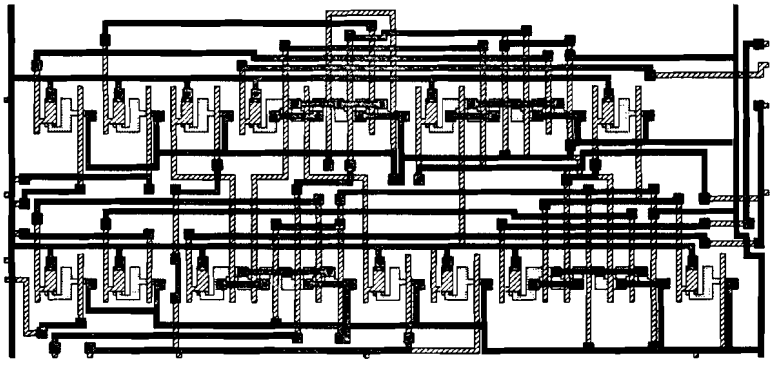

Figure 25. Layout for a small multiplexer.

2.6.2.4 The controller

The controller has to wait until it gets a signal from the main controller that the next address has to be calculated, and the one now present in the selected pointer has to be put on the address bus. If the

calculations are finished, it signals this to the main processor by putting a PT line high.

The input for ESKISS is to be found in fig.26.

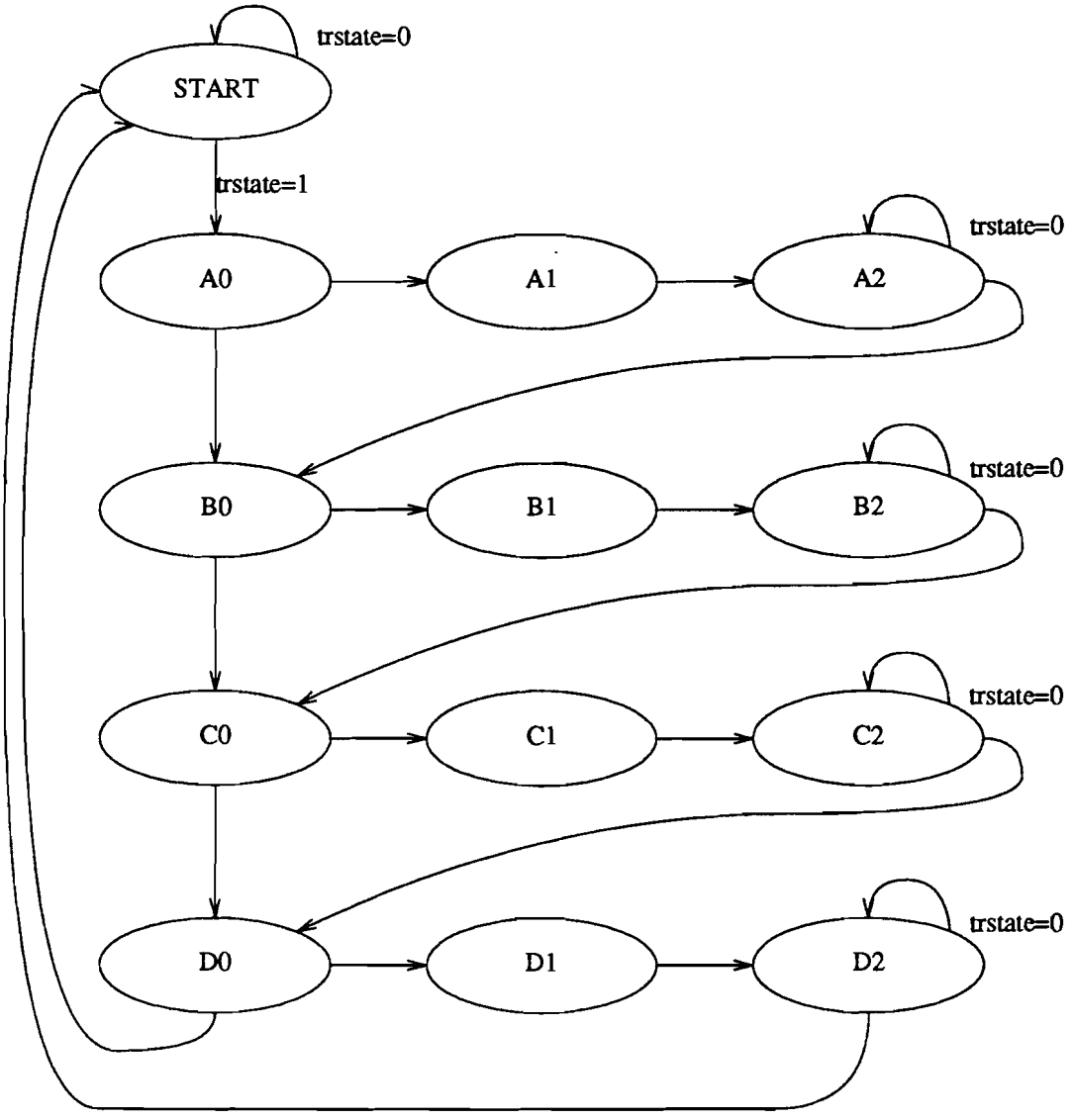

Figure 26. State machine for the address generator controller.

```
# State table for the address generation controller.
# INPUTS:
# usea
# useb
# usec
# used
# trsta
#
# OUTPUTS:
# enadr
# trpta
# trptb
# trptc
# trptd
# aen
# ben
# cen
# den


ABCDtr                  eABCDabcd
#stay in start until triggered (data ready)
-----1  -       start   000001000
----00  start   start   000001000
----10  start   A0      000001000

#calculate A addr only if needed
#enable b-channel registers is necessary
0----0  A0      B0      000000100
1----0  A0      A1      100001000

#delay to allow adder to finish calc. not yet implemented

#update pointer register, and signal main controller that
#address is valid
-----0  A1      A2      010001000

#wait for next trigger signal from main controller
----00  A2      A2      010001000
                                # the pt signal stays high.
----10  A2      B0      000000100

#the same cycle for B-channel
-0---0  B0      C0      000000010
-1---0  B0      B1      100000100

-----0  B1      B2      001000100

----00  B2      B2      001000100
----10  B2      C0      000000010

#the same cycle for C-channel
```

```
--0--0 C0      D0      000000001
--1--0 C0      C1      100000010

-----0 C1      C2      000100010

----00 C2      C2      000100010
----10 C2      D0      000000001

#the same cycle for D-channel
---0-0 D0      start   000001000
---1-0 D0      D1      100000001

-----0 D1      D2      000010001

----00 D2      D2      000010001
----10 D2      start   000001000
```

Figure 27. ESKISS description for the address generator controller.

Pluri-cell layout has been generated for this module, using the register generator of "log_mapper". Both the gate file and the extracted layout have been simulated with SLS.

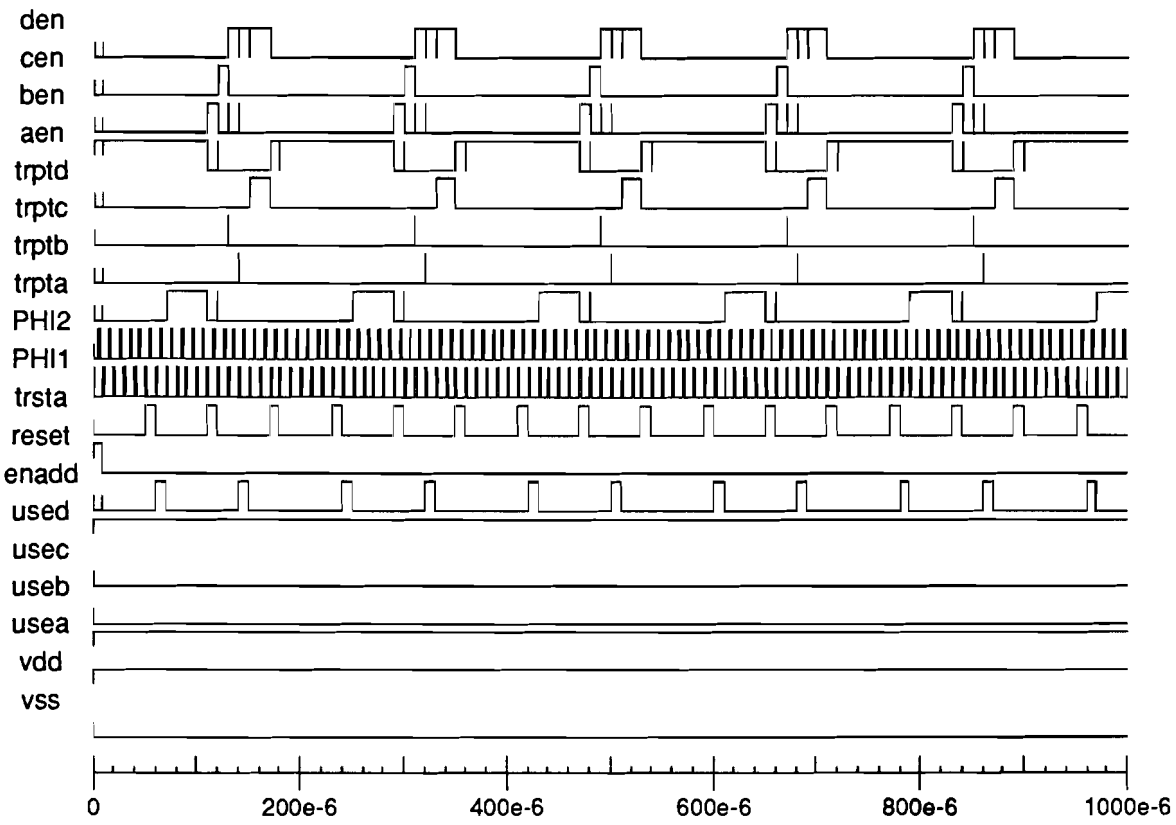

Figure 28. SLS simulation output for the controller.

In this simulation the lines USEA and USED were high. These addresses are calculated with enabling the the register (AEN or DEN high), and then storing the next address (TRPTA or TRPTB high). There are still many spikes on the output lines. They are generate when after a change of the inputs, the system is not yet settled. They can be removed by latching the outputs.

2.6.3 Logic unit.

The logic unit contains everything concerned with processing of the data. When comparing the Escher+ scheme with this on, we see one major difference. Because layout for the shifters became very large, we decided to use one shifter to shift both the A and B channel. This cost some additional multiplexers and registers, but saves one (large) shifter.

The registers and multiplexers used are in principle the same as those from the address generator.

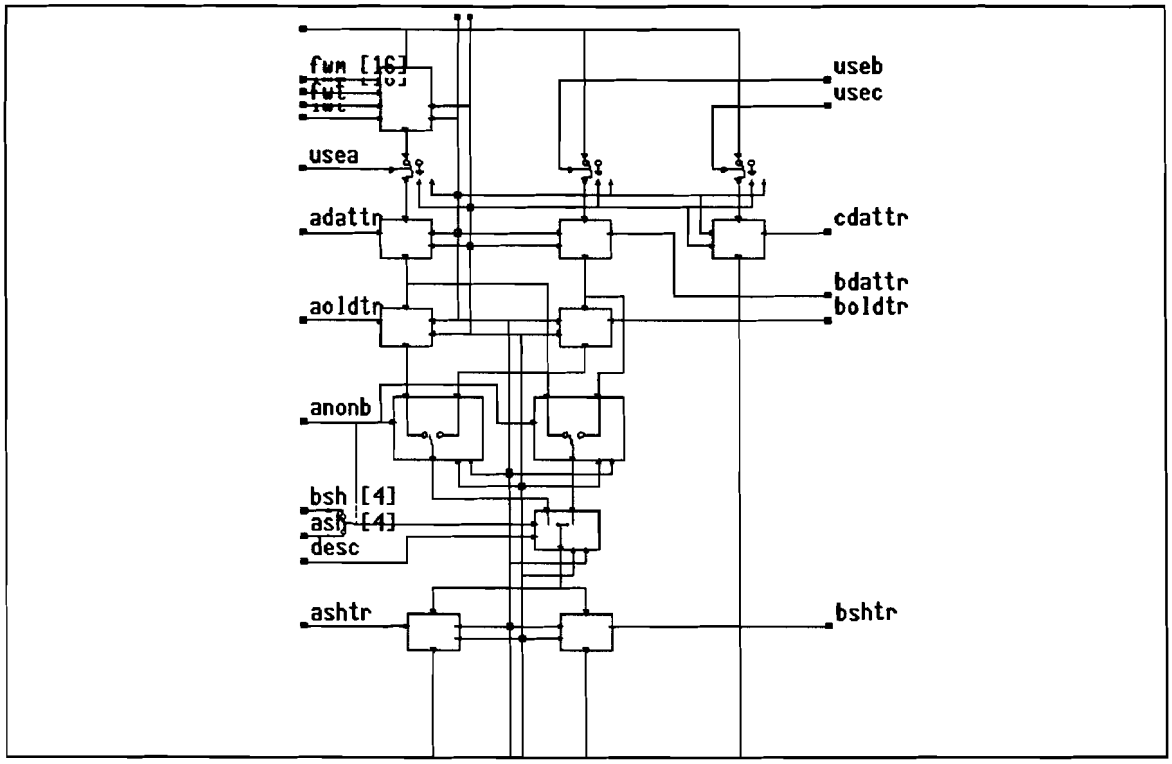

Figure 29. The logic unit.

2.6.3.1 mask

For the mask, the boolean description is obvious. Figure 30 gives the boolean description for one bit. The complete mask consists of 16 such bits.

```
k0  :  fwm0 + fwt';
n0  :  lwm0 + lwt';
out0 :  k0 n0 in0;
```

Figure 30. One bit mask.

This boolean description will give a masking unit described in section 2.3.6 and 2.5.4. A pluri-cell layout has been generated. After an extraction we simulated the mask with SLS, to verify the layout.

2.6.3.2 shifter

Shifting is done in stages. First we do an 8 bit shift, or not, depending on the most significant bit of the shift value. With this new bit vector we do 4 bit shift, followed by a 2 and a 1 bit shift. The generated layout for the shifter has been simulated with SLS. the line "desc" has a very large fanout. The value of "desc" will only change in the set-up phase, so this won't cause many problems.

2.6.3.3 logop

The logop consist as the masking unit of 16 equal cells. Figure 31 shows one such cell. X0 is the result of the boolean function chosen with s0-s7. With this value the filling is performed, if necessary. The output variable "fc0" has to be connected to the "fci" of the next cell.

```
x0  : s7 a0 b0 c0 + s6 a0 b0 c0' + s5 a0 b0' c0 + s4 a0 b0' c0' +
      s3 a0' b0 c0 + s2 a0' b0 c0' + s1 a0' b0' c0 + s0 a0' b0' c0' ;
fc0 : x0 fci' + x0' fci;
d0  : efe fc0 + ife x0 + ife fc0 + efe' ife' x0;
```

Figure 31. One cell for logop.

We made one file with the boolean description of the complete logop, and generated layout for it. The simulations showed, what we already expected, that the settle times for fill and no fill differ very much.

2.6.4 The size controller.

The size controller takes care that the blitter stays in its window. During the set up phase the register SIZE is loaded. The upper 10 bits contain the number of lines, and the others the number of words in one line. The main controller can load the counters with its value. When the controller now triggers the counter, it will decrease the value in the counter. The output lines Y_ONE and X_ONE will become high if the value in the corresponding counter becomes one. This line will also be used as last word timer.

The registers used here are the same as the registers described in section 2.6.2.2.

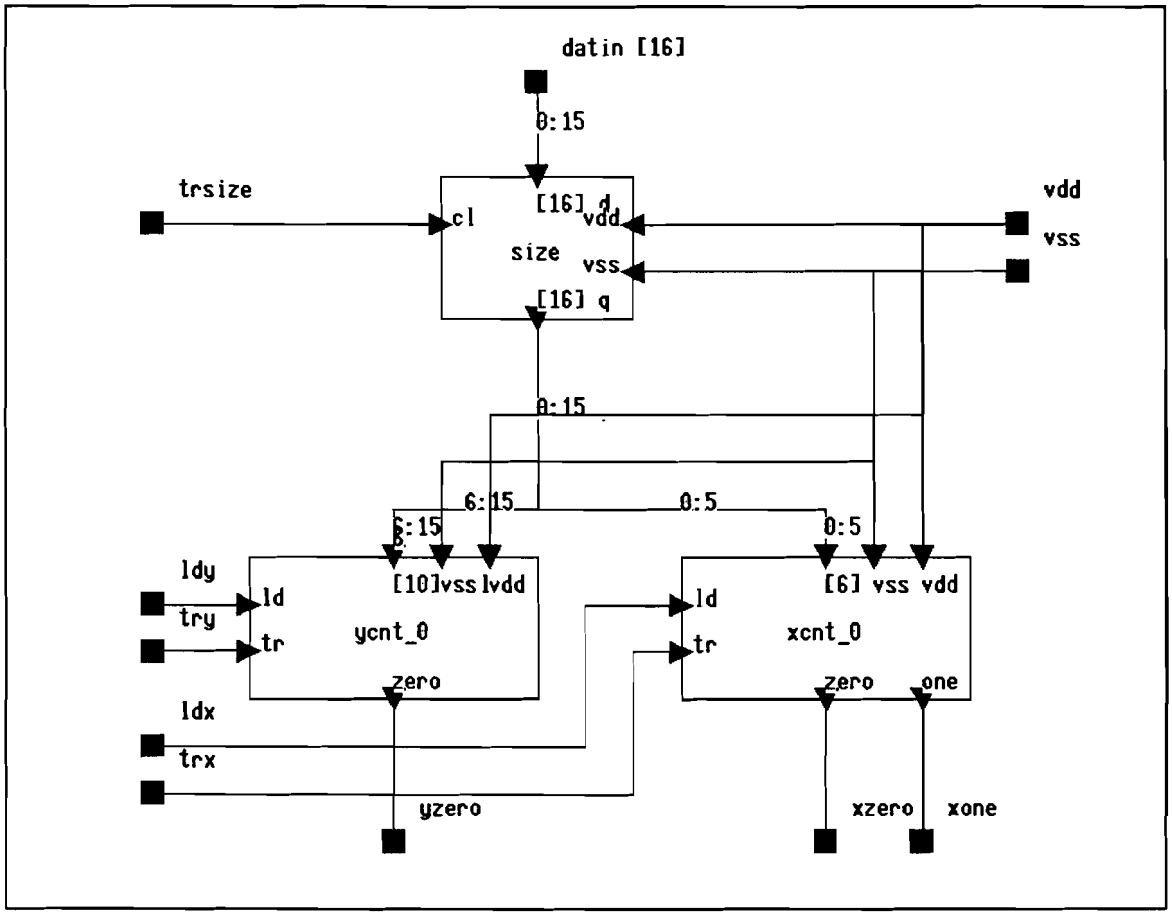

Figure 32. The size controller.

The counters have not yet been implemented. The counter will be triggered by the main controller during the blit, and have to signal to the main controller whether, the end of a line is reached or not. This leads to the following features:

- load facility
- one comparison or zero comparison

or:

- reset facility
- comparison

2.6.5 The register address decoder.

The purpose of the address decoder is simple. It selects the right register during the set-up phase. The bit needed from the address bus, and the selected register will enabled.

The minterms of the combinations of the input bits have been used directly as a boolean description, and a pluri-cell layout has been generated for it.

We have thought about giving the register other addresses. It might be for example useful if two specific bit are always the code for the A, B, C or D channel. This appeared not to be necessary, and we chose the keep the addresses compatible with the Amiga blitter.

3. The ES design system.

In this chapter we will give a short description of each tool, used to design the blitter. In section 2.2 we already explained each tool briefly. Now we will discuss for each tool in detail: input, output, and the problems we encountered during the blitter design.

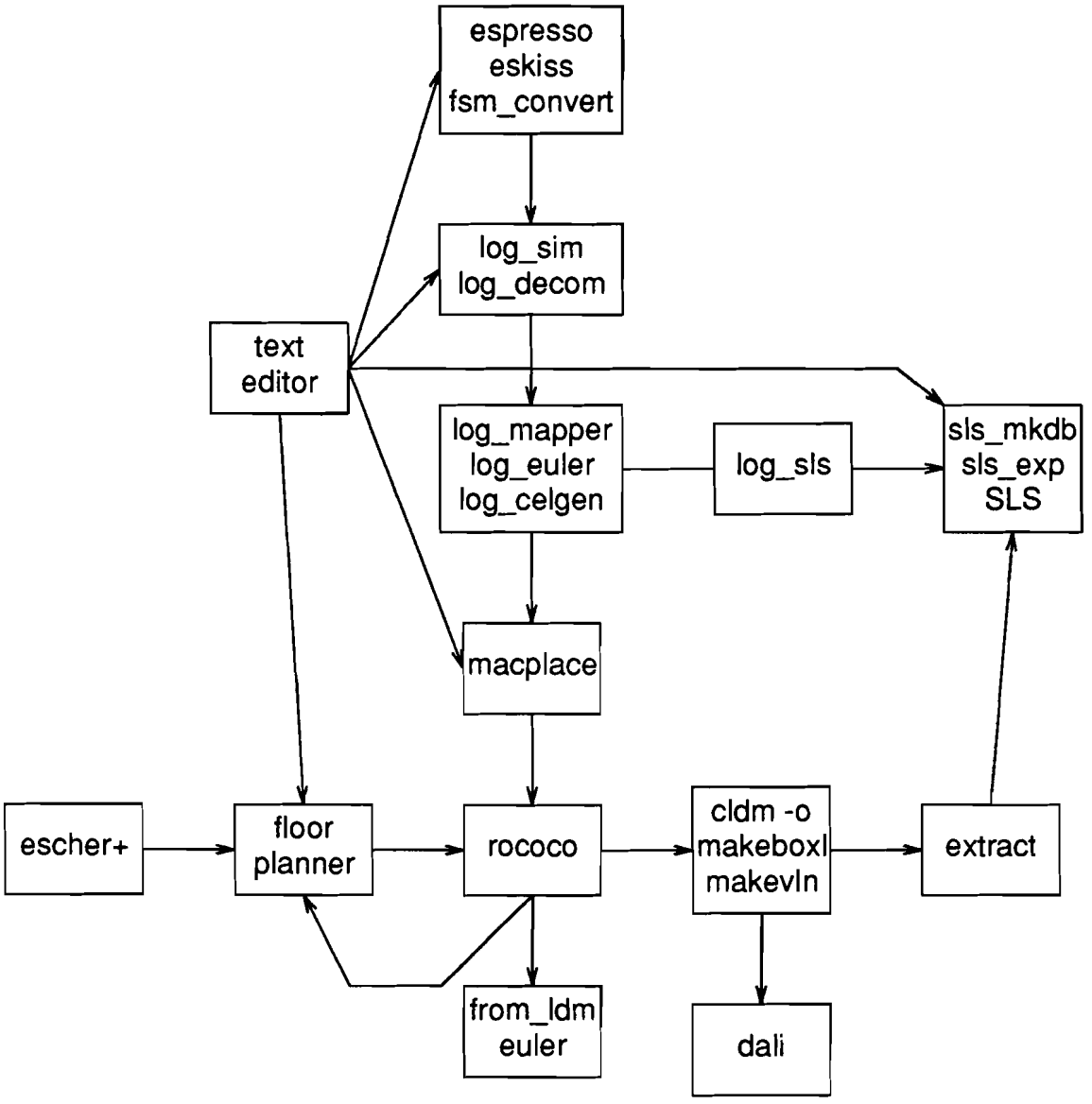

Figure 33. The connection between the tools.

Figure 33 shows the connections between the tools. Only the tools that have been used in this project are shown. The arrows represent a data streams. An arrow from the text editor, indicates that the intermediate files have to be edited.

3.1 ESKISS

This is a set of tools that generate a boolean description for a state machine. The description of the state machine has to be prepared with espresso, before eskiss can process it. The syntax for the input file can be found in the the manual for espresso. The order of the lines is important. The lines with the keywords inputvars, outputvars, mv, type and kiss have to be in the same order as in figure 34.

```
.inputvars usea useb usec used reset trstate
.outputvars enadder trpta trptb trptc trptd
.mv 9 6 -12 -12 5
.type fr
.kiss
----1--        start   00000
----00 start   start   00000
----01 start   A0      00000
0---0- A0      B0      00000
1---0- A0      A1      10000
----0- A1      B0      11000
-0--0- B0      C0      00000
-1--0- B0      B1      00000
----00 B1      B1      00000
----01 B1      B2      10000
----0- B2      C0      10100
--0-0- C0      D0      00000
--1-0- C0      C1      00000
----00 C1      C1      00000
----01 C1      C2      10000
----0- C2      D0      10010
---00- D0      start   00000
---10- D0      D1      00000
----00 D1      D1      00000
----01 D1      D2      10000
----0- D2      start   10001
.end
```

Figure 34. Inputfile for ESKISS.

The numbers after the ".mv" key have the following meaning:

- The number of input variables plus 3.
- The number of input variables.
- The number of different states in the second collumn (a "don't care" is not counted as a state).
- The number of different states in the third collumn.
- number of output variables.

The 3th and 4th entry are preceded with a "-" (minus).

First the input has to be prepared with espresso, using the .kiss entry. Then the program ESKISS can compute a state encoding for state machine. The output of espresso and ESKISS put together, and changed a little. These changes are done with an "awk"[5]. The resulting boolean description in espresso format has to be simplified again with espresso, before it can be converted to the logic syntax with fsm_conv.

All these programs are called by a script, "state_enc", that should do the whole procedure. But this script doesn't work properly. Instead of the "awk", it uses a program with a bug.

3.2 EUCLID

3.2.1 LOG_SIM The logic simplifier.

Log_sim does a simplification and minimization of the input. It handles an expression given as a sum of minterms. The used syntax is the "logic_syntax".

The input is read from standard input. The output are two files:

"listing", contains a listing of the input, and error messages, if there are any,

"data_sim", contains the simplified output.

As an example of the results and the used syntax. The input file:

```
a : c b + b' ;
d : e c b ;
```

The output: "data_sim"

```
a :c +b' ;
d :c b e ;
```

The file listing, for a different input, with some error messages:

```
    1 a : c b + b'
    2 d : e c b ;
##REMARK:   ^14,15
##REMARK:     ^13


***** Error Summary *****

Number of errors in this compilation:  3
    13 : ":" expected; inserted
    14 : ";" expected; inserted
    15 : identifier expected

Number of lines processed  :  2
```

If there is already a file "listing", then this file will not be removed. This can be confusing when log_sim reports an error, but you can't find it in the listing. Remove the file listing, and run log_sim again.

If you have big input files, with many different variables, pay attention. If the number variables in the input exceeds a certain number (something between 30 and 40), log_sim will generate bad, but syntactic good, output. A solution for this program is to split the input file into pieces, and run log_sim on each piece separately. Since log_sim can only find simplifications within a line, the results will be the same.

Log_sim doesn't accept the comment in the form of a line starting with an "#". Runtimes for log_sim are small. Up to a few minutes for very large examples.

3.2.2 LOG_DECOM

Decomposition and minimization of boolean expressions." log_decom decomposes a set of boolean

expressions. A boolean expression is a non redundant set of cubes, a cube is a product term. To obtain a nonredundant set of cubes "log_sim" can be used. log_decom tries to find all the common subexpressions in the given boolean expressions. For each common subexpression a new intermediate variable "intxxx" is created.

It has to be called with:

```
log_decom <inputfile> [<outputfile> <configurationfile>]
```

When no output file is specified, the output is written to the file decom.out. The default configuration file is decom.config. If this file isn't present it uses a default set of parameters. In general these values are good. The default values are listed int the manual page. The expressions must be given as a sum of products. The expressions must be represented by a minimum prime irredundant cover. This can be achieved by the program "log_sim".

```
min_kernel_size   2
min_kernel_amount 2
min_cube_size     2
min_cube_amount   2
max_kernel_subst  5
max_kernels       100
maxdelaytime      100
```

Figure 35. An example of the file decom_config.

Finally a small example:

```
F1 :a c' g' +e f' +a' b' e g' +a' d' e g' +b e' f g +a e' f g +c d e' f g ;
F2 :b e' f g' +a e' f g' +a c' b d f g' +a b' d' f g' +a' c' b' e' f +a' b' d' e' f
  +a c b e f' +a c f' g +a b f' g +e f' g ;
```

Figure 36. Input example for log_decom.

And the results:

```
a : b' + c ;
#  schedtime =    0
#  delay     =    6
d : b c e ;
#  schedtime =    0
#  delay     =    4
f : a + c e ;
#  schedtime =    6
#  delay     =    4
F1 : e g' int10' + f g int10 e' + a c' g' + e f' + e a' d' g' + c d f g e'
        ;
#  schedtime =   10
#  delay     =    5
F2 : f d' e' int10' + f c' e' int10' + f int10 e' g' + a b d f c' g' +
       a f b' d' g' + a b c e f' + a c g f' + a b g f' + e g f' ;
#  schedtime =   10
#  delay     =    5
int10 : b + a ;
#  schedtime =    6
#  delay     =    4
#  gatecount =    6
#  torcount  =   76
```

Figure 37. Output of log_decom for the little example.

Cyclic expressions can cause a crash. You'll probably get this error message:

```
unable to unwind stack because of invalid stack frame (process
manager/process fault manager)
```

Runtimes are small. Up to a few minutes for very large examples.

3.2.3 LOG_MAPPER

Log_mapper maps the input-file, which must be in 'logic- syntax' format, onto a set of standard cells[6]. The standard cell to chose from are: "aoi" (and-or-invert gates), "nor", "nand" or "ao" (and-or) gates. The user can also specify the size of the gates. The default values are right for the standard NMOS process.

There are a number of different functions the user can chose from. Some of these functions can lead to very long runtimes. The functions are described in the manual page.

Log_mapper has to be called with:

```
log_mapper <input-file> [options]
```

The output are files, <filename>.db and <filename>.gf. The file <filename>.db contains a readable description of what log_mapper has done, and some debugging information. The file <filename>.gf contains a description of the circuit in "logic_syntax", and can be used as input for log_celgen.

With the options the users can chose between the different optimizations, the kind of cells to be used, and the size of those cells. Default type is the aoi-cell with a size of 3, 3.

Log_mapper also computes the inputs and outputs of the circuit. If no inputvars or outputvars are specified, they will be listed after the entries "# inputvars" and "# outputvars". If they are already specified, log_mapper will give a warning if there are differences between the calculated lists and the specified lists. The list specified by the user will be put in the outputfile. These entrys are not standard "logic_syntax". If the user specifies for one input both the not inverted and the inverted signal, then those signals will be used, and there will not be made an inverter for that signal. Also for the outputs, if the user specifies also the inverse of a variable, then log_mapper will also generate this inverse.

Figure 38 is the output for the example of log_decom.

```
# gate-type AOI
# inputvars b c e g
# outputvars F1 F2
# functions
sub10' : ~(b + c ) ;
sub9' : ~(c e ) ;
sub8 : ~(c + d' ) ;
sub7 : ~((sub9 b + g sub10 ) a + g e ) ;
sub6 : ~(d' + c' ) ;
sub5 : ~(int10' e' ) ;
sub4 : ~((sub8 b + d' b' ) a + e' int10 ) ;
sub3 : ~((a' d' + int10' ) g' + f' ) ;
sub2 : ~(c d + int10 ) ;
sub1 : ~(g f ) ;
int10' : ~(c + b' + b ) ;
F2 : ~((sub7 + f ) ((sub4 + g ) (sub5 + sub6 ) + f' ) ) ;
F1 : ~((sub3 + e' ) (g + c + a' ) (sub1 + sub2 + e ) ) ;
f' : ~(e c + c + b' ) ;
d' : ~(b c e ) ;
a : ~(b c' ) ;
g' : ~(g) ;
e' : ~(e) ;
c' : ~(c) ;
b' : ~(b) ;
a' : ~(a) ;
d : ~(d') ;
f : ~(f') ;
int10 : ~(int10') ;
sub9 : ~(sub9') ;
sub10 : ~(sub10') ;
```

Figure 38. Example of the output of log_mapper.

The register generator.

Log_mapper can add master-slave registers for a two phase non-overlapping clock for you. This is only possible if you use "aoi" gates. To use the build-in register generator, include the following line in the inputfile:

```
#register <inputvar> <outputvar>
```

Log_mapper now automatically includes a register in the outputfile. The clock signals will have the names "PHI1" and "PHI2".

An example of a register declaration:

```
#register SN4 OS4
```

The corresponding output lines for one of the registers:

```
# register
OS4 : ~(G21 (PHI1' + G22));
G21 : ~(OS4 (PHI1' + G23));
G22 : ~(G23 (PHI2' + SN4));
G23 : ~(G22 (PHI2' + SN4'));
```

An other facility is a comparison and a tautology test of two sets of boolean functions.

The inverter optimalization is sensitive to long carry paths. For example my function logop, which has a very long carry path took over half an hour. But the shifter, with even more transistors, takes only five minutes.

Log_mapper was written in Lisp. There were some problems because the lisp system needs a lot of memory. If there are cyclic definitions in the input, you will get a lisp stack overflow. The machine then enters the lisp debugger. You can kill the debugger with the command "(quit)".

If the delay times estimated by log_mapper are compared with those calculated with extraction and simulation, Log_mapper seems to be rather optimistic.

3.2.4 Cell generation.

3.2.4.1 Log_euler

Log_euler is the next step towards a pluri-cell layout. The pluri-cells consist of a column of transistors for one cell, or a linear transistor array. Log_euler finds a transistor ordering for this linear transistor array. The input for the program is a file with gate description in the gate_syntax format. Log_euler also determines the width of each driver transistor. The result is such that for each input pattern for the cell, the resulting low output voltage will be smaller then 0.5 Volt.

Log_celgen has to be called with:

```
log_euler <inputfile>
```

The transistor ordering is written to standard output, and can be redirected, or piped into log_celgen. Log_celgen generates from this output the layouts of the cells. This output contains some control characters, which complicates reading. In general this isn't a problem, because the user can't do anything with the intermediate results between log_euler and log_celgen.

Log_euler writes to stder, the name of the input file, the error messages, and whether the inputfile contained errors or not.

The version installed now is from April 1988. When it was installed, it still contained a nasty bug. The output generated was syntactic all right, but not correct.

3.2.4.2 log_celgen

Log_celgen reads the transistor ordering made by log_euler, and generates layout descriptions in ldm for the cell, and a netlist.

The format of the netlist is a simplified version of the standard network format (for example used by Escher). In the netlist written by log_celgen, a connection has the following format:

<net name> <instance name> <pin name>

It has to be called with:

```
log_celgen <filename>
```

The input is read from standard input. The layout will be in the file "<filename>.ldm", and the netlist in "<filename>.nlt"

The output of log_euler will usually be piped into log_celgen, there are no other programs that can deal with output of log_euler, and the user can't use it because of the control characters.

Figure 39 shows the lay-out of a cell generated by log_celgen.

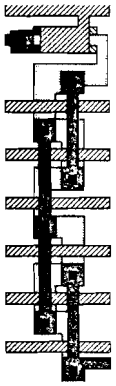

Figure 39. Lay-out for a cell generated by log_celgen.

The layout description generated by log_celgen contains errors. In some of the "box" declarations the coordinates are swapped, the first is larger then the second figure. This is not allowed in the ldm syntax! Such an error has already been deleted once. These errors were re-introduced when a new version of the program was installed, end of April. Already in March I discovered the same kind of error in the old version of log_celgen.

3.3 Placement and routing.

After log_celgen, we have layout for the modules. But these module are not yet placed and routed. This can be done with macplace and rococo. When the layout for a module is ready, if can be placed in a floorplan, with the floorplanner, and routed with rococo.

3.3.1 MACPLACE The Pluri-cell Placer

Macplace computes a placement for the cells generated by log_celgen. The input for macplace is the netlist, in general the netlist generated by log_celgen, and an interface file. The interface file has to have the following format:

```
module <module name>
shape <width> <height>
pin <name> <interval>
pin .....
end
```

After the keyword "module" the user has to give the name of the module. With "shape", the user can control the aspect ratio. The width and height should be given in "layout units". If the given area is to small, macplace will give a warning, and generate a module with the desired aspect ratio.

With the "pin" definitions, the interval in which the pin has to be placed, can be defined. The interval is two floats in the range from 0 to 4 that describe the preferred placement of the terminal:

e.g. upper side: 2 3
left up : 3 3.5

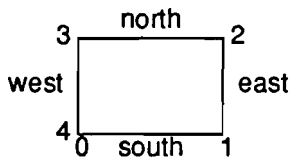

Figure 40. Places of the intervals in the interface file.

The interfacefile ends with the keyword "end".

The netlist generated by log_celgen is not complete. Nets for the connection of the terminals have to be added. An item in the netlist has this format:

<netname> <modulename> <pinname>

Typically you have for each pin entry in the interface file one extra line in the netlist. With the <modulename> the same as in the interface file and <netname> and <pinname> equal to the name of the pin in the interface file. Supply pins are treated differently way by the router. It requires two pins for each supply net, thus we also need two nets in the netlist for each supply net.

Macplace is called with:

```
macplace <options> <netlist file> <interface file>
```

Different aspect ratios will lead to different areas. The differences can be very large. Table ??? shows the results for a rather small example. This causes the strange effects for 1x3 and 3x1. The layout for 2x1 was much higher than necessary.

aspect ratio	width	height	area
1/3	444	1098	487512
1/2	426	917	390642
1/1	756	789	596484
2/1	972	1678	1631016*
3/1	1050	571	599550

TABLE 2. The areas of pluri-cell layout for a small multiplexer for different aspect ratios.

For a larger circuit the results are stated in table 3.

	normal					
aspect ratio	width	height	area	width	height	area
4 / 1	11016	1865	20544840	10124	2011	20359364
3 / 1	8840	2041	18042440	8520	1969	16775880
2 / 1	6178	2587	15982486	6310	2561	16159910
1 / 1	3752	4132	15503264			
1 / 2	2755	7669	21128095	2786	6979	19443494
1 / 3	2189	10184	22292776	2579	11006	28384474
1 / 4	2195	12317	27035815	2243	12643	28358249

TABLE 3. The areas of pluri-cell layout for the shifter for different aspect ratios.

The placement generated by macplace often show a "hill" in the center, The columns in the center are much higher as those at the left and right sides. The option "-f" helps. See the results of table 3. Sometimes the layout with this option will become smaller. Because it adds extra width, it can also give a larger layout.

If there is no place for a pin in the specified interval the router will respond with something like: "too much south terminals".

If Macplace responds with the error message: "interface pin not found in the netlist", it is also possible that the module name in the interface file is not the same as the <module name> used in the netlist for the interface pins.

The placer takes quite a lot of time. To place an example with about 100 transistors it took about half an hour.

3.3.2 The floor planner.

The floorplanner computes a plan for the floor. It chooses a shape for the modules to be placed in the floorplan, and a placement of the modules. Then the modules with the wanted shape have to be generated, and the floorplan can be routed with rococo. Due to time limitations, the floorplanner has not been tested thorough in this project. We didn't use the possibility to let the floorplanner choose from different shapes for the modules. This resulted in a non optimal layout. Using all possibilities of the floorplanner will probably give a better layout.

The netlist has been generated from the escher connection file of the corresponding scheme. For this conversion a emacs-lisp function has been written. This also changes the names of the supply nets into their proper names. In the current escher, it is not possible to give a net a specific name. This will be possible in the new versions of escher.

The placer consists of several programs, to be executed through a make-file. They are not easy to use. Many programs operate upon absolute file names, and error handling is not poor.

All programs were encoded in Pascal. Net and modules are generally stored in arrays. When placing large circuits, it is possible that the arrays are not big enough, and thus they have to be recompiled with larger constants. In one case, the nets were stored in a set. Because in the Apollo Pascal the number of elements in a set is limited to 256 (in other dialects it might even be less). This puts an absolute limit to the number of nets of 256. With Pastoc (PAScal TO C compiler), a C version was made, that allows far more.

3.3.3 ROCOCO The Router.

After all modules have been placed they have to be connected. Rococo (Routing with Contour Compaction) takes care of this[7]. To do this it needs four files:

— <technology file>

— <ldm file>

— <netlist>

— <interface file>

In the technology file there has to be a description of the used technology. The technology file in "/usr/local/lib" for the nmos process, is nearly all right. Only the at the tag "NAMES", the name of the floorplan has to be added at the end of the line. Default this is "root" for the placements made by macplace, and "floor" for those by the floorplanner. Thus default it is not possible to use always the same technology file. In the Cakefile used by us the name "root" is alway replaced by "floor", to be able to use always the same technology file.

The ldm-file has to contain layout descriptions of all the used models, and a description of the floorplan. The floorplan has to be the last module in the ldm-file.

For placements made by macplace the same netlist can be used as for macplace. For a placement made with the floorplanner, the nets for the terminals have to be added to the file "terminal". This has to be done after the the floorplan has been made, because the floorplanner can't cope with these nets.

The interface file is similar to the one used by macplace. When you use the floorplanner, it has to be made manually, else the file made for macplace can be used here also. In our cake system, this file will be generated form the escher interface file.

Rococo has to be called with:

```
rococo -t<tech file> <ldm file> <netlist> <interface file>
```

There is not yet a manual page. Run times are small. For my largest examples it took up to 10 minutes.

The supply nets are routed in a special way. They are the only nets which will be implemented completely in metal. All other nets will routed using both metal and poly. To route the supply nets, the router starts searching from the left bottom point for one supply net, and at right top point for the other. From those points it starts to grasp around the modules (see figure 40).

To be able to guarantee successful routing of the for general floorplans, this means that every module has to have two terminals at opposite sides for each supply net, as in the figure. For placements made with macplace this is not necessary. The modules generated by log_celgen have only single supply pins, but the placement by macplace guarantees reachability.

When routing a floorplan with modules, made with macplace and rococo, rococo can provide this special supply pin placement, when making the modules. To do this, there have to be two pins in the interface file for each supply net. The names of these pins may only differ in the last character, and that character has to be a letter, e.g. vssA and vssB. These pins have to be connected to the suply nets, in entering for each an entry in the netlist. Rococo should now provided a proper placement of the pins.

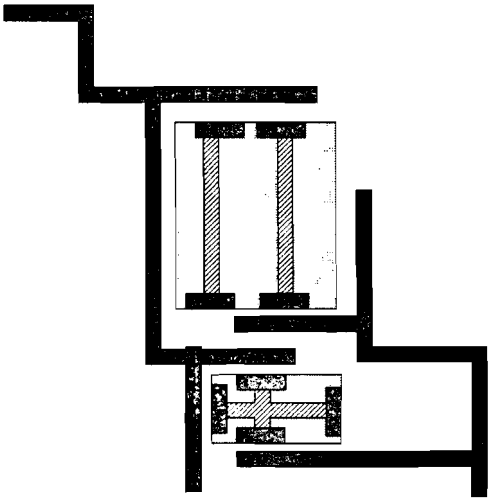

Figure 41. Routing example for a supply net.

It took a lot of time to get this working properly. The main problem was that formerly the supply pins were not always placed on the side of a box by the router, because this was not always necessary, and it saves area. But the router itself has to determine to witch side the terminal belongs, to be able to determine if it is possible to reach the terminal or not. In the old version the terminal belonged to the side closest to a terminal side. This could be wrong if the supply terminal was not at the bounding box. This has been fixed by putting the terminal at the bounding box, and adding a piece of metal from the old position to the new position. Also an other another way to determine to which side terminal belongs has been added, to handle

the problems that occur if a terminal lies exactly in the corner.

There are still troubles with the supply pin placement. Some times, the terminal will not be situated at the box side, and the router will add a description for a piece of metal with a negative size. The main problem is the piece of metal because this not according to the ldm-syntax, and it causes other programs trouble.

Also the the layout, generated by rococo, is much to "high". At the top of a module there is a large gap, filled only with lines to the terminals connected to the upper side. the possibility that this will happen is very small, but if it does, the only cure is to change the aspect ratio in the interface file.

In the ldm syntax nested modules are not allowed. All modules names are known globally. This can cause problems, if pluri-cell modules are enclosed in a floorplan. If a name for a module occurs more then once, this will be fatal for the router. These names can be changed, and rococo started again, but there is an other solution. There is a tool that can make a module flat. It deletes all hierarchy. It is called "flat", and has to be called with:

```
/usr/local/bin/flat /usr/local/lib/nmos.tech < <input-file> > <output-
file>
```

There have been many problems with this rococo. The version installed is still an old Pascal implementation of the program. A C version of the program is ready (but not yet installed). When using this new implementation we encountered many bugs. But is seems that most of them have been found and removed.

3.4 ESCHER+ Schematic Editor and Behaviour Evaluator.

3.4.1 Introduction to escher.

Escher is a schematic entry program. The network drawn will be saved in the standard database format, and can be used by other programs. It is completely menu driven, and all menus are obvious, and explained in [8].

For each module there is a directory in the escher library. Two environment variables control the name and location of the escher library. There are several versions of escher. The main difference between them is the way they store their graphics data. The version used for the blitter design, saves its graphics in a binary file, called "escher_data". When the circuit was correct at the time of saving, it also saved the network data files.

The network will be stored in three files. The file "connection", contains the nets, in the file "interface" there is a list of the terminals of the module. And the file "call" has a list of the instances of modules in this module. If the file "escher_data" becomes corrupt (it occurred a few times with our design) the circuit is lost. Saving modules with an other name can cause such an error. It is very dangerous to copy or rename modules with UNIX commands.

Escher has the tendency to crash sometimes for unknown reasons, especially on the HP system. Due to many checks if the circuit is still legal, escher will become very slow, for large circuits.

It is not yet possible to give a net a user specified name. For example in combination with the router, the supply nets have to have very specific names.

3.4.2 The escher + simulator.

After escher was finished, the behaviour evaluator has been added. It is also menu driven, and all menus are clear. Under Escher+ the user has the possibility to add a behaviour description to each template. The circuit can now be simulated.

These behaviour descriptions have a lisp-like syntax. In v.d. Steen [9] there is a syntax description. The command list is not complete any more. Several commands have been changed of added since then.

The syntax of the behaviour description are not checked thorough. So if there are syntax errors in your descriptions escher+ crashes. If Escher+ does complain about an error, it does not say where it is. Because of this bad error handling, it is difficult to get acquainted with the syntax.

User interrupt handling has been improved recently. Formerly it was impossible the interrupt the simulator when there was a infinite loop in one of the behaviour descriptions. This will not cause no problems any more. The simulation can be interrupted by pushing any key.

It is only possible to do simulations on one level, for all modules in one scheme there has to be a description. This is awkward, because in most cases the user will want to have the possibility to define the behaviour of a module on a lower level also.

Using Escher+ we came to the following list of things that could be improved:

- *Simulations can only be done on one level.*
- *There should be a syntax check of the behaviour files.*
- *It should be possible to define functions.*

- *Global variables and constants should be added.*
- *The error messages you get now do not say in which template the error is.*

At the time of writing, two new versions of Escher were nearly ready. Most of the problems are likely to have been solved in the new ones.

3.5 EULER The Layout Editor.

Euler accepts only <filename>.lay files. The conversion of a ldm file can be done with from_ldm:

```
from_ldm <filename without extension>
```

For the nmos process respond to both questions "1". It will create <filename>.lay file that can be viewed at and changed with euler. It can be converted back to an ldm file with "to_ldm":

```
to_ldm <filename without extension>
```

Euler has problems with the names for modules generated automatically. Names with control characters or longer than 6 characters aren't allowed by euler. Euler allows no underscores or names that start with capital letter. It will try to find other, legal, names. While doing this it sends a lot of errors and warnings. When working on the Apollo you even don't have enough time read them.

If a .lay file is loaded the first time with euler, it often crashes. It will quit with the message "segmentation violation" or just hang up. Kill it and start again.

For very large layout both from_ldm and euler will become very slowly. For the floorplan of the logic unit, from_ldm took 4 days, and euler 30 minutes (only for starting up).

3.6 SLS Switched Level Simulator.

SLS simulate the logical behaviour and the timing of digital MOS circuits. A user manual is enclosed in the ICD book[10]. SLS needs two files, a description of the network and file with the simulation commands. The network description can be extracted from a layout. See the manual for the syntax of the circuit description and the command file.

The simulator uses the Delft database. Here is step by step how to build one. Do it manually the first time, to know each program, and write a shell script to do it for you the second time. Because of all those sub-directories it will be impossible to comprehend the intermediate results anyway!

```
mkpr <project name>
```

It is not allowed to have a file or directory <project name> in the current directory. Now you an amount of empty directories and sub-directories. You can chose any project name, it isn't used elsewhere.

```
cd <project name>

cldm <ldm file>

makeboxl <module name>

makevln  <module name>

extract -L <module name>
```

Use the -L option to get a SLS network description. The <module name> has to be the same as you used with macplace. Afterwards the file <module name>.sls will contain the description. If you want to change the description, do it in this file.

```
sls_mkdb <module name>.sls

sls_exp <module name>
```

When sls_exp responds:

```
      "circuit/test/mc", item NNN: integer expected
```

you probably have a transistor with width or length "10" as NNN-th item. For some curious reason this is not allowed, change it into "9" or "11", and try it again.

After writing the command file you are ready to start the simulation with:

```
sls <module name> <command file>
```

Sls produces two files. The file <module name>.out with the results in a readable format, and <module name>.res, which contains the result in a format suitable for further processing, with for example "slsmenu" or "lpsig". For examples I have done up till now (with a few hundred transistors) the runtime

for the creation of the database is about 20 minutes, and for the simulation itself about 20 seconds.

For each program there is a manual page.

In the output of the extractor, the vss and vdd were interchanged. All depletion transistors were connected to vss. Check this in the file <module name>.sls.

I also tried the postprocessing tools lpsig and slsmenu. Slsmenu is easy to use and all menus are clear. Figure 28 is an example of the output of lpsig.

3.7 DALI Delft Advanced Layout Interface.

Before DALI can be used to view or edit an ldm file, the layout has to be converted to a "Delft Database". This involves the steps executing the programs: "mkpr", "cldm", "makeboxl", and "makevln". In the section about SLS there is a description of these steps. So if you want to look at a piece of layout, for which a SLS simulation has to be made anyway, this doesn't involve extra work. when not, this means a lot of extra work.

For small layouts, EULER is almost always faster, and easier. For large pluri cell modules, it depends if the module is flat or not. If it is flat it doesn't make much difference. If not, the performance is poor compared to EULER, due to the enormous amount of directories in the database.

For "big" floorplans, many large modules, DALI beats EULER. In this case both take their time to show the layout on the lowest level, but you can't see anything in it anyway. DALI has, in contrary to EULER, the possibility to show also a limited number of modules on a lower level. With DALI you can also zoom in on a small part of the floorplan very quickly (EULER not).

In general the menu's are clear, but it is strange the the item to leave the editor is not located at the top level. It is located in a sub menu called "dm_menu".

A. The C-source.

```
#include <stdio.h>
#define BIT(i,a)        (((a) >> (i)) & 1)

typedef unsigned short  WORD;
typedef unsigned char   UBYTE;
typedef unsigned long   ADDRESS;


/* defining the width and the hight of the screen */
#define SCR_W         4    /* byte count !! */
#define SCR_H        20
#define MEM_SIZE      SCR_W*SCR_H

typedef struct PORT
{
            short fildes;
            short size;   /* byte count !!*/
            char  *name;  /* port name   */
} PORT;


/*------------------------ INITIALISING PORT ----------------------------------*/

PORT *port( size , name)
short size;
char  *name;
{
     char *calloc();
     PORT *newport;
            if (newport = (PORT *) calloc( 1, sizeof( PORT )))
            {
```

```
                newport->size   = (size+7)/8;
                newport->fildes = 0;
                newport->name   = name;
                return( newport );
        }
        printf( "Cannot allocate port: size = %d0, size );
        exit(1);
}

/*---------------------- GET ----------------------------------------------*/

unsigned int get( port )
PORT *port;
{
        unsigned int data;
        printf( "-->%15s" , port->name );
        scanf( "%x" , &data );     /* input is supposed to hexadecimal!! */
        printf( "  %u0 , data );

/*      read( port->fildes, ((char * ) (&data+1)) - port->size, port->size );
*/
        return( data );
}

/*---------------------- PUT ----------------------------------------------*/

put( port, data )
PORT *port;
int  data;
{
    printf( "<--%15s %u0 , port->name , data );

/*      write( port->fildes, ((char * ) (&data+1)) - port->size, port->size );
*/
}



/*---------------------- DISPLAY ------------------------------------------*/
WORD  mem[ MEM_SIZE ];

display()
{
    int x , y , i;
        printf( "This is in memory:0 );
        for( y = 0; y < SCR_H; y++ )
        {
                printf( "0 );
                for( x = 0; x < SCR_W; x++ )
                        for( i = 15; i >= 0; i-- )
                                printf( "%1d" , BIT( i , mem[ y * SCR_W + x ] ));
                printf( "%1d" , y % 10 );
        }
        printf( "0 );
}
```

```
/*---------------------- LOAD MEMORY FROM THE FILE "SCREEN" ----------------*/

loadmem()
{
    FILE *fp, *fopm();
         int  i,j;

         if (( fp = fopen( "screen" , "r" )) == 0 )
         {
             printf( "can't open file 'screen'0 );
             exit(1);
         }
         else
         {
             for ( i = 0; i < MEM_SIZE; i++ )
             {
                      fscanf( fp, "%d" , &j );
                      mem[i] = j;
             }
         }
}


#include <stdio.h>
#include "blitter.h"

/*---------------------- BLITTER REGISTERS ----------------------------------*/
/* (see Appendix-A page 2-4)                                    */

WORD bltcon0 = 0;          /* control register 0 */
WORD bltcon1 = 0;          /* control register 1 */
WORD bltsize = 0;          /* size of window, if written starts blit */

ADDRESS bltCpt = 0;        /* high/low source C pointer */
ADDRESS bltBpt = 0;        /* high/low source B pointer */
ADDRESS bltApt = 0;        /* high/low source A pointer */
ADDRESS bltDpt = 0;        /* high/low destination D pointer */

WORD  bltAfwm = 65535,
   bltAlwm = 65535;        /* first/last word masks for A  */
WORD  bltCmod = 0,
   bltBmod = 0,
   bltAmod = 0;            /* modulo's for C, B, and A respectively */
WORD  bltDmod = 0;         /* modulo for destination D */
WORD  bltAdat = 0,
   bltBdat = 0,
   bltCdat = 0;            /* source data registers */
WORD  bltDdat = 0;         /* destination data register */

/*---------------------- SCREEN MEMORY --------------------------------------*/
WORD  mem[ MEM_SIZE ];


/*---------------------- AUXILIARY REGISTERS --------------------------------*/
/* (see Blitter schema fig 6-15, page 196)                      */
```

```
WORD  bltAold; bltBold, bltCold; /* source data registers */

/*---------------------- I/O PORTS ----------------------------------------*/
/* (see Appendix-C page C-2)                                               */

PORT *reg_addr_port,    /* R/W port: R register address; W part RAM address */
   *ram_addr_port,    /* W port:  rest RAM address                  */
   *data_bus,         /* R/W port: data reading and writing         */
   *data_bus_req,     /* W port:  request for data bus              */
   *dma_req,          /* R port:  acknowledge from DMA controller   */
   *ram_write,        /* W port:  selects read/write from/to RAM    */
   *chip_select,      /* R port:  register read enable              */
   *interrupt;        /* W port:  interrupt when blit completed     */

PORT *port();
unsigned int get();
loadmem();

/*---------------------- MAIN ROUTINE -------------------------------------*/

main()
{
        UBYTE reg_address;
        WORD  data;

        /* these should be declarations actually */
        reg_addr_port = port( 8 , "reg_addr_port" );
        ram_addr_port = port( 10 , "ram_addr_port" );
        data_bus      = port( 16 , "data_bus" );
        data_bus_req  = port( 1 , "data_bus_req" );
        dma_req       = port( 1 , "dma_req" );
        ram_write     = port( 1 , "ram_write" );
        chip_select   = port( 1 , "chip_select" );
        interrupt     = port( 1 , "interrupt" );

        /* initialisation of the screen memory */
        loadmem();

        while (TRUE)
        {
                reg_address = get( reg_addr_port );
                if ( get( chip_select ))
                {
                        data = get( data_bus );
                        write_register( reg_address, data );
                        if (reg_address == BLTSIZE)
                        {
                              blit();
                                    put( interrupt , 1);
                        }
                }
         }
}
/*---------------------- WRITING A REGISTER -------------------------------*/
```

```
write_register( addr, data )

UBYTE addr;
WORD  data;
{
        switch (addr)
        {
        case BLTCON0: bltcon0 = data; break;
        case BLTCON1: bltcon1 = data; break;
        case BLTSIZE: bltsize = data; break;

        case BLTAFWM: bltAfwm = data; break;
        case BLTALWM: bltAlwm = data; break;

        case BLTAPTL: PUTLOW( bltApt, data ); break;
        case BLTAPTH: PUTHIGH( bltApt, data ); break;
        case BLTBPTL: PUTLOW( bltBpt, data ); break;
        case BLTBPTH: PUTHIGH( bltBpt, data ); break;
        case BLTCPTL: PUTLOW( bltCpt, data ); break;
        case BLTCPTH: PUTHIGH( bltCpt, data ); break;
        case BLTDPTL: PUTLOW( bltDpt, data ); break;
        case BLTDPTH: PUTHIGH( bltDpt, data ); break;

        case BLTAMOD: bltAmod = data >> 1; break;
        case BLTBMOD: bltBmod = data >> 1; break;
        case BLTCMOD: bltCmod = data >> 1; break;
        case BLTDMOD: bltDmod = data >> 1; break;

        case BLTADAT: bltAdat = data; break;
        case BLTBDAT: bltBdat = data; break;
        case BLTCDAT: bltCdat = data; break;
        case BLTDDAT: bltDdat = data; break;
        }
}

/*---------------------- ACTUAL BLITTING ----------------------------------*/

blit()
{
        WORD logop();
        int i, j, fco;

        for ( i = HEIGHT( bltsize ); i>0 ; i-- )
        {
            bltAdat = 0;    /* reset registers at the start of a     */
            bltBdat = 0;    /* new line.                             */
            bltCdat = 0;
                fco = BIT( FCI, bltcon1 );  /* reset the fill carry bit  */
            get_data();
                bltAdat = bltAdat & bltAfwm;

                for ( j = WIDTH( bltsize ) - 2; j>0 ; j-- )
                {
                        get_data();
                        bltDdat = logop( SHIFT( bltAold, bltAdat,
```

```
                                        SH( bltcon0), BIT( DESC , bltcon1 ) ),
                                        SHIFT( bltBold, bltBdat, SH( bltcon1 ),
                                            BIT( DESC, bltcon1 ) ), bltCold, fco );
                        put_data();
                }

                if (WIDTH( bltsize ) > 1)
                {
                        get_data();
                        bltAdat = bltAdat & bltAlwm;
                        bltDdat = logop( SHIFT( bltAold, bltAdat,
                                        SH( bltcon0 ), BIT( DESC , bltcon1 ) ),
                                        SHIFT( bltBold, bltBdat, SH( bltcon1 ),
                                        BIT( DESC, bltcon1 ) ), bltCold, fco );
                        put_data();
                }
                else
                        bltAdat = bltAdat & bltAlwm;
                bltDdat = logop( SHIFT( bltAdat, 0, SH( bltcon0 ),
            BIT( DESC, bltcon1 ) ), SHIFT( bltBdat, 0,
                  SH( bltcon1 ), BIT( DESC, bltcon1 ) ), bltCdat,
                        fco );
                put_data();
                if (BIT( DESC, bltcon1 ))
                {
                        bltApt -= bltAmod;
                        bltBpt -= bltBmod;
                        bltCpt -= bltCmod;
                        bltDpt -= bltDmod;
                }
                else
                {
                        bltApt += bltAmod;
                        bltBpt += bltBmod;
                        bltCpt += bltCmod;
                        bltDpt += bltDmod;
                }
        }
}

/*---------------------- LOGIC OPERATION ----------------------------------*/

WORD logop( Adata, Bdata, Cdata, fco )

WORD Adata, Bdata, Cdata;
{
    WORD ddat;
        int j;

        ddat = ( (BIT( 7, bltcon0 ) *  Adata &  Bdata &  Cdata ) |
              (BIT( 6, bltcon0 ) *  Adata &  Bdata & ~Cdata ) |
              (BIT( 5, bltcon0 ) *  Adata & ~Bdata &  Cdata ) |
            (BIT( 4, bltcon0 ) *  Adata & ~Bdata & ~Cdata ) |
                    (BIT( 3, bltcon0 ) * ~Adata &  Bdata &  Cdata ) |
              (BIT( 2, bltcon0 ) * ~Adata &  Bdata & ~Cdata ) |
```

```
            (BIT( 1, bltcon0 ) * ~Adata & ~Bdata &  Cdata ) |
            (BIT( 0, bltcon0 ) * ~Adata & ~Bdata & ~Cdata ));
        if( BIT( IFE, bltcon1 ) | ( BIT( EFE, bltcon1 )))
            for( j=0; j<16; j++ )
                {
                    fco = fco ^ BIT( j, ddat );
                        if BIT( EFE, bltcon1 )
                        {
                            if ( fco == 1 )
                                    ddat = ddat | MASK( j );
                            else
                                    ddat = ddat & ~MASK( j );
                        }
                        if BIT( IFE, bltcon1 )
                        {
                            if ( fco | BIT( j, ddat ))
                                    ddat = ddat | MASK( j );
                            else
                                    ddat = ddat & ~MASK( j );
                        }

                }
        return( ddat );
}

/*---------------------- READING MEMORY ----------------------------------*/

WORD getword( addr )

ADDRESS addr;
{
/*reading from memory is done in the array mem  */
/*  put( data_bus_req, 1 );                 */
/*  while( get( dma_req ) == 0 );           */
/*          put( ram_write,    0 );         */
/*          put( reg_addr_port, REG_ADDR( addr ) ); */
/*          put( ram_addr_port, RAM_ADDR( addr ) ); */
/*          return( get( data_bus ));       */

    return( mem[ addr ] );
}

get_data()
{
        bltAold = bltAdat;
        bltBold = bltBdat;
        bltCold = bltCdat;
        if (BIT( USEA, bltcon0 ))
          bltAdat = getword( bltApt );
        if (BIT( USEB, bltcon0 ))
          bltBdat = getword( bltBpt );
        if (BIT( USEC, bltcon0 ))
          bltCdat = getword( bltCpt );
        if (BIT( DESC, bltcon1 ))
        { bltApt--; bltBpt--; bltCpt--; }
```

```
        else
        { bltApt++; bltBpt++; bltCpt++; }
}

/*---------------------- WRITING MEMORY ----------------------------------*/

putword( addr, data )

ADDRESS addr;
WORD    data;
{
/* writing inmemory is done in the array mem       */
/*        put( data_bus_req, 1 );                  */
/*        while( get( dma_req ) == 0 );            */
/*        put( ram_write,    1 );                  */
/*        put( reg_addr_port, REG_ADDR( addr ) );  */
/*        put( ram_addr_port, RAM_ADDR( addr ) );  */
/*        put( data_bus,     data );               */

    mem[ addr ] = data;
}

put_data()
{
        if (BIT( USED, bltcon0 ))
          putword(bltDpt, bltDdat);
        if (BIT( DESC, bltcon1 ))
                bltDpt--;
        else
                bltDpt++;
}
```

```
#include <stdio.h>

#define TRUE    1
#define FALSE   0
#define PORT    unsigned int    /* just to fool the compiler */

typedef unsigned short  WORD;
typedef unsigned char   UBYTE;
typedef unsigned long   ADDRESS;

#define MIN(a,b)       (((a)<(b))?(a):(b))
#define MAX(a,b)       (((a)>(b))?(a):(b))
#define BIT(i,a)       (((a) >> (i)) & 1)
#define SHIFT(a,b,n,d) (WORD)(d) ?
        ((((((long) (a) << 16) | (b)) << (n)) >> 16) & 0xFFFF) :
        ((((((long) (b) << 16) | (a)) >> (n)) & 0xFFFF))
#define MASK(i)        ((1) << (i))
/* Reading shift values from the control registers, rest is extracted using BIT */
#define SH(a)          (((a) >> 12) & 0xf)
#define USEA           11
```

```
#define USEB          10
#define USEC           9
#define USED           8
#define EFE           4
#define IFE           3
#define FCI           2
#define DESC           1
#define LINE           0

/* Reading sizes from size register */
#define WIDTH(a)       ((a) & 0x3f)
#define HEIGHT(a)      (((a) >> 6) & 0x3ff)

/* On input addresses written in two parts: bits 1-15 (LOW), 16-18 (HIGH) */
#define PUTLOW(a,d)    ((a) = ((a) & ~0x7fff) | (((d) >> 1) & 0x7fff ))
#define PUTHIGH(a,d)   ((a) = ((a) &  0x7fff) | (((d) & 0x7 ) << 7))

/* Addresses are written to two address ports: reg (1-8) and ram (9-17) */
#define REG_ADDR(a)    ((a) & 0xff )
#define RAM_ADDR(a)    (((a) >> 8) & 0x1ff )

#define A       0
#define B       1
#define C       2
#define D       3

#define BLTCON0   0x40
#define BLTCON1   0x42

#define BLTAFWM   0x44
#define BLTALWM   0x46

#define BLTAPTH   0x50
#define BLTAPTL   0x52

#define BLTBPTH   0x4c
#define BLTBPTL   0x4e

#define BLTCPTH   0x48
#define BLTCPTL   0x4a

#define BLTDPTH   0x54
#define BLTDPTL   0x56

#define BLTAMOD   0x64
#define BLTBMOD   0x62
#define BLTCMOD   0x60
#define BLTDMOD   0x66

#define BLTADAT   0x74
#define BLTBDAT   0x72
#define BLTCDAT   0x70
#define BLTDDAT   0x0

#define BLTSIZE   0x58
```

```
/* defining the width and the hight of the screen */
#define SCR_W          4    /* byte count !! */
#define SCR_H         20
#define MEM_SIZE      SCR_W*SCR_H



#include <stdio.h>
#define  SCR_H   20
#define  SCR_W   4
#define  MEM_SIZE SCR_W*SCR_H
#define  BIT(i,a) (((a) >> (i)) & 1)


int  i,
   mem[ MEM_SIZE ];

main()
{

 loadmem();
 for( i = 0; i < MEM_SIZE; i++ )
  printf( "%d", mem[ i ] );
 display();
}

loadmem()
{
     FILE  *fp, *fopm();
           int  i,j;

           if (( fp = fopen( "screen" , "r" )) == 0 )
           {
                printf( "can't open file 'screen'0 );
                exit(1);
           }
           else
           {
                for ( i = 0; i < MEM_SIZE; i++ )
                {
                          fscanf( fp, "%d" , &j );
                          mem[i] = j;
                }
           }
}


display()
{
     int x , y , i;
           printf( "This is in memory:0 );
           for( y = 0; y < SCR_H; y++ )
           {
                     printf( "0 );
```

A. Eskiss input for the main controller.

```
# this state machine controls the the blitting process.

# Wim Philipsen July 19 1988.

# It takes care of loading the x and y counters,
# and of the first word timer. The counter is supposed to take care
# of the last word timer.


# To this controller belongs a logic circuit with 2 downcounters,
# one for x and one for y. And a register containing the size
# value. The x counter has an output line that is high if the contence of
# the register is one. This line is used as last word timer.

# inputs       zero_x  x zero flag
#              one_y   y one flag
#              ce      chip enable
#              size    size addressed by the decoder? 1 if yes
#              usea
#              useb
#              usec
#              pt
#              read_ready

# outputs      load_y          (re)load y counter
#              load_x          (re)load x counter
#              fwt             first word timer
#              tr_x            trigger x counter
#              tr_y            trigger y counter
#              tr_aold         load aold register
#              tr_bold         load bold register
#              zero    a=b=0   do not use a and b
#              addr_en_r       read from address port
#              addr_tr         trigger address bus latches
```

```
#             data_en_r      read from data port
#             data_tr        trigger data bus latches
#             dec_en         enable address decoder (for register load)
#             tr_sta         address generator can calculate the next adderss
#             read_start     start the read cycle
#             adat_tr        store a data
#             bdat_tr        store b data
#             cdat_tr        store c data

xycsabcpr                                  yxfxyabzaadddtrabc
--0------     wait0          wait0         0000000010100000000 # wait until chip addressed
--1------     wait0          getdat        0000000011110000000 # go get data from the bus
---------     getdat         load_reg      0000000010101000000 # load the registers
---0-----     load_reg       wait0         0000000010100000000 # wait for next data
---1-----     load_reg       first         1000000000000000000 # start the blitter, because
                                                               # sizereg has been loaded
                                                               # load y
# get the first data words, and load x
----1----     first          f_wt_a_pt     0100000000000000000 # go get A
----01---     first          f_wt_b_pt     0100000000000000000 # go get B
----001--     first          f_wt_c_pt     0100000000000000000 # go get C
----000--     first          new_wrd       0100000000000000000 # do nothing

# pt = pta+ptb+ptc+ptd, signal that
# the last calculation is finished.
-------0-     f_wt_a_pt      f_wt_a_pt     0000000000000000000 #wait until pt = 1
-------1-     f_wt_a_pt      f_rd_a        0000000000000001000 #start read-cycle

# get data first
# wait until data available
# And if available load a_data
--------0     f_rd_a         f_rd_a        0000000000000000000 # wait until pt=1
-----1--1     f_rd_a         f_wt_b_pt     0000000000000000100 # save adat and go get B
-----01-1     f_rd_a         f_wt_c_pt     0000000000000000100 # save adat and go get C
-----00-1     f_rd_a         new_wrd       0000000000000000100 # save adat and goto end

# for the B-channel

# pt = pta+ptb+ptc+ptd, signal that the last calc is finished

-------0-     f_wt_b_pt      f_wt_b_pt     0000000000000000000 # wait until pt = 1.
-------1-     f_wt_b_pt      f_rd_b        0000000000000001000 # start read-cycle

# wait until data available

--------0     f_rd_b         f_rd_b        0000000000000000000 # wait until read
------1-1     f_rd_b         f_wt_c_pt     0000000000000000010 # save bdat and go get C
------0-1     f_rd_b         new_wrd       0000000000000000010 # save bdat and goto new_wrd

# pt = pta+ptb+ptc+ptd, signal that
# the last calculation is finished

-------0-     f_wt_c_pt      f_wt_c_pt     0000000000000000000 #wait until pt = 1
-------1-     f_wt_c_pt      f_rd_c        0000000000000001000 # start read-cycle
```

```
# wait until data available
--------0      f_rd_c          f_rd_c          0000000000000000 # wait until read
--------1      f_rd_c          new_wrd         0000000000000001 # save c_dat and goto new_wrd

#get data
----1----      new_wrd         wt_a_pt         0000000000000000 # load x, goto getdata_first
----01---      new_wrd         wt_b_pt         0000000000000000 # go get B
----001--      new_wrd         wt_c_pt         0000000000000000 # go get C
----000--      new_wrd         calc            0000000000000000 # do nothing

# pt = pta+ptb+ptc+ptd, signal that
# the last calculation is finished.
-------0-      wt_a_pt         wt_a_pt         0000000000000000 #wait until pt = 1
-------1-      wt_a_pt         rd_a            0000000000001000 #start read-cycle

# wait until data available
# And if available load a_data
--------0      rd_a            rd_a            0000000000000000 # wait until pt=1
-----1--1      rd_a            wt_b_pt         0000000000000100 # save adat and go get B
-----01-1      rd_a            wt_c_pt         0000000000000100 # save adat and go get C
-----00-1      rd_a            calc            0000000000000100 # save adat and goto calc

# for the B-channel

# pt = pta+ptb+ptc+ptd, signal that the last calc is finished

-------0-      wt_b_pt         wt_b_pt         0000000000000000 # wait until pt = 1.
-------1-      wt_b_pt         rd_b            0000000000001000 # start read-cycle

# wait until data available

--------0      rd_b            rd_b            0000000000000000 # wait until read
------0-1      rd_b            calc            0000000000000010 # save bdat and goto calc
------1-1      rd_b            wt_c_pt         0000000000000010 # save bdat and go get C

# pt = pta+ptb+ptc+ptd, signal that
# the last calculation is finished

-------0-      wt_c_pt         wt_c_pt         0000000000000000 #wait until pt = 1
-------1-      wt_c_pt         rd_c            0000000000001000 # start read-cycle

# wait until data available
--------0      rd_c            rd_c            0000000000000000 # wait until data read
--------1      rd_c            calc            0000000000000001 # save c_dat and goto calc
---------      calc            put             0000000000000000 # data loaded, wait or calculations


---------      put             wt_pt_d         0000000000000000 # wait logop finished
-------0-      wt_pt           wt_pt_d         0000000000000000 # wait until pt
-------1-      wt_pt           en_d            0000000000000110 # enable ddat and d-address
---------      en_d            wr_d            0000000000000111 # start write cycle
--------0      wr_d            wr_d            0000000000000110 # wait until ready
--------1      wr_d            tr_x            0011000000000000 # d saved

0--------      tr_x            other           0000000000000000 # get next data
```

```
1--------       tr_x            tr_y            000010000000000000 # trigger y count
---------       tr_y            l_put           000000000000000000 # put data

# save last data word of this line
---------       l_put           l_wt_pt_d       000000000000000000 # wait logop finished
-------0-       l_wt_pt         l_wt_pt_d       000000000000000000 # wait until pt
-------1-       l_wt_pt         l_en_d          000000000000000110 # enable ddat and d-address
---------       l_en_d          l_wr_d          000000000000000111 # start write cycle
--------0       l_wr_d          l_wr_d          000000000000000110 # wait until ready
--------1       l_wr_d          tr_y            000010000000000000 # trigger y count
0--------       tr_y            ld_x            010000000000000000 # put last word
1--------       tr_y            wait0           000000000000000000 # ready
```

REFERENCES

1. Commodore Business Machines, Inc., "Amiga Hardware Reference Manual", Addison-Wesley, 1986.
2. William Cramer, Gerry Kane, "68000 Microprocessor Handbook.", Osborne McGraw-Hill, 1986.
3. M.E. Sloan, "Computer Hardware and Organization", Science Research Associates, Inc. 1983.
4. J.T.H. Verhoeven, "A Software Inter..."
5. "unix reference manual"
6. M.R.C.M. Berkelaar, "Technology Mapping from Boolean expressions to Standard Cells", EUT Research Report, 1987.
7. L.P.P.P. van Ginneken, "Gridless Routing for Generalized Cell Assemblies", EUT Research Report.
8. A. Lodder, M.T. van Stiphout, J.TJ. van Eijndhoven, "Eindhoven SCHematic EditoR, Reference Manual.", EUT Research Report.
9. H.L.J. van der Steen, "Interactive Event-driven Simulation", Master thesis TUE.
10. P. Dewilde, "The Integrated Circuit Design Book", Delft University Press, 1986.