Eindhoven University of Technology
Faculty of Electrical Engineering
Section Digital Systems

Design of a Raster Image Processing unit
for a laser printer front-end

P.S.E. Steemers

Report on a final project performed during the period of september 1990 to may 1991.

Professor: Prof. Ir. M.P.J. Stevens

Coaches: Prof. Ir. M.P.J. Stevens,
Ir. J.W.M. Jacobs (Océ Nederland B.V.),
Ing. J. de J. Manrique (Océ Nederland B.V.).

Location: Océ Nederland B.V. Venlo,
Research & Development, DV2

The faculty of Electrical Engineering does not accept responsibility for the contents of reports on training assignments and final projects.
Acknowledgements

I would like to thank Océ Nederland B.V. for providing the possibility to perform my work at the Research Department. Especially I thank Prof.Ir. M.P.J. Stevens (EUT’s Faculty of Electrical Engineering, Section Digital Systems), Ir. J.W.M. Jacobs and Ing. J. de J. Manrique (both Océ Nederland B.V.) for their coaching and support. Furthermore I wish to thank all the people at the Research Department who helped and advised me during my work and who created a pleasant working atmosphere.

Special thanks go to Texas Instruments in the Netherlands and Germany and to Océ for granting my request to attend the TMS34020 Graphic System Design workshop in November 1990 at TI Freising, Germany.

The writer of this report was enabled by Océ Nederland B.V. to perform research, on which this report is mainly based. Océ Nederland B.V. does not accept any responsibility for the correctness of the information, studies and conclusions mentioned in this report, which are totally on account of the writer.
Summary

This report reflects the work that has been done by the author during a final project at the research department of Océ Nederland B.V. in Venlo, the Netherlands. The final project forms the last part of the author's study in Electrical Engineering at the Eindhoven University of Technology.

The subject of the final project comprises the design of a Raster Image Processing unit, which forms a part of laser printer front end systems. The design should be based on the TMS34020 Graphics System Processor and it should meet the latest requirements in functionality, flexibility and performance. The resulting specification of the design must be suitable for an implementation in which programmable logic can be used, such as LCA’s, FPGA’s or maybe even “real” ASICs.

After a general description of the functionality of a Raster Image Processing unit the criteria for a thorough evaluation of the TMS34020 are derived. Based on these criteria the 34020 is evaluated on both hardware aspects and instruction set aspects. It is concluded that the 34020 has strong graphics capabilities that however only can be fully exploited by using the specific graphics instructions from the instruction set. Specifically it is concluded that the 34020 can be used for the Raster Image Processing unit. Especially a comparison with the processor’s predecessor, the TMS34010, shows that the use of the 34020, in upgrades for 34010 based systems, can significantly enhance performance.

Using a 34010 based design as a start-off point, a concise list of requirements for a 34020 based design is derived. In order to meet these requirements alternative architectures for the system’s memory, including implications for implementation, are compared. Furthermore the method of sharing this memory between the 34020 and a bitmap reader with high performance is developed. It is concluded that the system is best designed as a multiprocessing system, with the 34020 and the bitmap reader sharing the same system bus and one contiguous memory. The memory has a basic size of 16 Mbyte (32 4Mbit DRAMs), and is expandable by piggy backed memory blocks. To enhance the performance of memory sharing the concept of scanline buffering has been developed.

Next in this report, the interface specification of the bitmap reader is derived, together with a description of its global behavior. Then the top level architecture of the Raster Image Processing unit is drawn up. From this point the report concentrates on the further specification of the bitmap reader. The Viewlogic CAE environment and Viewlogic’s version of VHDL are used to decompose the bitmap reader into functional blocks at a lower hierarchical level. Then, each of these blocks is specified with use of state machines and Boolean equations. From these specifications the blocks are described in VHDL and verified by functional simulation. Furthermore the total bitmap reader is verified. It is concluded that the bitmap reader’s specification is complete and functionally correct.

Finally logic synthesis of the bitmap reader’s VHDL blocks is carried out, using Xilinx Logic Cell Array (LCA) technology. The resulting logic is verified to be correct and after integration of the synthesized blocks the bitmap reader is then fully specified by it’s interfaces, the decomposed architecture of functional blocks and the gate level designs of each of these blocks.

The last part of the report describes placement and routing of the bitmap reader into a XC3064 LCA. The design, which is 189 CLB’s in size, is succesfully placed and routed. However simulation shows that the design does not meet the maximum allowed timing delays. Recommendations for a more successfull implementation trajectory are given.

Generally it is concluded that the TMS34020 can very well be used in a Raster Image Processing unit. Furthermore the resulting top level Raster Image Processing unit design is able to meet the functional requirements. It is advised to follow this design when the unit is implemented in hardware. Although straightforward implementation of the bitmap reader was not immediately successful, the full functional specification of the bitmap reader forms a framework for various alternative implementations.
# Table of contents

1 **Introduction** 1

2 **High speed laserprinter front-end** 3

2.1 **Basic architecture** 3

2.2 **Functionality of the Raster Image Processing unit** 3

2.3 **Criteria and means for processor evaluation** 4

3 **TMS34020 hardware description** 5

3.1 **Architectural overview** 5

3.2 **Registers and cache** 6

3.3 **Host interface** 9

3.4 **Local memory interface** 10

3.5 **Multiprocessor interface** 12

3.6 **Memory map and addressing mechanisms** 13

3.7 **Hardware supported data structures** 14

4 **Instruction set features** 16

4.1 **General purpose characteristics** 16

4.2 **Graphics instructions** 16

4.3 **Conclusions on the instruction set** 17

5 **Software development for the TMS34020** 19

5.1 **Drawing a thick line** 19

5.2 **The TMS34020 Software Development Board** 20

5.3 **Code generation tools** 20

5.4 **High level language debugger** 22

5.5 **Running the thick line benchmark** 23

6 **Conclusions on the evaluation of the TMS34020** 28

6.1 **Hardware** 28

6.2 **Software and tooling** 28

6.3 **Comparison with the TMS34010** 29

7 **Design requirements for the RIP unit** 30

7.1 **Analysis of a 34010 based RIP unit design** 30

7.2 **Sizes of bitmaps** 32

7.3 **Memory size requirements** 33

7.4 **Reader functional requirements** 35

8 **Memory architecture alternatives** 36

8.1 **Sharing of the memory** 36

8.2 **Architecture with separate banks** 36

8.3 **16 Mbyte architecture; the concept of scanline buffering** 41

8.4 **Memory architecture choice** 42

9 **Further system issues** 44

9.1 **Guaranteeing refresh** 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2</td>
<td>Clearing on demand</td>
<td>44</td>
</tr>
<tr>
<td>9.3</td>
<td>XY-addressing</td>
<td>45</td>
</tr>
<tr>
<td>9.4</td>
<td>Mirroring</td>
<td>45</td>
</tr>
<tr>
<td>9.5</td>
<td>TMS34082 coprocessor</td>
<td>45</td>
</tr>
<tr>
<td>10</td>
<td>Using FIFOs for scanline buffering</td>
<td>46</td>
</tr>
<tr>
<td>11</td>
<td>Bitmap reader interface derivation and global behavior</td>
<td>50</td>
</tr>
<tr>
<td>11.1</td>
<td>Programmability</td>
<td>50</td>
</tr>
<tr>
<td>11.2</td>
<td>Bus control interface</td>
<td>50</td>
</tr>
<tr>
<td>11.3</td>
<td>Engine interface</td>
<td>50</td>
</tr>
<tr>
<td>11.4</td>
<td>DRAM interface</td>
<td>51</td>
</tr>
<tr>
<td>11.5</td>
<td>FIFO interface</td>
<td>51</td>
</tr>
<tr>
<td>11.6</td>
<td>Clock inputs, interrupt and reset</td>
<td>51</td>
</tr>
<tr>
<td>11.7</td>
<td>Reader block diagram and flow chart</td>
<td>51</td>
</tr>
<tr>
<td>12</td>
<td>Top level system architecture</td>
<td>53</td>
</tr>
<tr>
<td>12.1</td>
<td>Hierarchical design methodology using Viewlogic CAE env.</td>
<td>53</td>
</tr>
<tr>
<td>12.2</td>
<td>The architecture</td>
<td>53</td>
</tr>
<tr>
<td>12.3</td>
<td>The TMS34020</td>
<td>54</td>
</tr>
<tr>
<td>12.4</td>
<td>System memory, mirror function and transceiver</td>
<td>54</td>
</tr>
<tr>
<td>12.5</td>
<td>Address latch and decoder</td>
<td>54</td>
</tr>
<tr>
<td>12.6</td>
<td>Bitmap reader and FIFOs</td>
<td>56</td>
</tr>
<tr>
<td>12.7</td>
<td>Design notes</td>
<td>56</td>
</tr>
<tr>
<td>13</td>
<td>Bitmap reader architecture</td>
<td>58</td>
</tr>
<tr>
<td>13.1</td>
<td>A detailing step</td>
<td>58</td>
</tr>
<tr>
<td>13.2</td>
<td>Bitmap reader programmable control</td>
<td>59</td>
</tr>
<tr>
<td>13.3</td>
<td>Writing synthesizable VHDL code</td>
<td>61</td>
</tr>
<tr>
<td>13.4</td>
<td>Reader decoder</td>
<td>65</td>
</tr>
<tr>
<td>13.5</td>
<td>Requester</td>
<td>67</td>
</tr>
<tr>
<td>13.6</td>
<td>DRAM controller</td>
<td>69</td>
</tr>
<tr>
<td>13.7</td>
<td>FIFO controller</td>
<td>72</td>
</tr>
<tr>
<td>13.8</td>
<td>The architecture schematic</td>
<td>76</td>
</tr>
<tr>
<td>14</td>
<td>Implementation trajectory</td>
<td>78</td>
</tr>
<tr>
<td>14.1</td>
<td>Logic synthesis using the XI3000 library</td>
<td>78</td>
</tr>
<tr>
<td>14.2</td>
<td>Porting to an XC3064-100 LCA</td>
<td>79</td>
</tr>
<tr>
<td>14.3</td>
<td>Result</td>
<td>79</td>
</tr>
<tr>
<td>14.4</td>
<td>Recommendations</td>
<td>80</td>
</tr>
<tr>
<td>15</td>
<td>Conclusions</td>
<td>81</td>
</tr>
<tr>
<td>Literature</td>
<td></td>
<td>82</td>
</tr>
<tr>
<td>Appendix A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appendix B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appendix C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appendix D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appendix E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appendix F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. Introduction

In recent years the presentation of information by means of documents with high quality text printing and incorporated graphics has enormously gained popularity. This way of presentation is referred to as Desktop Publishing (DTP) and its popularity has resulted in a massive demand for it by the public market as well as by the office automation market. This demand causes an extensive growth of the need for laserprinter systems where medium to high volume printing, high speed and high quality are becoming the keywords for the years to come. It is therefore understandable that much research effort is done by printer manufacturers in the area of high speed laserprinter architectures.

When a high speed laserprinter system is broken down one roughly recognizes three subsystems. See figure 1.1. First there’s the source or printer server that generates code (in some Page Description Language or PDL) to describe the page or document to be printed. Then the second subsystem, called the high speed front-end, takes the source code and converts it to image data. The third subsystem is the printer’s back-end or printer engine which takes the image data and puts it on paper.

![Figure 1.1: laser printer system](image)

It is the high speed front-end (HSF) architecture that is of interest for this report. One of the problems to be solved in research on HSF’s is the way of implementation of the so-called Raster Image Processor (RIP) part. The first key issue is to be able to make a choice for the type of processor that is suited for the job on a solid basis. Among the possibilities are solutions that use one or several parallel transputers, RISC or CISC processors or Graphic System Processors (GSP’s). This report will deal with this choice-problem by evaluating a new type of GSP, the TMS34020 by Texas Instruments (chapters 3 to 6). The results will have to establish a means for decision-making in designing high speed front-ends.

The second key issue of the problem of implementing the RIP part is to design an architecture for it, based on a chosen processor, that meets the latest and future demands in volume, quality, speed and cost. After the evaluation of the TMS34020 this report deals with the design of a RIP and a bitmap reader architecture, together known as a Raster Image Processing Unit (RIPU) to constitute a solution to this second issue.

As a start-off point a RIPU architecture based on TI’s TMS34010 will be studied (chapter 7). Then, after establishing the design requirements for the RIP unit architecture, the unit’s memory archi-
tecture will be designed. A method of sharing this memory between the TMS34020 and the bitmap reader will be developed including an implementation with FIFOs (chapters 8 to 10). After the derivation of the interfaces of the bitmap reader (chapter 11) a top level architecture for the RIP unit can be designed (chapter 12). Finally, a structural design method using the Viewlogic CAE environment and VHDL [VUSR], [VDES] will be applied to fully specify the bitmap reader (chapter 13). To end with, a possible implementation trajectory for the bitmap reader, based on LCA’s, is treated in chapter 14.

The two problems, tackled in this report, form the author’s final project assignment. Figure 1.2 illustrates the scope of the assignment in its context to other research.

![Diagram](figure1.2.png)

From figure 1.2 it can be seen that similar evaluation research is performed regarding other types of processors. The evaluation forms the first part of the total assignment, the design of a high speed front-end Raster Image Processing unit based on the TMS34020.
2. High speed laserprinter front-end

Before starting the evaluation of the TMS34020 it seems wise to describe a high speed front-end and the major functional blocks that it consists of. The basic architecture will be given. Then the Raster Image Processor Unit’s functionality will be treated in more detail. After this one will have a clear view on the tasks for the TMS34020 when it is applied to implement a RIP unit. As a result it will be possible to state the criteria for the evaluation of the TMS34020.

2.1. Basic architecture

Figure 2.1 shows the basic architecture of a high speed front-end.

From figure 2.1 it can be seen that within the high speed front-end the interpreter takes care of the Page Description of the document or page. For this description the source uses a Page Description Language (PDL), for instance PostScript [POST] or PCL. The result of the interpreters action is a stream of graphical primitives passed on the Graphical Primitives Interface (GPI). The Raster Image Processor rasterizes these primitives and places the data into a bitmap. Finally the bitmap reader (BMR) gets the rastered data, transforms it to image data and sends it to a printer engine. The RIP, the bitmap and the BMR together form the Raster Image Processing Unit.

2.2. Functionality of the Raster Image Processing Unit

This paragraph treats a general basic functional description of a Raster Image Processing Unit. In general the functionality of a Raster Image Processing Unit can be described as follows. When the interpreter (figure 2.1) has finished the interpretation of a page it signals the RIP. The RIP then takes ownership of the bitmap and starts reading graphical primitives from the GPI. According to each primitive the RIP accesses one of its rasterization algorithms and it places the primitive in the bitmap.

When the bitmap has been filled the RIP signals the bitmap reader that it can start reading out the bitmap. The BMR takes over ownership of the bitmap. When the printer engine has signalled to be
ready the BMR reads the rasterdata which it sends as image data to the engine. After the bitmap has been transferred the BMR signals the RIP that it is ready, releasing ownership, and the BMR starts waiting for a new signal from the RIP. The time the BMR is active is called page-time, whereas its waiting period is often referred to as interpage-time.

Suppose the bitmap memory is able to only contain one page at a time. Then it is clear that only during interpage-time the RIP is able to fill the bitmap. During page-time the RIP must be inactive on the bitmap memory. However the Raster Image Processor can communicate with the interpreter to already receive new primitive data which the RIP can preprocess in order to prepare for filling during the next interpage-time. Design requirements determine the precise way in which the RIP performs its tasks in time. For instance when performance has to be enhanced a designer can think of a bitmap memory architecture that allows bitmap filling also during page-time. Later on this observation will be used in the redesign of the RIP unit.

2.3. Criteria and means for processor evaluation

Now that the RIP unit is generally treated it is possible to state criteria for evaluating the TMS34020 Graphics System Processor. Due to the former analyses the criteria can be very application specific.

The desire to use the TMS34020 for design of the RIP firstly necessitates an evaluation on the hardware architecture of the GSP. Considering the fact that the most important activity for the TMS34020 will be the filling of bitmaps it is necessary to evaluate its competence to execute graphic primitive rasterization algorithms. This includes both hardware supported datastructures and instructionset features for graphics. Furthermore the supported memory architectures and the speed of the processor are very important for the filling of a bitmap.

As a consequence of having to communicate over the GPI-interface and with the bitmap reader interfaces and system bus structures of the TMS34020 have to be dealt with during the evaluation. Also general purpose processing capabilities must be evaluated as an implication of the possibility to allocate additional processes on the GSP. Finally tooling and growpath of the TMS34020 have to be considered.

The criteria can be summarized as follows:

- hardware architecture
- memory structure
- hardware support for bitmapfilling
- instructionset features for graphic algorithms
- speed
- tooling
- growpath

The evaluation will start with a specific description of hardware and instruction set. Then theory and design of a graphical primitive algorithm will be treated. As means for evaluating the TMS34020's specific graphics characteristics a benchmark, based on this algorithm is run on the processor.
3. TMS34020 hardware description

This chapter will describe the TMS34020 hardware. It treats the architectural overview, registers and instruction cache and the various interfaces for local memory (DRAM and VRAM), host- and multiprocessing. Additionally the memory organization along with addressing mechanisms are treated. Specific hardware supported datastructures for graphics will be dealt with.

3.1. Architectural overview

The TMS34020 is a full 32-bit processor with hardware support for graphics operations. Also a full set of general purpose instructions is included that can support high-level languages. The processor can operate at 32 or 40 MHz, having a instruction cycle time of 125 or 100 ns, with each instruction cycle consisting of four clockcycles. Due to its 32-bit architecture the bit addressable address range is 4 gigabit or 512 Mbyte. The best way to quickly get an impression from the TMS34020 is to look at its blockdiagram. See figure 3.1.

![Block diagram of TMS34020](image)

In figure 3.1 the processor’s core can be recognized. It consists of the Arithmetic Logical Unit (ALU), the Program Counter (PC), the Status Register (ST), a 32-bit barrel shifter and two register files A and B. To speed up instruction execution a 512-byte instruction cache, utilising a Least Recently Used (LRU) mechanism, is implemented on-chip. The cache gives a 10 MIPS peak performance for iterative loops because when the loop’s code is in cache the processor can execute one of the loop’s instructions every 100 ns without having to execute memory access cycles.

Instructions are decoded out of the cache and interpreted by microcontrol ROM. 32-bit internal data-paths interconnect the functional blocks and connect to the 32-bit multiplexed address/data bus, the Row/Column Address bus (13 bit multiplexed) and the 32-bit Host Address bus via a buffer, a multiplexer and a host address latch respectively. Furthermore there’s a functional block that takes care of interrupts and reset.
Sofar this architecture is quite common and straightforward except from the on-chip instruction cache. On chip however there are more functions added that make the TMS34020 attractive for graphics system solutions. For instance it has an on-board local memory controller and bus timing unit that connects to the DRAM and VRAM interface, the bus interface and the bus arbitrator. This simplifies the implementation of DRAM and VRAM memory. The controller is designed to optimize memory access. It automatically performs bit alignment and masking necessary to access data located at arbitrary bit boundaries. Furthermore it operates autonomously and it has a write queue of 1 to 32 bits deep that allows completing a memory write cycle without delaying instruction execution. Burst (page mode) accesses are supported by the control signals.

The TMS34020 has several control signal groups for the host interface, the multiprocessor interface, the emulator interface, and for video timing and control. Finally on-chip functions include sixty-four programmable I/O registers which are memory mapped and control CRT-timing, communication with the host, interfacing to local memory, interrupts and parameters required by some graphics-drawing instructions. A description of the individual signal pins can be found in [GSPUSR].

3.2. Registers and cache

In this paragraph a closer look is taken to the TMS34020 registers and their individual functions. A division into two groups can be made. The first group consists of the CPU registers and the cache. The second group is formed by the memory-mapped I/O registers.

Registerfiles A and B, stackpointer

In the processor core there are two registerfiles, A and B, of fifteen 32-bit registers each. Furthermore there's the system stack pointer (SP) which is addressed in both registerfiles as a sixteenth register. Boolean, arithmetic, pixel processing, byte and field (data of arbitrary bitsize between 0 and 32 bits) move instructions operate on the data in the registerfiles. Registerfile A contains general purpose registers to support high level programming languages such as C that can use registers for variable storage. Registerfile B contains registers that are assigned hardware-dedicated functions during pixel operations, controlled by graphics instructions execution. Pixel operations, such as PixBLT, will be described in detail in paragraph 4.2 (Graphics instructions). The B-registers have to be loaded with parameters before pixel operations can take place. Every register B0 to B14 has its specific function. For instance B0 (SADDR) holds the source pixel address, B2 (DADDR) the destination pixel address, B7 (DXDY) holds the length and width of a pixel array, and so forth. Whereas these registers are implied operands for pixel operations they can be used as general purpose registers for data storage and manipulation when no pixel operations are executed. Figure 3.2 shows registerfiles A and B and the stackpointer.
The stack pointer keeps track of the top of the system stack. This stack is used for storing the program counter and status register during the execution of interrupt service routines or subroutines. The stack grows in the direction of smaller addresses.

**Program counter and status register**

Other CPU registers are the PC and the status register. The status register contains information that results from instruction execution. See figure 3.3 for the contents of the status register.

The TMS34020 allows the addressing of fields of arbitrary size between 1 and 32 bits. The size of two fields, field 0 and field 1, can be specified in the status register in the five bit fields FS0 and FS1. When a field is loaded into a register it is right justified and is extended with ones or zero's according to FE0 and FE1. The IE bit is the interrupt enable. SS when set to one allows single stepping. BF
is set when a bus fault occurs. Finally IX is the interruptible instruction executing bit that indicates with a one that an interruptible instruction was executing when an interrupt occurred. Information like N(egative), C(arry), Z(ero) or (o)V(erflow) can be used to perform tests and direct program execution.

An important CPU register is the program counter (PC). The PC points to the next instruction to be fetched out of the cache or the local memory. Its four least significant bits are zero so addressing of instructions always takes place on word boundary.

**Cache operation**
The 512-byte instruction cache realizes unimpeded access to instructions and is transparent to software. The cache is partitioned into four 128-byte segments, numbered 0 to 3. Each segment has a 22-bit segment start address (SSA) register associated with it that identifies the address section in memory that correspond to the contents of the cache segment. Indeed with the 22 SSA bits 4 million different memory segments can be accessed. 4 Million 128-byte sections make up the total of 512 Mbyte memory. Each cache segment is further divided into eight subsegments of four long-words (32 bit). Every subsegment has a flag P to indicate whether the subsegments data is valid. A LRU-stack of four places of two bits is used to implement the LRU cache replacement algorithm.

The CPU can only get memory data from the cache that is associated with the PC so opcodes, immediate operands and absolute addresses. Instructions and data can be in the same part of memory and therefore also data can be copied in the instruction cache. The cache however is bypassed when data is to be fetched from memory.

When the CPU fetches an instruction it compares the 22 MSB’s of the address to the four SSA registers to see if the addressed section of memory is in the cache. When a match is found the three subsequent address bits are used to select a subsegment from the cache. When the P-flag indicates that the data in the subsegment is valid the next two address bits select one of the four long-words. This is called a cache hit and the instruction is fetched from the cache.

When a cache hit occurs the number of the used segment is put on top of the LRU-stack, pushing the other segment numbers down. The top of the stack will always contain the Most Recently Used segment whereas the Least Recently Used segment is at the stacks bottom. Whenever a cache miss occurs due to not finding a match with one of the SSA-registers (segment miss) the bottom of the LRU-stack is read to get the number of the cache segment whose code can be discarded. All of the eight subsegment P-flags are cleared and the SSA-register’s contents are replaced with the 22 MSB’s of the requested address. The four-word subsegment in memory that contains the requested instruction is read into the cache and placed in the appropriate subsegment. The P-flag of this subsegment is set, the number of the new segment is transferred from the LRU-stack bottom to the stack top and the instruction is read from the cache.

A cache miss due to a P-flag indicating non-valid data in a subsegment (subsegment miss) causes the subsegment in memory that contains the requested instruction to be read into the cache. The flag is set, the segment number is placed on the top of stack and the instruction is read from the cache.

It is possible to flush the cache, clearing its contents and all the flags. Furthermore the cache can be disabled. Using the cache enhances instruction execution considerably as explained in section 3.1. Especially when executing program loops the cache is an advantage to the GSP’s speed. See [GSPUSR] for a schematic of the instruction cache.

**I/O registers**
The TMS34020 supports a set of memory mapped I/O registers that support:

- Communication between the TMS34020 and a host processor
■ TMS34020's interface to local memory
■ Interrupts
■ Video timing and screen refresh
■ Graphics drawing operations

The I/O registers are accessed like any other memory location and reside in on-chip memory. They occupy locations C000 0000h through C000 03FFh. See [GSPUSR] for a detailed description of the registers.

### 3.3. Host interface

Now that the processor's core has been treated the different interfaces of the TMS34020 and the control signals can be dealt with. In this part the host interface is described.

The interfacing of the TMS34020 with a host is shown in figure 3.5.

![System with TMS34020 and host](image)

**Figure 3.5: System with TMS34020 and host**

HA5-HA31 form the address of a 32-bit long word in local memory space. With HBS0-HBS3 (Host Byte Select) the host can address a specific byte within the long word. HCSn is an active low chip select to indicate that the host wants to communicate with this particular TMS34020. With HREADn and HWRITEn the host indicates its request to read or write from the TMS34020's local memory. The data is latched in bidirectional transceivers. HRDY informs the host that a host-initiated access will complete. HOEn (Host Output Enable) is designed to enable a transceiver to output data to the TMS34020 local memory bus. HDST (Host Data Strobe) acts as a strobe for a latch during a host read from the local memory bus.
The I/O registers associated with the host interface are two 16 bit registers HSTCTLL and HSTCTLH. They are necessary to completely specify the communications between host and Graphic System Processor (GSP). For compatibility with TMS34020’s predecessor, the TMS34010, the registers HSTADRH, HSTADRIL and HSTDATA are added in the memory map.

In figure 3.5 one can recognize that the data flow takes place outside of the TMS34020, only the address is passed to the processor. This is a significant different way of interfacing compared to the host interface mechanism used by the TMS34010. The TMS34010 passes the data via the HSTDATA I/O register.

The protocol for communication between host and GSP is a handshake based on the read or write request and the HRDY signal. Using this mechanism it is possible for the host to approach the maximum bandwidth of the TMS34020 local memory interface. A read or write without wait states takes two machine cycles of 100 ns. Such an access of 200 ns yields a throughput of 160 megabits per second.

3.4. Local memory interface

In figure 3.6 the local memory interface is shown.

The interface consist of two separate busses which are used to communicate with the memory or memory mapped peripherals. First there’s the multiplexed local address/data bus (LAD bus). The second bus is the row/column address bus (RCA bus) that is used for interfacing to DRAM/VRAM. Associated with the buses are the control signals. See for their description figure 3.6. and [GSPUSR].

The I/O registers belonging to the local memory and DRAM/VRAM interface are:

- CONFIG: system configuration
- PMASK: plane mask
- REFADR: refresh address
- BSFLTD: bus fault data
- BSFLTST: bus fault status
The CONFIG register is important because it holds the Row Column Mode (RCM) bits. These bits, RCM0 and RCM1, must be set according to the size and architecture of the used DRAM device.

**Memory device flexibility**

During a DRAM or VRAM access via the 13-bit RCA bus first the row address is put on the bus lines. When the Row Address Strobe (RASn) has been asserted low shortly afterwards the column address is output. Then the Column Address Strobe (CASn) signal(s) are passed to the memory devices. Finally data is transferred via the LAD bus. During the time that the row address is on the RCA bus the LAD bus contains the 32 bit linear address. In case of a write to memory, when the column address is on the RCA bus, the LAD bus contains data to be transferred. The CASn signals become active well before the LAD bus data will be replaced by the address of the next access. The LAD bus should be used when Static RAM or peripherals are addressed.

To provide flexibility in memory architecture and to allow mixing of different generations of DRAM and VRAM devices the RCA bus can provide access to 64KxN, 256KxN, 1MxN and 4MxN DRAMs. When the above mentioned RCM bits are set properly device connection and mixing is straightforward without additional logic besides the bank selection decode. The RCM bits should be set to match the smallest DRAM device in the system.

According to the Column Address Mode (CAMD) input logical level different memory device mixtures can be applied. This input can be decoded from the local address decode/memory bank decode. The advantage of determining the smallest device architecture with the RCM bits and the device combination with the CAMD signal is that the total memory architecture can be organized in a way that all devices use the same RCA bus signals. For details refer to [GSPUSR].

**Memory cycle examples**

To get an impression of the timing of memory accesses two example timings will be given. Firstly a general read/write cycle timing diagram is given. Then a page mode access is illustrated. See figure 3.7.
Local-Memory Cycle

As can be seen page mode speeds up data throughput due to the fact that RASn can be kept low while new column addresses and CASn signals are supplied to the DRAM or VRAM. Subsequent accesses now take one machine cycle of 100 ns. The PGMDn-signal informs the processor when page mode accesses are possible with the device under access.

3.5. Multiprocessor interface

As can be seen from figure 3.1 the multiprocessor interface consists of three signals: Gin, R0n and R1n. The interface supports a general protocol that can be used with external arbitration logic to form a system of multiple processors sharing a common local memory space.

Performance can increase when using multiple TMS34020s that share local memory. This is because the internal cache permits long time intervals during program execution when no external memory accesses are required. This available time can be used by other GSP’s to access memory.

Of course at a certain number of processors the total memory bandwidth will be used and adding more processors is useless. The expected number of processors that form an economical system is three, although this is dependant on the application [GSPUSR].

The Gin signal, when low, informs the GSP that it is master of the bus. The R0n and R1n signals are used by the GSP to encode a desired priority with its bus request.
3.6. Memory map and addressing mechanisms

Although the TMS34020 is a bit addressable machine that can address 4 gigabit, memory is physically accessed 32 bits at a time. The memory map in figure 3.8 therefore shows locations as 32 bit long words.

The memory is divided into several regions. From address 0h to BFFF FFE0h and from C000 2000h to FFFF DFE0h there is space for general use such as executable code storage and data storage. Addresses C000 0000h through C000 03E0h are reserved for the I/O registers, described in section 3.3. 34 long words are reserved from FFFF FBC0h to FFFF FFE0h for the interrupt, reset and trap vectors. Extended trap vectors can be stored as shown in figure 3.9. Finally there is reserved memory from C000 0400h to C000 1FE0h for future expansion of the I/O registers.

As far as addressing is concerned the TMS34020 physically performs linear addressing of long words. However for graphics instructions, pixel processing and window checking and clipping so-called XY-addressing is supported so one can address memory according to a coordinate system. When XY-addressing is used the GSP will convert the addresses to a linear format before actually accessing memory. First linear addressing is treated, then XY-addressing and converting XY-addresses to linear addresses will be described.

linear addressing
The total address range is from 0h to FFFF FFFh with each address pointing to an individual bit. This bit addressable range is called the logical address range. The TMS34020 addresses memory over the 32 bit LAD bus and reads a complete 32 bit long word every access. Writes may be 8-, 16-, 24- or 32-bit values by using the appropriate CASn strobes.Accesses always start on 32 bit boundary so the 5 LSBs of a 32-bit starting address are always zero. The 4 LSBs of an address do not appear on the LAD bus and bit 4 is output only when 16 bit wide memory is used. Instead when the GSP addresses a field within a 32 bit long word the 5 LSBs are used internally so masking and bit alignment can be performed automatically to extract the field from memory. Similarly, read-modify-
write cycles with masking and shifting is performed internally when a write is done to a field within a long word.

**XY-addressing**

As mentioned before XY-addressing simplifies graphic operations for the user. For example a pixel can be addressed by its coördinates on a display device. When a general purpose register contains an XY address the 16 MSBs contain the 16 bit right-justified signed Y coördinate as a signed integer. The 16 LSBs contain the X coördinate as a right-justified signed integer. To convert an XY-address to a 32 bit linear address the TMS34020 uses four parameters:

- Logical pixel size: amount of bits that represents a pixel, always a power of two between 1 and 32 bits. Size defined in PSIZE register.
- Pitch conversion factor: defined in CONVSP, CONVDP, or CONVMP I/O registers.
- Actual pitch, linear difference in memory addresses of two vertically adjacent pixels in a pixel array, defined in SPTCH, DPTCH or MPTCH (B register file).
- Offset: specifies the XY-origin, in the OFFSET B register.

The GSP uses the following formula to convert the address:

\[
\text{address} = \left( \left( Y \times \text{display pitch} \right) + \left( X \times \text{pixel size} \right) \right) + \text{offset} \quad (1)
\]

In (1) **address** is the linear address to be calculated. Y and X are the Y and X coördinates. The display pitch is the difference in addresses of two vertically adjacent pixels on the display device.

Because the pixel size is always a power of two the multiplication of the X component is done with a shift operation by the barrel shifter. This barrel shifter is able to performs shifts of 1 to 32 bits in one machine cycle.

Multiplying the Y component is somewhat more intricate. When the pitch is a power of two or the sum of two powers of two the amount of shifting (pitch conversion factor) is contained in the CONVxP register. A pitch of two powers of two adds one machine cycle to the conversion. Pitches however may be arbitrary. In case of an arbitrary pitch the calculation cannot take place by shifting but a full 16-bitx32-bit multiplication has to be performed using the actual pitch value from the appropriate xPTCH register. This adds about 12(!) cycles to the conversion.

### 3.7. Hardware supported data structures

The last part of this chapter describes the datastructures that the TMS34020 supports by hardware. These are:

- fields
- pixels
- pixel arrays

**fields**

As was stated with the description of the status register the GSP supports two field data structures that are defined by FS0, FS1, FE0 and FE1 in the status register. The starting address of the field is the long word address on the LAD bus and the 5 LSBs of the total address.

In a worst case a field straddles a 32-bit long word boundary and neither the fields start or end is aligned to long word boundaries. An insertion of such a field takes two read-modify-write cycles. Field extraction and insertion is performed by internal logic and is transparent to software.
pixels
The pixel data structure is defined by the starting address (the 5 LSBs) and the pixel size. Pixels are treated as a special kind of fields that are constrained to sizes of powers of two within the range of 1 to 32 bits. This causes pixels to never straddle long word boundaries and it causes long words always to contain an integral amount of pixels. When pixels are loaded into a general purpose register they are right-justified within the register and extended with zeroes.

To insert a pixel of less than 8 bits requires two memory cycles, one read and one write, two machine cycles at the least each. Inserting 8-, 16- or 32-bit pixels takes a single write unless plane masking is enabled. Extracting pixels always takes one read cycle.

pixel arrays
Figure 3.9 shows how a two-dimensional pixel array can be thought of.

```
<table>
<thead>
<tr>
<th>0</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>
```

The pixels of a row are packed together into adjacent memory locations. Adjacent rows aren’t usually stored in adjacent places in the memory. If for instance the pixel array resides in display memory rows are separated by the display pitch. The amount of separation of vertically adjacent pixels within the actual array is called the array pitch.

The pixel array data structure is defined by the width DX, the height DY, the display pitch and the starting address. The starting address can be specified in terms of either XY coordinates of the starting pixel (referred to as an XY-array) or the linear address of the starting pixel (linear array). Pixel block transfer actions support both XY- as linear pixel arrays.

The next chapter will treat the TMS34020 graphic capabilities caused by the instruction set features. Also some general purpose characteristics will be described.
4. Instruction set features

Chapter 4 goes into detail about the TMS34020 instruction set. This description will demonstrate the GSP's specific capabilities for graphics operations and general purpose processing. Though the intention is to use the GSP in the design of hardware for a Raster Image Processing unit the insight gained by studying some software aspects will be of substantial benefit. It will contribute to judging the TMS34020 and enhancement of knowledge needed for the system design.

4.1. General purpose characteristics

General purpose tasks comprise arithmetic operations, boolean operations, manipulation of variables of various types such as integers, floats, arrays and so forth by moving them and perform arithmetic on them. Another general purpose characteristic is often recognized in the ability of a processor to execute conditional loops, calls to subroutines, jumps, and interrupt handlers. Furthermore communication with other processors, hosts, peripherals, which can be seen as general I/O is part of general purpose operation.

In the instruction set of the TMS34020 there's a subset of instructions that enable the GSP to perform general purpose tasks. Firstly the instruction set contains a variety of MOVE instructions that allow moving data from memory to the GSP's register files and vice versa. Also register to register and memory to memory moves are possible. Setting bits in specific registers for I/O is also possible with moves. Secondly unconditional and conditional jumps, calls and returns allow for control of the program flow and implementation of loops. Finally a large set of arithmetic and boolean instructions is furnished including add, subtract, divide, multiply, and, or and the like.

Due to this general purpose abilities of the TMS34020 it is possible that when this GSP is used in the RIP unit the activities are not constrained to getting primitives from an interpreter and placing them in bitmap memory. The general purpose characteristics allow for extra functionality such as some final interpretation of higher level primitives that need further calculations, preprocessing, or management of datastructures such as a font cache.

4.2. Graphics instructions

As was seen in chapter 3 the TMS34020 hardware determines the characteristics of this graphics engine by supporting flexible DRAM and VRAM and video interfacing, supplying registers for drawing operands, allowing pixel block transfer operation, pixel processing and supporting datastructures such as pixel arrays.

The graphics part of the instruction set is meant to take full advantage of the hardware. A division can be made into four categories of instructions: single pixel, line, pixel array and fill instructions.

Single pixel instructions
These instructions are at the lowest level. They can be used to manipulate a single pixel in memory. An important instruction of this group is the DRAV (draw and advance) instruction. It is useful in implementing incremental drawing algorithms such as drawing circles and ellipses.

Line instructions
This category consists of instructions that draw single lines. The algorithm used to execute these instructions is the well known Bresenham [BRES] incremental line draw algorithm. The LINE instruction draws a line from which the endpoints can be specified as XY-addresses. The instruction checks if a line needs to be clipped to a defined window. A faster line instruction is FLINE, which also uses Bresenham's algorithm but needs linear addresses for the endpoints and does not allow clipping to a window.
To help initialization of implied operands for the line instructions the LINIT instruction is added to the instruction set.

**Pixel array instructions**
The pixel array instructions consist of a variety of pixel block transfer or PixBLT instructions. They allow moving two dimensional pixel arrays from a source to a destination. Linear and XY addressing is supported as well as the use of masks. Furthermore there are PixBLTs that take a one bit per pixel array and combine the zeroes with a background color and the ones with a foreground color. These operations perform color expansion. One important PixBLT is the PixBLT L,M,L. PixBLT L,M,L takes an array that contains a pattern, combines it with a mask and moves the result to a destination. In this way it is possible to place characters that have a gray pattern. This can be of advantage in laser printer applications. See figure 4.1.

![source array](source array) ![mask array](mask array) ![destination array](destination array)

**Figure 4.1: PixBLT L,M,L instruction mechanism.**

**Fill instructions**
To perform fast drawing and filling of larger areas there are several fill instructions. For instance PFILL fills an area with a pattern. Another important fill instruction is TFILL. This instruction fills a trapezoidal area by filling a series of lines in the X direction that are adjacent in the Y direction.

It is important to note that when one uses graphics instructions a general sequence is to be gone through. This sequence consists of the following steps:

- Definition of implied operands
- Execution of the graphics instruction
- Use of a graphics operation with the instruction

By definition of implied operands the specification of for instance the source pitch, the destination address or the DYDX dimensions is meant. Instructions such as LINIT help in doing this. The actual execution takes place when the graphics instruction, for instance DRAV, is encountered. During this execution several graphics operations can be performed that influence the result. These operations are from the categories: window checking, transparancy mode, masking and pixel processing.

**4.3. Conclusions on the instruction set**
The TMS34020 instruction set as now is clear combines strong graphics capabilities with general purpose possibilities. It is expected that exactly this characteristic can be of benefit when the GSP is applied in the RIP unit. With the functionality of the RIP unit in mind (paragraph 2.2) some ideas bubble up. On the one hand the graphics capabilities can be used for the filling of bitmaps or rasterization. It is clear that primitives like lines, trapezoids and characters can be "drawn" in memory easily because of the available graphics instructions.
On the other hand the general purpose capabilities come in useful when some preprocessing of primitives has to be done. One can imagine that a thick line primitive can be divided into three draw operations of filled trapezoids. The calculation for this division needs general purpose arithmetic processing power. Another preprocessing task is selecting subroutines that must be executed according to data from the primitive table datastructure. Jump and call instructions are necessary for this. The move instructions can be useful in for instance the management of rastered font bitmaps. General purpose I/O is necessary for control over the GPI communications.

Now that besides the GSP's hardware the instruction set has been treated the impression is that on both aspects the GSP has possibilities for implementation of the Raster Image Processor inside the RIP unit. Refer to figure 2.2 and 2.1. To further evaluate aspects such as execution speed of the GSP within a system and software development, chapter 5 will describe theory and design of graphical primitive algorithms. The results will be used to develop software which will be benchmarked on the TMS34020 software development board. This benchmarking will shine a light on the GSP's performance and the way software can be developed. As a specific example testing of the quality and efficiency of the GSP's C-compiler can be mentioned here.
5. Software development for the TMS34020

This chapter describes experiences that have been gained about the software development for the TMS34020 by using the Software Development Board (SDB) that Texas Instruments has designed for these purposes. A C program has been written that places thick lines in a bitmap. With this piece of software the C-compiler belonging to the 34020 can be evaluated. An optimized version of the C-routine that uses in line assembly to call upon specific rasterization instructions will also be treated in this chapter. A conclusion of the effect of this optimization will be given. Also a conclusion about the development environment can be given after the activities with the system are fulfilled.

5.1. Drawing a thick line

In rastergraphics graphical primitives are represented by specific points in a coordinate system. The marking of the points practically can be viewed as turning on pixels on a video screen or printing dots on a laser printer page. In these practical systems this rasterization happens linewise. The surface on which rasterization takes place is divided into scanlines. To rasterize on the whole surface the lines are scanned one after the other and the appropriate pixels are turned on within the scanline. On a screen it is the electron beam that scans the screens surface and turns on the picture’s pixels, in laser printers it is the laser that scans the page’s surface and specifies the dots that have to be printed.

A thick line can be regarded as a graphical primitive. Drawings can include thick lines. Also many geometric forms can be divided into thick lines of any shape by an interpreting process. These thick lines that make up the desired form can then be printed. The PostScript [POST] page description language, which is one of the most important and widely used description languages, supports the use of the thick line. In PostScript a thick line can be characterized by two coördinates and a width. Figure 5.1. shows a thick line in the first quadrant together with scanlines.

![Figure 5.1: A thick line](image)

When the thick line is regarded in figure 5.1, it is seen that rasterizing it means filling up the scanlines with pixels between an upper and a lower Y-coördinate for every X-coördinate within the line's
area. The C-software that has been written for rasterization of thick lines is based on an algorithm that sequentially performs this scanline filling for incrementing values of $X$ [HEUV].

Firstly the algorithm has to calculate the corner coordinates $(X_1, Y_1), (X_2, Y_2), (X_3, Y_3)$ and $(X_4, Y_4)$. For the first part of the thick line ($X = X_1$ to $X = X_2$) the upper coordinates of the parts of the subsequent scanlines to be filled are determined by the straight line between $(X_1, Y_1)$ and $(X_3, Y_3)$. The lower coordinates are determined by the line between $(X_1, Y_1)$ and $(X_2, Y_2)$. Determining the right upper and lower pixels within the scanline uses the equation of these lines and a line rasterization algorithm. The reader is referred to the publications on the well known Bresenham algorithm and the runlength algorithm [BRES]. The thick line's rasterization goes on using the two parallel lines in the second part ($X = X_2$ to $X = X_3$). Finally the third part is rasterized ($X = X_3$ to $X = X_4$).

The algorithm for rasterization has already been implemented in OCCAM [HEUV], a programming language used in parallel processing. It has been converted into C for the evaluation of the TMS34020. For the code see appendix A.1.

5.2. The TMS34020 Software Development Board

The Software Development Board [SDB] is meant as a tool for programmers writing application software for the TMS34020. It can be viewed as a high-performance AT-bus compatible graphics card. It features a 32 MHz version of the 34020, 1 Mbyte of zero wait state Video RAM, 1 Mbyte of zero wait state Dynamic RAM, a 32 bit local data bus. Screen resolutions are selectable from 640 x 480 pixels, 16 colors to 1024x769 pixels, 256 colors.

A typical environment consists of an IBM PC or compatible, a separate high resolution graphics monitor (i.e. a second monitor that is driven by the board) and of course the SDB itself correctly installed in the PC. For more detail refer to [SDB]. The development system further consists of a C-compiler/optimizer, an assembler and linker. Furthermore a third party high level language debugger [PONS] is supplied to run on the PC during program development.

5.3. Code Generation tools

The development of code for TMS34020 systems follows a development flow as depicted in figure 5.2:
In figure 5.2 the most common paths are highlighted. It can be seen that the two major entries are assembler source or C-source. C source is compiled by the C compiler into assembler source. Then a common path is followed through the assembler and the linker. The result, an executable COFF (Common Object File Format) file can be loaded on the 34020 target system with the debugger.

5.3.1. The C compiler

The TMS34020 C compiler consists of a preprocessor, a parser and a code generator. Optionally the compiler can use an optimizer program. The C compiler package from TI includes a shell program that runs the four compiler passes, the assembler and optionally the linker. The linker is not executed by default, but on demand if the user specifies the -z option in the command line.

Preprocessor, parser and code generator can also be run separately. For these programs a wide set of command line options are available, refer to [GSPCO]. The assembler and the linker can also be run separately.

The TMS340 C language is the C language that is designed for the 34020 as well as for its predecessor the 34010. It is based on the Unix system V C language that is described by Kernighan and Ritchie [KER]. The compiler does not in any way generate assembly code that contains the specific 34020 graphics instructions such as TFILL or PixBLT. It is therefore necessary to have an interface between C code and assembly code.

One way of mixing of assembly and C is to write separate sources that can be linked together. A more elegant way is provided within TMS340 C by support of the "asm"-instruction. When the acronym "asm" is encountered by the compiler in the first field of a C source line, the remainder of that line is directly accepted as assembler. This allows the creation of parts of software in assembler within a C source code. However, one needs to be very careful not to disrupt the C environment. In-
Introducing jumps, calls and so forth can cause unpredictable results because the assembly code is not checked by the compiler.

5.3.2. Run-time support
Some functions that can be performed in a C program are not included in standard C. Examples are floating point arithmetic, string operations and dynamic memory allocation. To be able to perform these tasks runtime support functions are provided with the C compiler. The functions are included in the runtime support library (rts.lib).

Runtime support functions are declared in header files. A header file contains a set of related functions, the types that are needed are declared and macros are declared. A few of the included header files for TMS340 C are:

```
- math.h: defines several trigonometric, exponential and hyperbolic math functions. The functions expect double precision floating point arguments and return double precision floating point values.
- errno.h: declares a variable named errno that can be used for checking of error conditions.
- stdlib.h: declares common library functions such as string conversion functions, searching and sorting functions, program exit functions and integer arithmetic functions that are not a standard part of the C language.
```

Most C compiler environments have the socalled stdio.h (standard I/O) file. It is used for file I/O. It is important to know that no stdio.h header file is provided for TMS340 C. This means that I/O functions for for example reading and writing to files are not available.

5.3.3. Assembler and linker
The assembler translates the assembler source files into machine language object files. The assembler can produce a source listing that can be user formatted. Furthermore a cross reference listing can be produced.
With the linker assembled files (object files) can be linked together into a COFF output file. The most elegant way to link files is to invoke the linker and include the name of a socalled linker command file in the command line.

The linker command file is important because it allows the user to specify the allocation of memory to different program sections. Also the input files can be listed in the command file. A -c option can be specified that forces the processor to begin with a standard initialization and variable declaration when a program is loaded and run [GSPCO]. In case of using the SDB this option must be used.

The linker command file also specifies which libraries must be used during linking. Libraries that are provided are for instance rts.lib for runtime support and flib.lib for arithmetic emulation. For generating code for a system that includes the TMS34082 floating point coprocessor libraries (flicb.lib and flibcb.lib) are provided that form a IEEE function library for this coprocessor.

5.4. High level language debugger
To provide software debugging facilities on the Software Development Board Texas Instruments provides a third party high level language debugger. This debugger is manufactured by Ponsor Corporation and is included in the software package that comes with the SDB. The debugger is referred to as "Portable TMS34020 Debugger" or PTD [PTD]. It should be noted here that the PTD is not specifically meant for use with TI's SDB only, it can be reconfigured and adapted to almost any target system including the TMS34020. By default however it is configured for the SDB.

Probably the most important feature of the PTD is that it allows debugging of code at both assembler and C level. The user can load the code and choose which way of debugging is desired. The usual functionality such as placing breakpoints, modifying code while debugging, running, halting, resetting, single stepping is provided. Also the processor status with values of registers and the
memory can be inspected. Switching between C and assembler provides good insight in how C code is being compiled.

A drawback of the debugger and the Software Development Board is that no means for timing is provided. First of all the board does not have any timer or counter that can be read at specific intervals. Second, the debugger does not give clock or instruction cycle information. When debugging, timing can only be done for example by exiting to the operating system, give the DOS time command, re-entering the PTD, executing the code and exiting again to DOS to give another time command. The difference between the two times gives an indication of the duration of the program execution.

The switching between the debugger and DOS however causes overhead and influences timing. In order to have a reasonably accurate timing result the code that is being timed must execute for a long time compared to the time needed for the overhead.

5.5. Running the thick line benchmark

The C source for drawing a thick line is divided into the following source files (appendix A.1):

- linemain.c
- linedraw.c
- lineloup.c
- lineuplo.c
- linetrap.c
- linerast.c

Linemain.c contains the main function that calls a function Draw_Line several times to draw thick lines. Linedraw.c contains DrawLine. This function calculates the corner coordinates for the thick line and determines which function has to be called for the calculation of the upper and lower Y coordinates of the filled part of each scanline. Linedraw.c furthermore contains the functions Draw_Horline and Draw_Vertline that can be called by Drawline to draw thick horizontal and vertical lines.

Lineloup.c contains the function Do_Lower_Upper that can be called by DrawLine for calculation of fill boundaries in the scanlines for a rising thick line. Lineuplo contains Do_Upper_Lower that does the same for falling thick lines.

There are cases where a thick line has the form of a single trapezoid instead of a rectangular form. In these cases DrawLine calls the function Draw_Trap that is contained in the linetrap.c.

When the boundaries of the part of the scanline to be filled have been calculated the functions can call RasterLine that is contained in the linerast.c file. This function acts upon a two dimensional array of bits in memory. This structure is called the bitmap. Inside the bitmap RasterLine sets the appropriate bits to one filling the bitmaps scanlines.

In order to see the result of the rasterization of the thick lines in the bitmap array, an assembler routine has been written for the TMS34020 on the SDB. This routine is called ShowBitmap and is contained in the file showbitm.c. The routine is called from the main function when the process of rasterization is completed. ShowBitmap transfers the bitmap array to the systems video memory by using the PixBlt instruction. The result is that the bitmap can be inspected on the separate graphics monitor that is connected to the SDB.

5.5.1. Compilation and result

Compilation of the source went very straightforward. The linker command file however had to be written very carefully. The use of the linkers -c option was crucial for a good result. This option takes
care that program execution on the SDB start at label c_int00 that contains code to designate values to initialized variables. Furthermore the memory had to be divided in spaces for program memory and memory for the bitmap array. Interference with the video memory had to be avoided.

In order to run the benchmark the debugger was used. Before being able to load the code an initialization code had to be loaded on the SDB that configures the GSP's I/O registers. This initialization code was provided by TI. The functional result when the thick line benchmark was run was correct. Figure 5.3. depicts the lines that were drawn on the screen.

![Figure 5.3: Lines drawn by the thick line benchmark](image)

Figure 5.3 shows thick lines with widths of 3 an 7 pixels drawn in various directions.

After the functional correctness had been tested the C source was modified for benchmark timing. Firstly the call to the ShowBitmap function that puts the result on the screen was deleted. Then a loop construction was built around the main function in order to rasterize the pattern of thick lines many times. This was required because of the rather poor way of timing that had to be used. From the debugger an exit to DOS must be made to execute the time command and put the result in a file. After the benchmark another exit and time command must be made. To eliminate this overhead the main function was run 2000 times. Compared to an earlier timing session with 1000 times execution this gives only 0,1 % difference. So executing 2000 times is believed to be accurate enough.

Three sessions were carried out. For the first one the C source was compiled without using the Optimizer. The second and third sessions made use of level 0 and level 1 optimization respectively [GSPCO]. The total execution time was divided by 2000 to get the result for executing the benchmark one time. These results are in table 5.1.
<table>
<thead>
<tr>
<th>Optimization</th>
<th>Execution time (s)</th>
<th>Length of COFF file (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>3,269</td>
<td>106935</td>
</tr>
<tr>
<td>Level 0</td>
<td>1,799</td>
<td>104589</td>
</tr>
<tr>
<td>Level 1</td>
<td>1,904</td>
<td>104444</td>
</tr>
</tbody>
</table>

From Table 5.1 it can be concluded that in this case level 0 optimization gives a major speed up of execution of the benchmark (45%). Code is reduced with 2%. According to [GSPCO] level 1 optimization should give further improvement. Although level 1 optimization slightly reduces code further, the execution speed is not as fast as with level 0 (42% compared to no optimization).

5.5.2. Optimization of the benchmark through TFILL instruction
As described in chapter 4 the TMS34020 performs fast rasterization through single instructions. It has also been noted that the C compiler doesn’t employ these instructions during the compilation so the programmer must write pieces of in line assembly to fully exploit the 34020’s capabilities.

In order to speed up the execution of the drawing of thick lines the TMS34020’s TFILL instruction can nicely be used. This instruction rasterizes trapezoidal forms and thick lines can be split up into trapezoids. In some cases a thick line can even be drawn with one trapezoid. Figure 5.4 depicts the splitting up of a thick line into three trapezoids.

Figure 5.4: Splitting up a thick line
The first step that was taken for optimization was modifying the function Draw_Trap. The C code of this function was replaced by an assembly routine that draws the same trapezoidal form with the TFILL instruction. A listing is given in appendix A2.

The Draw_Trap function a mixture of C and assembler. The header and variable declaration is written in C, the code is further made up of in line assembly.

The next step was modifying the Draw_Line function. Calls to the functions Do_lower_Upper and Do_Upper_Lower were removed and three calls to DrawTrap that use the calculated corner coordinates of the line were inserted. Two extra Y-coordinates had to be calculated to fully describe the three trapezoids. These are marked with circles in figure 5.4.

After compilation and loading of the new code the result of the optimized program is practically the same as that from the original code. Close eye inspection reveals slight differences of the pixels that are drawn especially at the corners where one trapezoid ends and the other one starts. This can be explained by the fact that the TFILL instruction apparently uses a different algorithm for the filling of scanlines with pixels.

The resulting speed up of the execution of the benchmark is however spectacular. In the case of no optimization timing showed that the same drawing executes in 9.4% of the time as compared with the C program without TFILL instructions. Using the same method of timing the following results were yielded (Table 5.2)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Execution time (s)</th>
<th>Length of COFF file (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0.306</td>
<td>105471</td>
</tr>
<tr>
<td>Level 0</td>
<td>0.263</td>
<td>103582</td>
</tr>
<tr>
<td>Level 1</td>
<td>0.278</td>
<td>103298</td>
</tr>
</tbody>
</table>

Although functionality must be re-checked when using the optimizer the drawing result did not show any difference for each of the three cases. It is seen that level 0 gives less code than no optimization (1.8%) and that it gives substantial speed up in execution (14%). In this case again it is seen that level 1 optimization gives a slight further reduction in the size of the loadable COFF file but the execution is slower than with level 0 optimization (only 9% compared to no optimization).

Three conclusions can be derived from the benchmark sessions. Firstly it is very advantageous to develop assembler code that uses the TFILL instruction, and very assumably for that matter the other specific rasterization instructions furnished by the 34020, and to combine this assembler with C source. This way of software development as compared to only writing C source and compiling it straightforward, can give an improvement in execution speed of over a factor 10. This can be explained by the fact that the 34020 rasterization routines inhibit the processor from having to perform
many calculations in software.

Secondly the use of the optimizer was beneficial for both the sources. Using the optimizer the ex­
ecution time was reduced by 45% with the original C source version and by 14% with the source that
used TFILL.

Finally care should be taken as to which level of optimization to use. Empiric results for the optimiz­
atation level should determine the choice for a level. The next chapter will give the conclusions that
can be drawn from the evaluation of the TMS34020.
6. Conclusions on the evaluation of the TMS34020

This chapter treats the conclusions that can be drawn from the studies of the 34020’s hardware, software and tooling. Furthermore some interesting differences between the 34020 and its predecessor the 34010 will be described.

6.1. Hardware

The TMS34020 GSP is a full 32 bit microprocessor that can best be classified as a combination of a general purpose processor and a graphics engine with a wide set of extra on-chip functions. The GSP is available in a 32 MHz version (fully engineered, widely available) and a 40 MHz version (sample stage). Machine cycles are built up of four processor clock cycles and are 125ns and 100ns respectively. When operating from the on-chip 512 byte instruction cache this gives an 8 MIPS or 10 MIPS peak performance.

The processor core has general purpose architecture with two register files, an ALU, program counter, stack pointer, status register and barrel shifter. A set of on-chip I/O registers and one of the two register files that is graphics dedicated give the GSP its graphic characteristics. Through these registers parameters can be specified that allow the GSP to perform graphics operation such as pixel processing, masking, transfer of pixel arrays and the like.

The on-chip functions allow for flexible and direct DRAM interfacing and host interfacing. The TMS34020 can perform DRAM accesses in enhanced page mode, speeding up accesses by almost a factor two. Furthermore the GSP is capable of interfacing to display systems via the video interface. Multiprocessing can be performed due to the on-chip three signal multiprocessor interface. Finally the 34020 can interface to a floating point coprocessor, the TMS34082, for extended performance.

Through the above characteristics the TMS34020 is very well suited for graphic system applications that can have simple designs without much extra logic. The mix of general purpose and graphics dedicated allow for stand alone applications as well as pure graphic slave operation.

6.2. Software and tooling

Also in the instruction set of the 34020 the general purpose and graphics dedicated characteristics can be recognized. The general purpose instructions form a large and useful set and they allow for the fact that a full C-compiler has been implemented for the GSP. Code with coprocessor instructions can be generated for applications that use the TMS34082 FPU.

By writing C-source for the 34020 one doesn’t call upon the graphics instructions of the GSP instruction set. For graphics applications however in line assembly can be used to let the GSP perform its specific graphic operations. There are instructions for fast line drawing, filling, pixel block transfers, trapezoid filling and so forth.

It can be concluded that it is necessary to use these specific assembly instructions if full use of the 34020’s power is desired.

As stated earlier a full Kernighan and Ritchie C-compiler is available. Furthermore a third party high level language debugger existst for the GSP. For software development a board is available that implements a general 34020 system architecture for a graphics card. Because of the general character of this system software development and testing can be done on this board.

6.3. Comparison with the 34010
The 34020 is the second member of the TMS340 graphics family and it evolved from its predecessor the 34010. The most important differences are listed below, table 6.1.

<table>
<thead>
<tr>
<th>TMS34020</th>
<th>TMS34010</th>
</tr>
</thead>
<tbody>
<tr>
<td>full 32 bit processor</td>
<td>32 bit internal, 16 bit busses</td>
</tr>
<tr>
<td>enhanced page mode</td>
<td>no page mode</td>
</tr>
<tr>
<td>LAD bus for linear address and data, RCA bus for multiplexed row and column</td>
<td>row and column multiplex on LAD bus no RCA bus</td>
</tr>
<tr>
<td>512 byte instruction cache</td>
<td>256 byte instruction cache</td>
</tr>
<tr>
<td>100 ns machine cycle (40 MHz)</td>
<td>200 ns machine cycle (40 MHz)</td>
</tr>
<tr>
<td>extended graphics instructions</td>
<td>basic graphics instructions</td>
</tr>
<tr>
<td>multiprocessing capability</td>
<td>no multiprocessing</td>
</tr>
<tr>
<td>external 32 bit host data path: direct access</td>
<td>host data through processor: indirect access</td>
</tr>
<tr>
<td>extended pixel processing</td>
<td>standard pixel processing</td>
</tr>
<tr>
<td>expansion with 34082 FPU</td>
<td>no FPU</td>
</tr>
</tbody>
</table>

The performance enhancement of the 34020 compared to the 34010 can be in the range of 6 to 50 times as fast, depending on the application [PET]. After the evaluation it can now be concluded that using the 34020 in future system design can significantly enhance system performance.

Finally a note on growing path. Although the 34020 is only widely available in the 32 MHz version and the 40 MHz version is being sampled, TI has announced a 34030 GSP. Further details are unknown but it is expected that a future 34030 may have the FPU on-chip.

The next chapter treats the design requirements for a 34020 based Raster Image Processor unit.
7. Design requirements for the Raster Image Processing Unit

As part of the activities for the performance enhancement of printer systems a study has been made for an alternative design for the Raster Image Processing (RIP) Unit. This chapter treats the RIP unit design requirements. To start with, an analysis of a 34010 based RIP unit will lead to important system requirements. Then the possible sizes of bitmaps are studied leading to memory size requirements. Finally requirements for the bitmap reader hardware will be described.

7.1. Analysis of a 34010 based RIP unit design

A RIP unit can be thought of as a system that basically consists of a Raster Image Processor, local memory, bitmap memory and reader hardware. On the one hand a RIP unit interfaces with the actual printer engine’s Laser Scan Module (LSM). The LSM puts the dots on a page like an electron gun in a monitor. On the other hand the RIP unit needs to interface to a front end system. See the block diagram in figure 7.1.

![RIP unit block diagram](image)

The 34010 based RIP unit design features the TMS34010 Graphic System Processor for the RIP. It has 2 Mbytes of zero wait states local memory that is organized as a 16 bit wide memory consisting of 16 1 Mbit DRAMs. The bitmap memory is organized in two interleaved banks of 16 1 Mbit DRAMs making up 4 Mbytes of 16 bit wide zero wait states memory. The reader hardware can take ownership over the 4 Mbyte bitmap memory when the laser scan module requires image data. The reader can address the bitmap and delivers serial data to the LSM. The RIP unit communicates with the front end system over a VME-bus interface. A block diagram is given in figure 7.2.
A critical point in this design is the way that the bitmap memory is shared between processor (GSP), which has to perform rasterization in the bitmap and the reader hardware. When a page has to be printed the reader completely owns the bitmap memory. During this page time the GSP is condemned to inactivity or to actions that limit memory access to only local memory. The processor is not able to perform it's major task: rasterization. Only during the time between the end of a page and the start of a new page, the interpage time, the processor can act upon bitmap memory.

A theoretical calculation of page and interpage time will demonstrate the limits that the sharing of the bitmap in this way puts upon the GSP's percentage of time that is available for rasterization. Page time can be calculated from the time that is needed to print one scanline and the total of scanlines for a page. The total of scanlines for an A4 page (297mm x 210mm) equals the number of dots per mm times the page width in mm. For a 300 dpi printer at 45 pages per minute this gives:

number of scanlines = 12 * 210 = 2520

page time = 2520 * 340.10^{-6} = 0.86 seconds

where 340.10^{-6} seconds is the time to print one scanline.

The total time per page is derived from the number of pages per minute:

time per page = 60/45 = 1.33 seconds

The interpage time therefore has a duration of:

1.33 - 0.86 = 0.47 seconds
From this calculation it is seen that the processor can do its rasterization for only 35% of the time.

Other printer modes with different resolutions and printer speeds show the same relation for page and interpage time of 2/3 to 1/3 respectively. It is clear that the way the bitmap is shared is not optimal.

Another critical point in the design is the limited flexibility that it offers. This inflexibility lays in two areas. Firstly the configuration and size of the memory is fixed and secondly the reader hardware’s start address for reading the bitmap is not changeable. This causes the RIP unit to have limited ability where holding bitmaps of large sizes is concerned and it limits the number of bitmaps that can be held in memory simultaneously to only one. No back ups of bitmaps can be saved for for instance error recovery at bitmap level.

The design furthermore does not support the possibility of XY-addressing that the 34010 offers. This is mainly because XY-addressing asks for larger bitmaps and due to the limited memory size of the design it is not possible to store these bitmaps. Limited memory size also prohibits the use of a scratch memory, i.e. a scratch pad for bitmaps.

Finally the RIP unit does not provide a hardware possibility to clear bitmaps on demand, without having to read them first. When there would be the possibility of holding more bitmaps simultaneously for bitmap level page recovery, such a provision would be necessary.

The 34010 based design provides the possibility for mirroring. This means that the memory can be read and written in reversed bitorder. It is required for for instance the mirroring of font bitmaps.

From the above described analysis the following requirements can be summarized for the redesign:

- Solve the page/interpage time problem by a different way of memory sharing that increases system performance.
- Ability to hold several bitmaps at a time
- Ability to hold bitmaps of sizes that meet future resolutions
- Provision for scratch memory
- Programmable reader hardware with variable start and end address
- Support for XY-addressing
- Mechanism for clearing of bitmaps on demand
- Mirror function for reversed reading of the bitmaps
- Design should be as flexible as possible by being configurable to different printer modes

7.2. Sizes of bitmaps

The size of a bitmap in pixels is calculated by multiplying the width of the page in inches or millimeters by the resolution in dots per inch or per millimeter, and multiplying this by the product of the page’s height (inch or millimeter) with that same resolution. Multiplying the amount of pixels by the number of bits per pixel gives the bitmaps size in bits. This size can be easily expressed in bytes by a division by eight.

A 508 dots per inch (dpi) resolution equals a 20 dots per millimeter (dpm) resolution. An A4 (210 mm x 297 mm) bitmap with this resolution has a size of:

\[ 210 \times 20 \times 297 \times 20 = 24,9.10^6 \text{ pixels} \]

At one bit per pixel this gives:
(24,9/8) \cdot 10^6 \text{ bytes} = 2.97 \text{ Mbyte}

Table 7.1 lists the different bitmap sizes for different resolutions. Also the sizes required when XY-addressing is supported will be given. For these sizes the specification of XY-addressing with the TMS34020 GSP is used. This XY-addressing distinguishes three cases. In the first case the pitch, or distance between two scanlines is a power of two. This case requires the least time needed for the GSP to convert the XY-address to linear.

In the second case the pitch is the sum of two powers of two. This requires slightly more effort from the GSP for conversion. The third case handles arbitrary pitches and is rather inefficient in address conversion. For a detailed description on XY-addressing, see chapter 3.6.

### Table 7.1: Bitmap sizes (Mbyte) for different resolutions and XY-support

<table>
<thead>
<tr>
<th>XY-addressing</th>
<th>300 dpi/12 dpmm</th>
<th>480 dpi/19 dpmm</th>
<th>508 dpi/20 dpmm</th>
<th>600 dpi/24 dpmm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power of two (case 1)</td>
<td>1.23</td>
<td>3.90</td>
<td>4.10</td>
<td>4.92</td>
</tr>
<tr>
<td>Two powers of two (case 2)</td>
<td>1.23</td>
<td>2.92</td>
<td>3.08</td>
<td>4.92</td>
</tr>
<tr>
<td>Arbitrary or no XY-addressing (case 3)</td>
<td>1.07</td>
<td>2.68</td>
<td>2.97</td>
<td>4.28</td>
</tr>
</tbody>
</table>

The information from table 7.1 is necessary when the size of the system’s memory is to be determined.

### 7.3. Memory size requirements

The memory size requirements are determined by the following parameters:

- **Program memory size**
- **Provision for scratch memory**
- **Number of bitmaps that have to be stored simultaneously**
- **Size of the bitmaps**
- **Space needed for rasterized fonts (data caches)**

For the final architecture aspects such as memory device sizes and organization, cost price and availability determine the exact memory size.
The size of program memory depends on the required space for code for the 34020 and local data storage. In the 34010 based board local memory is 2 Mbyte large. This space is used for code and data storage (rastered fonts). For the 34020 based design 2 Mbyte of memory space for this purposes should be reserved.

A scratch memory can be considered to be as large as the bitmap size that is used in the application. This could mean a size of 1,07 Mbyte up to 4,92 Mbyte worst case. The same goes for the bitmaps to be stored.

The number of bitmaps that have to be stored of course has it’s minimum at one. However if the new design allows the GSP to perform rasterization during page time a performance increase could be possible if more bitmaps could be stored. On the one hand rasterization during page time allows the RIP unit to finish bitmaps before they have to be printed, that otherwise wouldn’t have been ready and would cause the printer engine to wait. On the other hand the GSP can start rasterizing new bitmaps during page time. This would mean more memory space for bitmaps. Suppose there is space for three bitmaps. The GSP can then for example have two bitmaps ready for printing and work on a difficult page while the two other pages are printed.

When the RIP unit should be able to perform page recovery at bitmap level, nine rasterized pages should be simultaneously in memory. The number of nine pages is determined by the specification of the printer engine. Although this consumes a lot of memory it is desirable to have the possibility.

The largest memory size required would be in the case where 2 Mbyte of program memory would be required together with a 4,92 Mbyte scratch memory and nine 4,92 Mbyte bitmaps. This would yield 51,2 Mbyte!

More realistic is a memory requirement for nine 2,97 Mbyte bitmaps, one 2,97 Mbyte scratch memory and 2 Mbyte of program memory, leading to 31,7 Mbyte.

When the scratch memory is skipped in this requirement 28,7 Mbyte would be necessary. It should already be kept in mind that price and device count influences the memory size. Smaller sizes would be needed when page recovery is left out for the high resolution pages. For instance the requirement for a 508 dpi printer with the possibility of XY-addressing and storage of three 4,1 Mbyte bitmaps. With a program memory space of 2 Mbyte 16 Mbyte can satisfy the needs. 16 Mbyte would allow page recovery, scratch memory and XY-addressing for a 300 dpi printer: 10 times 1,23 Mbyte together with 2 Mbyte program space adds up to 14,3 Mbyte. Table 7.2 gives an overview of the required amount of memory for the different memory configurations. Practical situations could cause the designer to offer a piece of program memory if hardware architecture is simplified by this.
Table 7.2: Amounts of necessary memory (Mbyte)

<table>
<thead>
<tr>
<th>Memory configuration</th>
<th>Resolution configuration</th>
<th>300 dpi</th>
<th>480 dpi</th>
<th>508 dpi</th>
<th>600 dpi</th>
</tr>
</thead>
<tbody>
<tr>
<td>pro, scratch, 9 bitmaps, case 1 XY-addressing</td>
<td>300 dpi</td>
<td>14,3</td>
<td>42,0</td>
<td>43,0</td>
<td>51,2</td>
</tr>
<tr>
<td>pro, scratch, 9 bitmaps, case 2 XY-addressing</td>
<td>480 dpi</td>
<td>14,3</td>
<td>31,2</td>
<td>32,8</td>
<td>51,2</td>
</tr>
<tr>
<td>pro, scratch, 9 bitmaps, case 3 XY-addressing</td>
<td>508 dpi</td>
<td>12,7</td>
<td>28,8</td>
<td>31,7</td>
<td>44,8</td>
</tr>
<tr>
<td>pro, 9 bitmaps, case 1 XY-addressing</td>
<td>600 dpi</td>
<td>13,1</td>
<td>37,1</td>
<td>38,9</td>
<td>46,3</td>
</tr>
<tr>
<td>pro, 9 bitmaps, case 2 XY-addressing</td>
<td>11,6</td>
<td>26,1</td>
<td>28,7</td>
<td>40,5</td>
<td></td>
</tr>
<tr>
<td>pro, 9 bitmaps, case 3 XY-addressing</td>
<td>6,92</td>
<td>17,6</td>
<td>18,4</td>
<td>21,7</td>
<td></td>
</tr>
<tr>
<td>pro, scratch, 3 bitmaps, case 1 XY-addressing</td>
<td>6,92</td>
<td>13,7</td>
<td>14,3</td>
<td>21,7</td>
<td></td>
</tr>
<tr>
<td>pro, scratch, 3 bitmaps, case 2 XY-addressing</td>
<td>6,28</td>
<td>12,7</td>
<td>13,9</td>
<td>19,1</td>
<td></td>
</tr>
<tr>
<td>pro, scratch, 3 bitmaps, case 3 XY-addressing</td>
<td>5,69</td>
<td>13,7</td>
<td>14,3</td>
<td>16,8</td>
<td></td>
</tr>
<tr>
<td>pro, scratch, 3 bitmaps, case 1 XY-addressing</td>
<td>5,69</td>
<td>10,8</td>
<td>11,2</td>
<td>16,8</td>
<td></td>
</tr>
<tr>
<td>pro, scratch, 3 bitmaps, case 3 XY-addressing</td>
<td>5,21</td>
<td>10,0</td>
<td>10,9</td>
<td>14,8</td>
<td></td>
</tr>
</tbody>
</table>

7.4. Reader functional requirements

Flexibility, possible support of XY-addressing, and the fact that several bitmaps are in memory simultaneously implies that the bitmap reader is programmable. When a bitmap has to be read the reader should have knowledge of the starting address of the bitmap. Furthermore the reader must know the length of one scanline and the pitch between scanlines (XY-addressing). This because of the fact that when XY-addressing is used there is data between the end of one scanline and the start of the next that must not be delivered to the printer engine. Finally the reader must know when a bitmap is totally read. This implies knowledge of the total amount of scanlines of the bitmap.

The reader can also be used for clearing of bitmaps. It should be able to read the bitmaps locations and clearing the location at the same access (read modify write) or to read the location only. Reading and not clearing is necessary when a page has to be printed more than once. Page recovery at bitmap level requires clearing-on-demand (clearing without reading). The reader could perform this clearing by writing to all the locations in the bitmap when this is desired.

The next paragraph describes the alternative architectures for the RIP unit’s memory.
8. Memory architecture alternatives

Using the requirements from the previous chapter a memory architecture must be designed including a memory sharing mechanism. It is the sharing problem that will be treated first.

8.1. Sharing of the memory

As was seen in chapter 1 the problem with sharing of the bitmap memory is that the processor cannot access the bitmap when the page is being read by the reader. When memory is able to store more than one bitmap one could consider to partition the memory physically into banks the size of one bitmap. In this way the reader can gain ownership over the memory banks that it needs for reading according to the start and end address information. This information can be given by the processor when signalling the reader hardware that a page is rasterized. With separate memory banks the advantage is that the processor can still access a large part of the memory when the reader has gained ownership over the bank(s) that contain the bitmap to be printed.

A disadvantage is that for every memory bank hardware is required to create separated data and address busses for the processor and the reader. Furthermore the banks must be able to switch between the two possible owners. The RIP unit would also be limited in the flexible allocation of memory for bitmaps. The memory banks cause the hardware to determine the possible start addresses for the bitmaps.

Another possibility for memory sharing is to design some kind of multiprocessing environment in which a separate arbitration mechanism receives requests from the processor or the reader and can decide, using a priority scheme, who gets the common bus. This way of sharing can be very structured and tailored to the needs of the system. Because the arbiter can use the TMS34020 multiprocessing protocol the behavior of the sharing mechanism is fully predictable. Furthermore a shared memory with a shared bus keeps the hardware less complex.

A first memory architecture alternative presented here will be based on bank partitioning.

8.2. Architecture with separate banks

As was seen in chapter one the requirements for memory size can be up to 51,2 Mbyte. It is therefore necessary to use memory devices that have a high density of memory locations. For this purpose dynamic RAMs or DRAMs are widely used.

The development of DRAMs is at a point where devices of 1 Mbit are in the maturity phase. They aren't expensive and they are easily available. The next generation in DRAMs is formed by the 4 Mbit devices. At the moment they are getting into maturity and they are taking over the 1 Mbit DRAMs. Availability is good and 4 Mbit is advised to be used in products that are due to the market in the next two years. The price per bit for 4Mbit is about equal to that of 1Mbit and a further decrease in price is expected.

In this paragraph the 4 Mbit DRAMs from Toshiba (TC514400J/Z-80 and -10, TC514100J/Z-80 and -10) will be studied [TOSH]. For these 4 Mbit DRAMs there are two major ways of device architecture: 4Mx1 and 1Mx4. See figure 8.1.
Both types of devices come in two versions, a slower and a faster one. They allow read and write cycle times of 150 or 180 ns according to the selected version. Access times are 80 and 100 ns respectively.

The devices differ in their timing for read-modify-write cycles. This type of cycle is very important for reading and clearing a bitmap location in one cycle. For the 4Mx1 device this cycle time is 175 or 210 ns. The 1Mx4 device is slower because of its bidirectional I/O lines. Read-write cycle timing for this device is 205 or 245 ns.

The 32 MHz version of the 34020 has memory cycle times of 250 ns for reads and writes. Read-modify-write is performed by the 34020 by a read of 250 ns with a subsequent page mode write of 125 ns, giving a total of 375 ns. For a 40 MHz GSP these times are 200 ns for read or write and 300 ns for read-modify-write. It should therefore be possible to design zero wait state memory with the DRAM devices.

For more information on the specifications of the DRAM devices see [TOSH].

The fact that the 34020 demands a 32 bit databus means that the smallest units of DRAM memory can be 32 4Mx1 devices, yielding 16 Mbyte, or 8 1Mx4 devices, yielding 4 Mbyte. See figure 8.2.
Figure 8.2a: 8(1Mx4) memory unit, 4 Mbyte
The memory unit that comprises eight 1Mx4 DRAMs is suitable for an architecture that consists of several separate memory banks. When the table with the sizes of the possible bitmaps is regarded, it is seen that most bitmaps would fit in one bank of 4 Mbyte. Only for a 600 dpi printer the bitmap would need a part of a second bank. Figure 8.3 shows the memory architecture for a 16 Mbyte memory consisting of four banks. The sixteen Mbyte size has been chosen based on the considerations of paragraph 7.3.
The memory architecture uses RASn lines 0 through 3 for bank selection. An address decoding mechanism can control these RASn lines. For bank allocation to reader or processor (GSP) the RDREN and GSPEN enable lines are used. Byte selection is possible via the CASn strobes 0 through 3.

From figure 8.3 it is seen that the architecture is rather complex because of the fact that the banks must be accessible via two separate systems of address and data busses. The RIP unit would need an administration system that controls the allocation of the banks to the processor and the reader hardware. This administration mechanism should allocate a whole bank to the reader if it wants to...
read a bitmap that is held in that bank. Suppose that a bank would be allocated to the reader during the page time the possible addresses for the processor to rasterize pages during page time would be limited. It is now that an architecture that comprises a single 16 Mbyte unit will be described.

### 8.3. 16 Mbyte memory architecture; the concept of scanline buffering

Figure 8.2b from paragraph 8.2 shows a very simple, flexible and straightforward memory architecture. It contains one bank of 16 Mbyte.

Sharing of this memory can only be done via shared address and data busses over which an arbiter decides on whether the processor or the reader owns them. At first sight one could say that such a solution wouldn’t enhance the way memory is shared compared to the 34010 based design. When the reader would claim memory during page time the whole memory would be closed to the processor and it would be still a situation in which the processor could act upon memory only for 1/3 of the time.

Significant performance increase could be yielded however if the reader could buffer image data. The reader could then access a part of the bitmap very quickly and store it internally in the reader hardware buffer. When the buffer is filled the reader can give the bus back to the processor. When the buffer runs empty the reader could again ask for the bus to fill it's buffer again.

Practically the actions would take place like this. First the reader gets the message that a page is ready at a certain start address. The reader signals the arbiter that it wants the bus. When the reader's request has been granted the reader accesses the memory in fast page mode. If the reader's buffer is full it relinquishes the bus so the processor can act on the memory until the buffer is serially emptied over the interface with the printer engine.

It is to be noted here that the TMS34020 offers a possibility for multiprocessing (see paragraph 3.5). By having a similar interface that follows the same protocol as the 34020 for the reader hardware the sharing of the memory can be treated as a multiprocessor situation with a shared bus and a shared memory. To get an impression of the performance a calculation for the percentage of time that the processor can act upon the shared memory during page time is given.

Suppose that the bitmap reader wants to fill the image data buffer every scanline. This implies a synchronization at scanline level. Worst case a scanline consists of 7128 pixels at 24 dpmm. Making this a multiple of 32 the total of 32 bit words to read for one scanline is 223 (no XY-addressing). When the reader has the same manner and speed of access as the processor (32 MHz) these 223 long words can be read in four complete and 219 page mode accesses (a page mode access session can consist of 64 accesses maximum in this case). The time needed is:

\[
4 \times 250 \text{ ns} + 219 \times 125 \text{ ns} = 28.4 \text{ microseconds.}
\]

For a 600 dpi or 24 dpmm printer with a 45 pages per minute printing speed the total time between two scanlines is 171 microseconds. So in this worst case the reader owns the bus for:

\[
\frac{28.4}{171} \times 100\% = 16.6\%
\]

and the processor owns it for:

\[
100\% - 16.5\% = 83.4\%
\]

of the page time. Of course the switching of the bus takes some time. However according to [GSPUSER] it takes a maximum of 250 ns after the processor’s grant input is driven inactive by an arbiter for the processor to drive all signals to high impedance. Furthermore it takes 250 ns maximum to recognize a renewed grant active and to take the bus. These timings are so small that they
are discarded in the above calculation.

Best case in this way of memory sharing is a 300 dpi/23 ppm printer’s calculation. The 12 dpmm resolution requires 112 accesses divided in two complete and 110 pagemode cycles. The time for these accesses is 14,3 microseconds. Time between two scans is 340 microseconds. This leads to 4,2 % of page time for the reader and 95,8 % for the processor.

Table 8.1 gives an overview for the different printers:

<table>
<thead>
<tr>
<th>Printer mode</th>
<th>% of page time for GSP</th>
<th>% of page time for reader</th>
<th>Total % of time for GSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>300dpi/23 ppm</td>
<td>95,8</td>
<td>4,2</td>
<td>97,2</td>
</tr>
<tr>
<td>300dpi/45ppm</td>
<td>95,8</td>
<td>4,2</td>
<td>97,2</td>
</tr>
<tr>
<td>480dpi/45ppm</td>
<td>89,4</td>
<td>10,6</td>
<td>92,9</td>
</tr>
<tr>
<td>508dpi/23ppm</td>
<td>93,7</td>
<td>6,3</td>
<td>95,8</td>
</tr>
<tr>
<td>508dpi/45ppm</td>
<td>88,3</td>
<td>11,7</td>
<td>92,2</td>
</tr>
<tr>
<td>600dpi/23ppm</td>
<td>91,1</td>
<td>8,9</td>
<td>94,1</td>
</tr>
<tr>
<td>600dpi/45ppm</td>
<td>83,4</td>
<td>16,6</td>
<td>88,9</td>
</tr>
</tbody>
</table>

Until now the two memory architectures comprised a total of 16 Mbyte. To meet more demanding requirements the final memory architecture for the system should allow expansion of the memory by an often used method called piggy backing. This is the expansion of facilities in general by use of the possibility to plug so called piggy back boards onto system boards. A memory piggy back of an extra 16 Mbyte seems to be very desirable.

The next paragraph describes the choices that have been made for the memory architecture of the 34020 system.

8.4. Memory architecture choice

The two main alternatives for the memory are an architecture consisting of four 4 Mbyte banks and a consistent memory of 16 Mbytes with no banks. The first alternative implies bank selection and allocation for the sharing mechanism whereas the second alternative implies a multiprocessing way of bus sharing with an arbiter. For this sharing system to be successfull a method of buffering of
image data is needed for the reader hardware. Expansion of the 16 Mbytes must be accomplished by piggy backing. This only has consequences for the system’s address decoder, the same memory architectures can be used for the piggy backs.

The banking alternative has the advantage that during page time not the total memory is unaccessible and the reader hardware doesn’t need a buffering mechanism. It has the bitmap to it’s own. However this architecture needs a lot of extra hardware in the form of DRAM drivers. Furthermore the memory flexibility is reduced.

Making up the memory with one contiguous 16 Mbyte block certainly simplifies architecture. Memory flexibility is optimal. When the reader hardware is able to buffer scanlines it has been calculated that the sharing of this memory allows 83 % or more of the page time to the processor.

Based on the above discussion it has been decided to choose the contiguous 16 Mbyte architecture for the implementation of the RIP unit’s memory. There is one complication however. Until now all the different parts of memory, i.e. program, font data, bitmaps, scratch, where assumed to be in the shareable memory architecture. This means that when the reader accesses the memory the GSP cannot make any accesses.

For the host system that transfers printer data to the RIP unit this means that during page time not all accesses can be serviced right away. In the case of a 600/45 printer access will be postponed for 16,6% of the accesses with a maximum wait period of 28 microseconds. As long as the host is able to wait 28 us in these cases the host-RIP interface functions properly.

It has been chosen to build the memory system out of one shared memory of 16Mbyte using 32 4Mx1 DRAMs. Furthermore arrangements must be made in the system to be able to expand bitmap memory with more 16 Mbyte blocks. The memory will be shared with an arbitration mechanism that uses the TMS34020 multiprocessing capability.
9. Further system issues

At this point the memory architecture is known to consist of shared bitmap memory for the 34020 and the reader. Because the reader will be able to buffer scanline data it has been calculated that during page time the reader needs access to the bitmap for 17% of the time. One access for buffering a scanline of a 600 dpi printer has a duration of 28 us. Bus arbitration can be done via the multiprocessor interface.

9.1. Guaranteeing refresh

The GSP performs CAS before RAS refresh cycles that have a duration of three machine cycles (= 375ns at 32MHz). The period between refreshes can be programmed to the needs of the DRAMs. All DRAMs that are planned to be used need a complete refresh every 16 ms. One refresh cycle refreshes one row of the DRAMs. With 1024 rows this gives a refresh period of 15.6 us. Normally a refresh is performed when the refresh period has expired and the GSP has not scheduled any access to memory, i.e. the GSP is performing host-default cycles. If the GSP is performing a CPU access (single cycle, page mode) the refresh is suspended and the refresh counter is incremented. The refresh counter can count up to 15 pending refreshes.

When 4 refreshes are pending a refresh is scheduled right after the current CPU access has ended. Host accesses however have higher priority and can still suspend a refresh cycle, even when four or more are pending. If 12 refreshes are pending this no longer is true. The refresh cycle gets higher priority over the host access.

Now, during page time with the reader having access to bitmap memory for 28 us one or two refresh periods of 15.6 us will expire. Suppose that two refresh periods expire. When these are the fourth and fifth refreshes two refresh cycles will be performed after the 34020 regains mastership and no host accesses are requested. The refresh for the DRAMs is guaranteed.

Suppose the GSP is performing host cycles to memory when the reader gains ownership over memory. Further suppose that the refreshes that are added to the pending list in this 28 us period are the 12th and 13th. Also in this case no refresh problems occur since the GSP can keep track of 15 refreshes and the pending 12th and 13th will be performed as soon as ownership is regained.

In the 34010 design refresh problems occur when the engine stops generating SOS during page time (master stop). Now these problems cannot occur since the reader always relinquishes the bus after it has filled the scanline buffer.

9.2. Clearing-on-demand

For error recovery at bitmap level or for initialization purposes the RIP system must be able to only clear a bitmap without having to read it, as opposed to reading and clearing it by read-modify-write when a page is printed and subsequently discarded.

Now the choice can be made to build clearing functionality in the reader hardware or to perform clearing by software with a page mode FILL instruction. Comparing the two methods gives the following. Suppose reader hardware can access DRAMs in page mode with a new row address every 1024 accesses. For a 4,28 Mbyte bitmap this gives 4,28*(1024*1024)/4 = 1,122.10^6 page mode cycles and 1094 row address cycles. The total of machine cycles is then 1,123.10^6 giving a time of 140 ms. A software FILL costs 1,122.10^6 plus 17,54.10^3 row cycles (every 64 page mode cycles the GSP gives a new row address). The total of machine cycles now is 1,139.10^6 giving a time of 142 ms. It is seen that the difference in time between hardware or software clearing is marginal and does not justify a hardware solution.

The above calculation doesn’t include the time taken by refreshes. During a software clear the GSP
can schedule refreshes between the page mode accesses. Building the clear function in the reader hardware means that the reader now must be able to be interrupted for refreshes rather than taking the bus when its request is granted and releasing it when it has finished.

9.3. XY-addressing

Supporting XY-addressing would simplify software design since coordinates need not be converted by software to linear addresses. The speed of rasterization however is not significantly increased. It does however introduce the need for more memory to store one bitmap.

Still it is desirable that the reader hardware supports XY-addressing. In fact this support is accomplished quite easily since the reader will be programmable with the length of one scanline, or better, the number of accesses that must be made to fill a scanline buffer. In the case of XY-addressing the 34020 simply tells the reader to fill up the buffer up to the number of bits that equals the nearest power of two or the nearest sum of two powers of two that is equal or greater than the scanlength. So doing the reader will fill the buffer using the same pitch that is used by the 34020 itself for XY-addressing. The buffer thus contains one scanline of data and the following 'don't care' data that the memory contains between the end of one scanline and the beginning of the next scanline. Automatically the reader uses the right address for the buffering of the following scanline.

This way of XY-addressing support works well since the printer engine only reads a scanline buffer as many times as needed to get the right amount of bits for one scanline. The rest of the data in the buffer, that is don't care is not read by the engine. Furthermore the scanline buffer is reset and filled anew every scanline so the engine will read the right data during the next scanline.

An example. Suppose a 600 dpi printer. The scanlength equals 7128 bits. When no XY-addressing is used by the 34020 the reader must be programmed to fill a scanline buffer by accessing memory 223 times (223*32bits = 7128bits). When the 34020 uses XY-addressing and the pitch is thus 8192 bits (power of two), the reader must be programmed to access memory 256 times in order to fill the scanline buffer. The engine will only read out the first 7128 bits in the buffer. For the next scan the buffer is reset and filled again.

9.4. Mirroring

By mirroring the following is meant. It should be made possible by hardware for the GSP to write data to bitmap memory in reversed bitorder. A solution to this is to replace the buffers on the data-busses between GSP and memory by Programmable Logic Devices (PLDs) that can mirror the data-lines. The command line that tells a PLD to mirror or not can be decoded out of the address on the LAD bus. The mirrored bitmap memory must be mapped in the memory map.

9.5. TMS34082 coprocessor

The RIP design must support the use of a TMS34082 floating point coprocessor. According to [GSPUSER] the coprocessor has to be connected to the LAD bus from the 34020. The coprocessor is a slave to the GSP. Coprocessor cycles are performed mainly by using the CASn signal. RASn is kept inactive. It is therefore possible to keep the coprocessor from influencing the rest of the system. The systems hardware should provide the coprocessors interface and socket.
10. Using FIFOs for scanline buffering

This chapter further elaborates on the method of sharing the system memory in an economic manner. In paragraph 8.3 the concept of having a data buffer for the bitmap reader was introduced. It was calculated that in the worst case, scanline buffering for a 600 dpi/45ppm printer, the percentage of time that the reader owns the system memory was as low as 16.6% during page-time and 11.1% overall. Furthermore in this case the filling of the scanline buffering or the duration of one contiguous period of ownership by the bitmap reader was 28 us.

Now let's look at the engine interface in more detail. This will give insight in how data must be delivered to the engine and what signals are involved.

The interface consists of five signals. Three signals are sent from engine to RIP unit. These are Start Of Page (SOP), Start Of Scan (SOS) and BURST. Two signals are sent from the RIP unit to the engine. These are Page Available (PAV) and Image Data (IMDA). Each time the printer engine is ready to print a page it asserts SOP low. SOP is kept low until the page has totally passed the laser scan module (LSM), i.e. the end of the page-time. After SOP low the engine checks if PAV is high. PAV high indicates that the RIP unit has a page ready to deliver to the LSM. If after 1 ms (PECSPEC) after SOP = low PAV is still low then the engine knows no page is ready. After SOP the engine sends SOS pulses every time a scanline is required. The BURST pulses that then follow will be used to clock data over the IMDA line from RIP unit to engine. Figure 10.1 shows the timing for the interface.

The concept of scanline buffering has been chosen for several reasons. First this buffering takes advantage of the fact that filling the buffer by the reader goes much faster than emptying it by the printer engine (See paragraph 8.3). Secondly the amount of BURST pulses could be too big or small
due to erroneous operation. Suppose that one buffer could hold more than one scanline such an
error would have the effect that data would be printed on wrong positions on the paper. Synchroni-
zation on scanlines prevents this.

The choice has been made to equip the RIP unit with two FIFOs that act as scanline buffers. The use
of two rather than one FIFO has the advantage that the bitmap reader can start buffering a scanline
right after the moment the processor signals that a page has been rasterized. After the first scanline
has been buffered the reader can synchronize on SOP and SOS and present IMDA in time from this
buffer. Meanwhile the SOS pulse initializes the reader to start a fill action on the second FIFO. This
way of operation gives the reader more than enough time to fill the second buffer until the second
SOS pulse arrives.

The FIFOs must be of the parallel-to-serial type. One FIFO must be able to be filled 32 bits in parallel.
Data can be shifted out serially. One FIFO has to contain one scanline. Table 10.1 lists the length of
scanlines for different printer resolutions.

<table>
<thead>
<tr>
<th>Printer resolution (dpi; dpmm)</th>
<th>Scanline length (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>300; 12</td>
<td>3564</td>
</tr>
<tr>
<td>480; 19</td>
<td>5643</td>
</tr>
<tr>
<td>508; 20</td>
<td>5940</td>
</tr>
<tr>
<td>600; 24</td>
<td>7128</td>
</tr>
</tbody>
</table>

It is seen that the longest scanlines occur in a 600 dpi printer. One scanline in this case is 7128 bits.
When XY-addressing is applied the nearest power of two is 8192. Therefore the FIFOs, having 32 bit
words locations, must be 256 locations deep. (256 * 32 = 8192).

A study has been made of the FIFOs currently available. The most suitable devices seem to be IDT’s
72105 256x16 parallel-to-serial FIFOs (See [IDT] for the datasheet). These FIFOs are 16 bit wide and
have 256 locations. Two of these devices can be used for width expansion to 32 bits. A functional
block diagram is given in figure 10.2
It can be seen that the FIFO consists of a 256x16 RAM array with a write and a read pointer to it. The write pointer is controlled by the signal on the Wn pin. The read pointer is controlled by the serial output circuitry. Each time a word has been serially shifted out a new word is moved to the serial output circuitry from the read pointers location. Furthermore the FIFO has flag logic that controls a full flag, an empty flag and a half full flag. Expansion logic controls the operations in expanded architectures of several FIFOs together.

The FIFO can be written 16 bits parallel with a write cycle time of 35ns minimum. Asynchronously and independently the FIFO can be written and read. The maximum serial shift frequency is 50 MHz. The 72105-25 is easily width expanded. See figure 10.3.
Both FIFOs need to be reset before beginning operation. This is done by issuing a reset pulse (active low, pulse width 25ns min, cycle time 35ns min) on the two reset pins. The two FIFOs must shift out data serially with the least significant bit first [PECSPSEC]. According to the datasheet this means that during operation both the FL/DIR pins must be logic '0'. During reset however the FL/DIR pin of FIFO #2 that contains the most significant half of the 32 bit word must be high. In order to get proper operation the FL/DIR pin of FIFO #1 is kept low by wiring it to logic '0' and the FL/DIR pin of FIFO #2 is fed with the inverse of the reset signal.

After reset the write and read pointers are set to the first location in the FIFO. The full and half flags are high and the empty flag is low. From this point the FIFOs can be filled by presenting data on the 32-bits bus and issuing an active low write pulse. Data set up and hold times are with respect to the trailing rising edge of the write pulse. The FIFOs are serially read by issuing a series of serial output clock pulses, active high, on the SOCP pin. For further details refer to the datasheet. It can be concluded that an architecture of two 72105 width expanded FIFOs can fill the needs of one scanline buffer for the reader.
11. bitmap reader interface derivation and global behavior

Chapters 3 to 6 thoroughly treated the TMS34020 processor. After this the design requirements for the RIP unit design were described. Chapter 8 studied the possible memory architectures that met the stated requirements. Then the concept of having a multiprocessing system with shared memory and busses was introduced. The concept of scanline buffering with its performance and an architecture with FIFOs to realize scanline buffering was further elaborated.

Now it is time to translate the requirements for the bitmap reader part of the RIP unit into a specification of a bitmap reader block with all the necessary interfaces. Since the other parts of the RIP unit are known the derivation of the readers interfaces for the most part can be carried out by making use of the principle of interfacing by mirroring. This is a method often used in System design [KOO]. When the interfaces are specified the global behavior of the reader will be explained with help of a flow chart.

11.1. Programmability

As stated in paragraph 7.4 the TMS34020 must be able to program the bitmap reader with a start address, the length of one scanline and the number of scanlines. Furthermore the GSP has to issue a start command to the reader. Also the GSP has to inform the reader whether the bitmap has to be read and cleared (read-modify-write) or only read (bitmap is saved for multiple printing). For this programmability first an 8-bit input bus has been specified for the reader (D[0:7]). In order for the reader to be selected an input select line has been added, Reader Chip Select (RCSn). For the data to be written into the reader STROBEn has been specified. The line RDWRn is an input read/write line although for now the reader will only be written to. For the reader to decode what kind of data is presented and where to store it four input address lines are defined, SELECT[3:0].

The timing for the interface can be identical to a DRAM early write cycle. The RIP unit’s decoding should activate RCSn based on a RASn pulse from the TMS34020 and a specific memory area. In this way the reader becomes a memory mapped device. STROBEn and RDWRn can be connected to the systems CASn and WEn lines. D[7:0] and SELECT[3:0] may be connected to the data bus and the latched address.

The program interface must also be used by the GSP to signal the bitmap reader that a page has been rasterized and that it may start filling the first FIFO and deliver the page to the printer engine. This start command is a dummy write to some address for the GSP.

11.2. Bus control interface

In order for the reader to get ownership of the system busses and memory two lines are defined, a request line (REQn, output) and a grant line (GRANTn, input). When the reader needs the bus it activates REQn. Upon GRANTn, issued by an arbiter, the reader can start accessing the system memory that holds the bitmap. Furthermore the data read out of the memory can be stored in one of the FIFOs.

When the reader has finished his actions on the memory it must inactivate REQn. The arbiter then can give ownership back to the 34020 and inactivate GRANTn. It should also be possible for the arbiter to inactivate GRANTn while the reader still holds REQn low. This way the reader can temporarily be put off the bus.

11.3. Engine interface

The reader interfaces to the engine with three inputs, SOP, SOS, and BURST and two outputs, PAV and IMDA. After the reader has filled the first FIFO as a result of the start command from the 34020 it can make PAV high and subsequently synchronize to the first SOP. Every following SOS the reader resets the FIFO that it has to write and it enables the BURST signal to the serial clock input of the ap-
propriate FIFO that then must be read. The reader passes the serial output of the FIFO being read over the IMDA line. In the meantime the reader owns the bus for filling the other FIFO.

11.4. DRAM interface

To be able to be fitted in the multiprocessing system together with the TMS34020 and the 16 MByte DRAM memory, the reader must have an interface to the common busses that is identical to that of the 34020. Therefore the following outputs are defined: RASn, CASn, WEn, RCA[10:0] and AD[4:0].

RASn is the Row Address Strobe that signals the DRAMs that the row address on the Row Column Address bus is valid and can be used. CASn, the Column Address Strobe does the same for the column address that is now on the RCA bus. WEn goes low half way the active low CASn pulse in case of a read-write cycle. This cycle is used as stated earlier for reading and subsequently clearing a memory location. WEn goes inactive together with CASn. In case of a read cycle WEn remains high.

The RCA bus has eleven lines so the reader can address 4Mbit DRAMs. To be able to address all possible addresses in the memory space of the TMS34020 the AD bus has five lines. These represent the five most significant bits of an address and can be used for selection of a particular 16 Mbyte memory bank.

11.5. FIFO interface

Given the architecture of a 256x32 FIFO, built up by width expansion of two 256x16 parallel-to-serial FIFOs the signals that the reader must have for controlling two of these FIFOs can be defined.

For resetting FIFO 1 and FIFO 2 the output signals RS1n and RS2n are defined. For controlling the FL/DIR pins of the FIFOs the reader has outputs FL1 and FL2. For the control of writing to the FIFOs signals W1n and W2n are defined.

Outputs SOCPl and SOCP2 are used by the reader to clock data out of the appropriate FIFO based on the BURST input of the engine interface. Finally inputs SO1 and SO2 are used to direct the data of the FIFO being read to the IMDA output.

11.6. Clock inputs, interrupt and reset

To clock the reader synchronously to the TMS34020 two clock inputs are defined: CLK1 and CLK2. These clocks can be used for inputting LCLK1 and LCLK2 of the 34020 [GSPUSR].

An interrupt output is needed for the reader so it can interrupt the 34020, signalling that the reader has finished reading a page and passing it to the engine. The 34020 can acknowledge this interrupt by writing to an address that the reader can decode via its program interface.

A reset input (RST) is defined for resetting the reader and bringing it in a known state.

11.7. Reader block diagram and flow chart

Figure 11.1 shows all the resulting interfaces for the bitmap reader. In appendix B a flow chart is given that explains the functioning of the reader.
Figure 11.1: Symbol and and interfaces for the bitmap reader
12. Top level RIP unit architecture

12.1. Hierarchical design methodology using the Viewlogic CAE environment

Now that the considerations about the requirements of the Raster Image Processing unit have been made it is time to draw up the resulting system architecture. For this purpose and for further design work the Viewlogic CAE environment will be used.

Viewlogic is a powerful design tool that allows hierarchical design and the use of VHDL. VHDL (VHSIC Hardware Description Language) is an IEEE standard hardware description language [VHDL]. Within the Viewlogic environment schematic entry is possible with Workview. Workview allows the creation of symbols that can be used in a schematic. The behavior of symbols can be specified by Viewlogic's subset of VHDL that allows behavioral modelling. Such symbols are said to be modular. Also symbols can contain a schematic at a lower level that determines the behavior. Such symbols are called composite and they introduce hierarchy.

With Viewlogic's Viewsim simulations can be carried out on any part of a design. Functional simulation is used to check if the design is functionally correct. With Viewwave timing diagrams of the simulations can be created.

VHDL code can be checked with Viewlogic's VHDL analyzer program. Viewlogic also supports logic synthesis and schematic generation out of VHDL code for specific PLD's such as LCA's, FPGA's or even ASICs.

In this report now the system architecture that has been created with Viewdraw will first be described. The different functional blocks that this toplevel design consists of will be treated in some detail such that the implications for the architecture from the considerations in the former chapter become clear. The report then concentrates on the further top-down design of the bitmap reader block that is in the system architecture. It is this work that will show how hierarchical design and the use of VHDL can nicely be used to specify a design.

Furthermore it will be demonstrated how logic synthesis out of VHDL code translates the functionality that the designer desires and specifies into logic gates. Also it will be seen that simulation can be used to check the desired functionality of the design and finally this report will focus on a possible implementation trajectory for the bitmap reader design.

The advantages of this design methodology are:

- Necessitates full functional specification
- Use of hierarchy keeps design comprehensible
- Functional verification by simulation on various hierarchical levels
- Recognition of design errors in an early stage
- Alternative designs are easily generated and integrated
- Technology independence
- Well documentation
- Allows logic synthesis
- Forms step towards real ASIC design

12.2. The architecture

Appendix C.1 shows the architecture of the RIP-unit. The RIP unit has been designed as a multiprocessing system. It can be regarded as a system that has two processing entities, the TMS34020 and the bitmap reader, that share the same system memory through three shareable busses. These busses are the local address data bus (LAD-bus), the Row Column Address bus (RCA-bus) and a bus for the control of the DRAMs, the system's address latch and the system's transceiver. This bus con-
Top level RIP unit architecture

contains the Row Address Strobe (RASn), the Column Address Strobe (CASn), the Write Enable (WEn) signal, the Address Latch (ALTCHn) signal and the Data Direction In (DDIN) and Data Direction Out (DDOUTn) signals. This shared bus will be referred to as the control bus.

It should be noted here that the architecture focusses on how the TMS34020 interfaces to the system memory, the arbiter, the decoder and the reader. Interfacing to a host (whether VME or not) falls out of the assignment’s scope. See chapter 3 and [GSPUSR] for general host interfacing. The next paragraphs will describe the various functional blocks of the system.

12.3. The TMS34020

The TMS34020’s local memory and DRAM interface directly connects to the LAD-bus, the RCA-bus and the control bus. To be able to operate in a multiprocessing environment the 34020 is connected to the arbiter with the R0, R1 and GI lines. Pins R0 and R1 output a request code according to the operation of the 34020.

Depending on the arbiter’s arbitration scheme the grant in (GI) line of the GSP is made active or inactive. When GI is sampled to be inactive by the 34020 it sets all it’s interface lines to the busses to inactive and finally to high impedance (the 34020 has tristate outputs for this purpose). See further chapter 3 and [GSPUSER].

The 34020 uses two internal clock signals LCLK1 and LCLK2 for its operations. These clocks are derived from the clock signal that is offered via the CLIN pin (32 or 40 MHz). For multiprocessing purposes these clocks are output on the 34020’s LCLK1 and LCLK2 pins. In the architecture these clock signals are being used to clock the bitmap reader.

12.4. System memory, mirror function and transceiver

In chapter seven it has been decided to implement the system memory with 4Mx1 DRAMs. 32 of these devices (32 bit wide bus) make up the basic memory block of 16 Mbyte. The architecture of appendix C.1 shows one such 16Mbyte block. The underlying architecture can be seen in figure 8.2. The RASn, CASn, WEn, and RCA lines are connected to the shared bus so 34020 or bitmap reader can access the memory.

The memory block has a 32 bit wide bus DOUT on which data is output by the DRAMs. The DIN bus is the input bus to the DRAMs. In order to have the mirror function the DOUT bus is lead to a functional block. Based on the value of linear address bit 31 this block puts the DRAM data through normally or mirrored. By mirrored a swap between bits 31 and 0, 30 and 1, 29 and 2, and so forth is meant. This mirror function divides the systems memory map in a mirrored and an unmirrored half. Bit 31 = ‘0’ means mirrored and bit 31 = ‘1’ means unmirrored. As will be seen in 12.5. the memory map for the 34020 will have the lower half of the locations occupied with a mirrored copy of the unmirrored upper half.

In order for the data busses to connect to the bidirectional LAD-bus a transceiver block is specified. The transceiver is controlled by the DDIN and DDOUTn signals. The 34020 activates DDOUTn when it performs a write action on the memory. Data from the LAD-bus is put on the DIN bus. In the case of a read from memory the GSP activates DDIN, directing data from the DOUT bus via the mirror block onto the LAD-bus. Refer to [GSPUSR].

12.5. Address latch and decoder

The LAD-bus contains the linear address of the memory location that needs to be accessed during the first part of a memory cycle. With ALTCHn this address is latched into the address latch. This way the address is available for the decoder and the mirror block during the whole access period.
Based on the bits 27 to 30 a selection can be made out of 16 banks of 16Mbyte. With the RASn, ALTCHn and LAD27-30 the appropriate RASn output from the decoder can be asserted. For now only 8 RASn lines are specified. One for the 16 Mbyte memory block of the architecture and 7 for future piggy-backed memory banks.

To select the bitmap reader for programming (for instance with the start address of a page) the decoder has the Reader Chip Select (RCSn) output. This implies that one address range of 16 MByte can be used to map the reader in the memory space of the 34020.

For resetting the bitmap reader the decoder has the RST output. This occupies in this case a full 16 Mbyte memoryblock since only the 5 most significant bits are decoded. With one 16 Mbyte block occupied by reset and one by the mapping of the reader's registers there are however still 14 16 Mbyte block available for program memory and bitmaps.

In order to recognize a CAS-before-RAS refresh cycle [GSPUSR] the address bits 0 to 3 must be decoded also every cycle. These bits contain the status code of the access under execution. In case of a refresh cycle these bits have the refresh code. The decoder must activate all RASn lines in this case in order to refresh every bank of memory. The memory map for the system is given in figure 12.1.

The memory map for the TMS34020 specifies that the highest 256 32bit words from address FFFF E000h (bit addresses) to address FFFF FFE0h are for interrupt and trap vectors. In the system these addressess must be mapped into the 16 Mbyte DRAM block. This is the reason why the system
memory is mapped on the highest addresses, from F800 0000h to FFFF FFE0h.

The area from 8000 0000h to 87FF FFE0h is used to map the reader. For resetting the area 8800 0000h to 8FFF FFE0h is used. Piggy-back memory can be mapped freely in the area 9000 0000h to F7FF FFE0h with the notion that on addresses C000 0000h to C000 1FE0h the on-chip I/O registers of the 34020 will be “shadow-mapped”. This area can therefore not be used for placing data.

The area 0000 0000h to 07FF FFE0h is the mirrored copy of the upper half of the memory map.

12.6. bitmap reader and FIFO’s

The bitmap reader is the other processing entity in the RIP unit’s system architecture. With its data input bus D[7:0] it is connected to the 8 least significant bits of the LAD bus. RCSn is connected to the decoder’s RCSn. The SELECT bus connects to the latched address bits 5 to 8. This is because the 5 LSBs of the LAD bus contain the status code during the address cycle rather than address information. The actual 27 bits address is on bits 5 to 31 [GSPUSR]. STROBEn is connected to CASn0 of the 34020 and RDWRn is connected to the WEn of the 34020.

With the bus control interface (11.2) the reader is wired to the system’s arbiter. The DRAM interface (11.4) is connected to the three busses. The two clock inputs and the interrupt line directly connect to the 34020. Reset finally is connected to the decoder output RST. This way the reader can be reset by a memory write from the TMS34020.

The width expanded FIFO architecture of figure 10.3 is represented by a block “256x32 parallel to serial FIFO” in the architecture of figure 12.1. Two of these blocks represent the two scanline buffers. With their respective RSn, Wn, FL/DIR and SOCP inputs, these scanline buffers are connected to the bitmap reader’s FIFO interface (11.5). The outputs SO from the buffers are connected to the bitmap reader’s inputs SO1 and SO2. The bitmap reader will have logic that combines these two inputs to form IMDA. The 32 bit wide data input busses to the buffers reside on the system bus DATA[31:0] so data to the buffers skips the transceiver. Effectively this means that when the reader owns the LAD bus it need only use this bus to issue the address for the decoder. The LAD bus is not used for transferring data and the transceiver can be controlled by the reader’s DDIN and DDOUTn to be inactive.

A functional pullup block is connected to the DIN[31:0] bus, i.e. the data input to the memory. When the reader owns the busses the DIN bus isn’t driven by the transceiver. The pullup block keeps the DIN lines at a logic high level. In case of read/write cycles by the reader the DRAMs will therefore be "cleared" with logic ‘1’.

12.7. Design notes

Chapter 13 will show that the reader combines the inputs SO1 and SO2 simply by ANDing them. The result is IMDA. The specification for the printer engine’s LSM requires that the IMDA line is at a logic ‘1’ when no BURST pulses are issued to the reader. ANDing SO1 and SO2 to get IMDA therefore only is correct when SO1 and SO2 are high when the FIFOs are not being clocked by SOCP1 and SOCP2. The datasheets for the 72105 FIFOs however specify that the serial output SO is at high impedance when there is no clock signal on the SOCP input. Therefore a pullup must be connected to the serial output of the FIFO architecture of figure 10.3.

A second note is on the specification for “white data” as it is understood by the LSM. It is specified that logic ‘1’ represents white data and logic ‘0’ represents black data. This is just the inverse of how the 34020 defines black and white! For the GSP logic ‘1’ represents black and logic ‘0’ represents white.
The problem is solved when the systems transceiver inverts the data when it is transferred from LAD bus to DIN bus and from DATA bus to LAD bus. This way data written to memory by the TMS34020 is invertedly placed into the DRAMs and when it is read again by the ’20 the inverted data is again inverted. The GSP doesn’t notice the inversion.

The reader however fills the FIFOs with inverted data and the DRAMs are cleared with ’1’s (that will be read as ‘0’s by the GSP), which is exactly what must happen.
13. bitmap reader architecture

The bitmap reader has been specified by its interfaces to the rest of the Raster Image Processing Unit and a flow chart that describes the behavior globally. Whereas all the other blocks of the RIP unit have been described in a sufficiently detailed way, the bitmap reader itself needs a more detailed specification. In this chapter the bitmap reader design will therefore be carried out. Chapter 14 will then give a possible implementation trajectory for the bitmap reader.

13.1. A detailing step

One possible strategy for further specification of the bitmap reader could be to fully describe the behavior with VHDL of the created symbol for the reader. There are several reasons however for not using this quite rigorous strategy. First of all Viewlogic's VHDL does not have the possibility for structural modeling (apparently the newest releases do have this possibility). Structural modeling is possible with standard VHDL and it allows the designer to specify different instantiations of various entities within the code for one component at a higher level. Not having this possibility means that a VHDL description for the reader should fully describe the behavior without further use of hierarchy. This would be too complex a task.

Furthermore it is desired that the possibility for logic synthesis from a VHDL description can be used. Writing synthesizable code puts restrictions on what constructions and syntax of VHDL one can use. This complicates the description of the bitmap reader with one piece of VHDL code even more.

Given the above argumentation it is wise to use hierarchy in the design of the bitmap reader by using schematic entry in Viewlogic's Workview. This is done by taking a step down into hierarchy towards the architecture of the bitmap reader. Separate functional blocks are created by designing their symbols with the symbol editor of Workview. These functional blocks with their interfaces make up the bitmap reader's architecture schematic. The functional blocks must be of a complexity that is suited for specification with synthesizable VHDL code. The result is a full specification of the bitmap reader that consists of its interfaces at the toplevel RIP unit design, an architecture at one lower level built up of functional blocks, a specification of the interfaces of the functional blocks at that level and a specification of the behavior of these blocks in synthesizable VHDL. Thus the behavior of the reader itself is known and the design is tailored to enter an arbitrary implementation trajectory by logic synthesis to programmable logic.

Figure 13.1 shows the design flow for the detailing step in the bitmap reader design.
One exception to this strategy is made there where a functional block's behavior can be easily specified by a schematic design at yet one lower level instead of by a VHDL description. This will be the case for three functional blocks from the reader that will realize the programmability of the reader and the page dependent control for the rest of the blocks. The next paragraph will show that standard programmable counters can do this job.

### 13.2. bitmap reader programmable control

In order for the reader to be programmable with the start address of a bitmap, the number of memory accesses required to fill one of the FIFO's and the total number of scanlines in a bitmap three functional blocks have been created. The first block is a programmable 32 bits synchronous counter with a 32 bits register that supplies the 27 bits addresses for the memory accesses of the reader. This block will be referred to as the address counter.
The second block is an 8 bits loadable synchronous down-counter with an 8 bits register. This block can be programmed with the number of access per scanline and counts down on every access of the bitmap reader to the memory until zero is reached. This block will be referred to as the scanlength counter. On zero this counter activates a To Carry (TC) signal that is used to enable the third counter block and to signal the end of scan (ENDOFSC) to the functional blocks of the reader.

The third block is a 16 bits loadable synchronous down-counter with two 8 bits registers. This counter can be programmed with the number of scans and counts down every time the scanlength counter has reached zero. It will be referred to as the nrofscan counter. In fact this is the second part of the 24 bit counter that the scanlength counter and this counter form together.

When the nrofscan counter reaches zero and the scanlength counter does so too the nrofscan counter activates its TC signal that is used to signal the end of page (ENDOFPG) to the reader system. Figure 13.2 shows the three counter blocks and their connections. The underlying schematics that are created with Workview can be found in appendices C.2 to C.4. The counter and register macro’s that have been used have been taken from the Xilinx LCA 3000 series macro library. This used library (and therefore the implementation technology) could have been any library but since the final implementation trajectory that will be treated in chapter 14 is based on Xilinx LCA’s this library was used.

In figure 13.2 it can be seen that a control register has been added. This register can hold information on whether the bitmap reader should read and clear the bitmap or only read it. The names of the input and output lines are added for simulation. In the bitmap reader architecture that will result from the composition of all functional blocks these names may change. SAS3 to 0 are four start address strobes for loading the address counter’s registers. NOSPE is the Number Of Scans Parallel Enable that is used for loading the address and nrofscan counters from their registers. CEN is the
counter enable. The scanlength counter is loaded with ScanLength Strobe (SLS) and parallel enabled with SLPE. The number of scans counter is loaded with NOS0 and NOS1 and finally the control register is loaded with CTL.

It is important to verify if the counters behave functionally correct. For this purpose the architecture of figure 13.2 has been simulated with Viewlogic's Viewsim. By specifying stimuli to the input signals of the network in a command file and executing the file in Viewsim it was tested if the programmed data was stored correctly, if the address counter supplied the appropriate addresses and if the nrofscan and the scanlength counters correctly counted down and generated ENDOFSC and ENDOFPG. The simulation data have been made visible with Viewwave. Viewwave generates timing diagrams for specified signals of the network. Figure 13.3 shows the simulation result.

In this simulation the counters are programmed with start address 0000301h, a scanlength of 8 (value in register is 07h) 32 bit accesses and a number of scans of 3 (value 0002h in the register). The control register is programmed with 01h, leading to a '1' on the CLEAR line, indicating that the reader should only read and not write.

It is seen that when after the programming sequence CEN is activated the address counter starts counting on the rising CLK edge. The scanlength counter counts down from 7h to 0h. When it is 0h it activates ENDOFSC. Through ENDOFSC the nrofscan counter counts down once on the next rising CLK edge. SLPE high resets the scanlength counter (CEN has been inactivated) and after CEN is activated the same actions repeat. At the end of the third count sequence ENDOFPG is activated as expected.

At this point the programmable and controlling heart of the bitmap reader has been fully specified by interfaces, architecture and functional behaviour and the design has been verified to be functionally correct.

13.3. Writing synthesizable VHDL code

The remaining functional blocks of the reader will be specified by their symbol with interfaces and a behavioral VHDL file. This paragraph describes how synthesizable VHDL can be written.
Logic synthesis with Viewlogic’s VHDLdesigner (VHDLdes) translates VHDL source into optimized digital logic. There are certain guidelines when composing VHDL source for synthesis. This is because there are certain legal VHDL constructs that have no counterpart in digital logic. Also some VHDL constructs may only be used in a restricted manner. See for a full description [VDES]. Some aspects of the restrictions will be described and two strategies for composing VHDL source will then be treated.

13.3.1. Objects and data types
Objects in VHDL are identified by their:
- Class
- Name
- Data type
- Value - depending upon the datatype

Legal object classes for VHDLdesigner are: constant, signal and variable. Data types that are supported are: boolean, vlbit and vlbit_vector. Under very restricted scenarios integers and characters are allowed. Vlbit is the Viewlogic version of the VHDL standard type bit. Vlbit_vector is an array of vlbit.

The restrictions on the data types are caused by the fact that the VHDL source must be projected on digital logic. This means that the source must be written with the common apprehension of digital systems in mind.

Constants only accept preassigned values. They are declared in the source file and their value never changes. In synthesis constants are equivalent to connections to logical ‘1’ and logical ‘0’. Constants are optimized during synthesis.

Variables can only be used in sequential statements. They must be declared in and be local to the processes in which they are used. Sequential and concurrent statements and processes will be treated in the next paragraph.

13.3.2. Processes, sequential and concurrent statements
In a VHDL source the behavior of the described component can be specified by one or more processes. Processes can be mixed with single statements.

A process is a group of statements that are executed sequentially. After the last statement of a process is executed the process loops and starts execution again from the first statement. Part of the definition of a process must include statements to suspend execution until some event occurs, otherwise it would endlessly loop. Different processes execute concurrently.

A combinational process has a sensitivity list that is included with the processes declaration. In this list all the signals are listed that are used in the process. A change on one of these signals causes the process to execute. Otherwise the process is suspended. A clocked process has one and only one WAIT statement.

A clocked process is suspended until the condition of the WAIT statement becomes true. Clocked processes are often used for the description of edge triggered clocked systems. Within the behavioral description of a component processes and single statements all execute concurrently. The statements within a single process execute sequentially.

13.3.3. Boolean equations
The first strategy for writing synthesizable VHDL that is described is suited for specifying combinatorial circuits. Such a circuit has outputs that are a function of the inputs. When these logic functions
are reduced and written in Boolean equations the VHDL source is easily composed. Suppose the circuit is a simple AND gate with inputs A and B and output C. The equation for C then is:

\[ C = A \cdot B \]

The VHDL source that specifies the AND gate is:

```vhdl
ENTITY and IS
PORT ( A, B: IN vbit;
       C: OUT vbit);
END and;

ARCHITECTURE behavior OF and IS
BEGIN
    C <= A AND B;
END behavior;
```

The first part declares the entity AND with its input and output signals. Then the behavior is specified in the ARCHITECTURE part by simply stating one concurrent signal assignment: \( C = A \) and \( B \). See [VDES] for an example of a one bit adder. VHDL descriptions of combinatorial circuits in this manner will always be synthesizable, which is trivial because the description implements the apprehension of digital logic.

### 13.3.4. State Machine with synchronous reset and registered output

The second strategy for synthesizable VHDL is recommended to be used in the design of synchronous, clocked systems. For the design of such systems often first a state diagram that describes the system is designed. This state diagram can be translated into VHDL source. The way of doing this in order to get synthesizable VHDL is the following.

Let's look at the symbol that defines the interfaces of a tristate controller (figure 13.4).

![Figure 13.4: Interfaces of the tristate controller](image-url)
Upon an active high reset on RST it inactivates the active low output GIn and it activates the active high TST. GIn is a grant signal for systems that start operating upon a grant. TST is a signal that controls tristate output buffers surrounding the systems. An active high on TST puts the buffers in tristate. When the GRANTn input is active low the controller inactivates TST and activates GIn. When GRANTn goes high again the first clock cycle of CLK1 the GIn is inactivated, enabling the other systems to go inactive. The next cycle also TST is activated, putting the output buffers in tristate.

The state diagram for the tristate controller is given in figure 13.5.

![State diagram for tristate controller](image)

In the diagram the states are denoted S0, S1 and S2. With the transitions the condition for the input GIn is given left of the slash. To the right of the slash the corresponding output vector (TSTn, GRANTn) is stated. The state diagram is constructed according to the concept of the Mealy machine. In effect this means that the state vector does not necessarily equal the output vector.

The VHDL source for the tristate controller is given in appendix D.1 (tstctl.vhd).

Looking at this code it is seen that first the entity with its inputs and outputs is defined. Then in the ARCHITECTURE part the type state_vec and the signal cur_state of that type are defined. The following process control first defines the variable next_state that is local and furthermore the constants for the different state names. Very important is the definition of the constant unknown that will be used in the final else statement of the process.

After these declarations the process begins with a WAIT statement that suspends the process until the CLK1 signal goes to logic ‘1’. This causes the controller to take the following actions on every rising CLK1 edge: first RST is tested to see if the controller should go into the S0 state (synchronous reset). Then in the following ELIF’s the controller determines in what state it is and this state combined with conditions on the inputs causes an assignment to next_state and the outputs.

The last ELSE statement cannot be omitted since the controller must be able to catch an unknown state. Omission of this ELSE leads to synthesis of unnecessary logic [VDES]. After the IF and ELIF statements the next_state is latched into the current state. This causes the actual state transition.

Describing the controller in this way causes the synthesizer to generate two flipflops that contain the current state of the state machine and a flipflop for every output of the controller. The system therefore will be fully synchronous, clocking the outputs every rising CLK1 depending upon the current
state of the state machine and clocking in the next state at the same time. See appendix F.5 for the synthesized logic.

13.4. Reader decoder

In order to write data into the bitmap reader registers a functional block is created that decodes the four select lines SELECT[3:0]. Based on the address on SELECT and the inputs RCSn, STROBEn and RDWRn this reader decoder must issue load pulses to the registers of the address counter, the scan-length counter and the nrofscan counter. Furthermore a load pulse is required for the control register. The registers need an active high asynchronous pulse so that they be able to clock in the data that is presented to their inputs. This data must be presented on the DATA[7:0] input bus. The pulses have the same width as the pulses on STROBEn which in fact are CASn pulses from the 34020.

The reader decoder also must decode a start cycle and an interrupt acknowledge cycle. The 34020 performs these cycles simply by performing dummy write operations to the corresponding addresses. The width of the decoded START and INTACK pulses are derived from the pulse on RCSn. This is done because these pulses must be recognized by the clocked functional blocks of the reader. The width must therefore be equal to or greater than the period of the clock to these blocks. Since RCSn is derived from RASn it will be at least one LCLK2 period long. As will be treated later the reader’s clocked blocks will be clocked by LCLK2 on the reader’s input CLK2. Therefore the START and INTACK pulses can be recognized.

Figure 13.7 shows the reader decoder with it's interfaces. Also the truth table is given for the decoding of SELECT, RCSn, STROBEn and RDWRn.
It is seen that the input lines to the block are RCSn, STROBEn, RDWRn and SELECT[3:0]. The outputs comprise four Start Address Strobes (SAS0 to SAS3) for the bytes 0 to 3 of the 32 bits address counter’s register, one Scan Length Strobe (SLS) for the eight bits register of the scanlength counter, two Number Of Scans strobes (NOS0 and NOS1) for the bytes 0 and 1 of the nrofscan counter and a Control (CTL) strobe for the control register. Byte 0 is always the least significant byte. For interrupt acknowledge and starting the reader the INTACK and START outputs are defined.

The specification of the decoder’s behavior is given in the file readdecode.vhd that is listed in appendix D.2. The decoder is described as a combinatorial circuit with the proper Boolean equations.

The reader decoder has been functionally simulated, the results of which are given in the functional timing diagram of appendix E.1. From the diagram it is seen that a standard DRAM timing can be applied to the decoder.
13.5. Requester

The requester symbol has been defined as showed in figure 13.8.

![Requester symbol diagram]

Clocked on the input CLK2, falling edge, the requester activates the REQn output after a START or a SOS, after synchronization with SOP, on the inputs. When ENDOFSC is detected REQn is inactivated again. The requester controls the parallel enables of the three registered counters. NOSPE (Number Of Scans Parallel Enable) controls both the address and the nrofscan counters. SLPE (ScanLength Parallel Enable) controls the scanlength counter. Between RST (ReSeT) and START, NOSPE is kept active high and SLPE is kept active high, enabling the counters to clock in the data. Furthermore between a detected ENDOFPG and a new START the outputs NOSPE and SLPE are active. In this way the start address for a new page and the scanlength and number of scans can be clocked in again into the counters. Between every ENDOFSC and a synchronized SOS only SLPE is kept active high to clock in the scanlength for the next scan, i.e. the scanlength counter is reset to the scanlength.

The requester additionally controls PAV for the engine interface (see 11.3 and 10) and it controls INTn for interrupting the 34020 when a page has been passed to the engine. Upon detection of ENDOFSC after the START pulse the reader has one FIFO full. Therefore the requester then issues PAV high. After ENDOFPG PAV is inactivated again and the requester activates the active low INTn. Although one FIFO is then still to be read by the engine the 34020 could already order the reader to start filling the other FIFO for the next page. After an INTACK the requester inactivates INTn.

Figure 13.8 show the state diagram for the requester in which it's functionality as described above can be recognized.
In the diagram with every transition the values of the input vector, i.e. the condition for the transition, is given before the slash. After the slash the output vector values are given. The input vector is: (START, SOP, SOS, ENDOFSC, ENDOFPG, INTACK). The output vector is: (REQ, SLPE, NOSPE, PAV, INT). The VHDL description is given in the file requester.vhd which is listed in appendix D.3.

The method of constructing VHDL for state machines with synchronous reset from paragraph 13.3.4. has been used to compose requester.vhd. A five bit vbit_vector outputs is used within the process request to assign outputs to from the state machine. Outside the process the assignments from specific bits of outputs to the real outputs of the requester are stated.
Using this behavioral model the requester has been simulated with Viewsim. The results are in appendix E.2.

The proper functioning can be verified to be correct from the functional timing.

13.6. DRAM controller

Since the reader must access DRAM memory through the DRAM interface (11.4.) a block is created that performs this functionality. See figure 13.10 for the interface specification of the DRAM controller.

The controller is clocked by CLK1 and CLK2. CLKINT1 and CLKINT2 are added for later implementation purposes on LCA's. They are connected to the same inputs of the reader as CLK1 and CLK2 (the CLK1 and CLK2 inputs of figure 11.2) and are used for combinatorial functions inside the DRAM controller rather than for clocking a state machine. After reset (RST) the DRAM controller is inactive. Upon GRANTn low the controller begins performing DRAM accesses. When GRANTn goes high again before ENDOFSC is detected to be high the controller temporarily goes inactive until GRANTn is low again. This gives the RIP unit's decoder the opportunity to withdraw bus mastership from the bitmap reader.

When after GRANTn low ENDOFSC becomes high the DRAM controller will go inactive and will wait for GRANTn to go high. The controller is then in the same state as after RST.

When performing DRAM accesses (active) the DRAM controller has three tasks. First it must issue RASn, CASn, WEn and ALTChn. Second it must multiplex the 22 least significant bits of the address counter, on input bus ADRIN[26:0], into a row and column address on the RCA[10:0] bus and it must output the five bits 22 to 26 on the A[4:0] bus for decoding by the RIP unit's decoder. Third the DRAM controller must enable the three counters of the reader according to the accesses it performs.

DDIN and DDOUT will be '0' and '1' constantly since the RIP unit's transceiver block must be disabled when the reader accesses memory.
13.6.1. DRAM functional timing

The reader accesses DRAM in page mode for speed purposes. This means that first the row address is output together with RASn going low. Subsequently only column addresses are output with CASn and WEn (in case of read-write) pulses. The number of subsequent column accesses is limited to 256 so one series of page mode can be as long as 256 accesses. This is denoted with a page size of 256. (Although the specification of the DRAMs that are planned to be used allows a page size of 4096 each factor two multiplication of the page size will ask for extra logic in the implementation. Therefore it was chosen to set the page size to 256)

To get a clear idea of how the DRAM controller must control it’s output signals figure 13.11 shows how the timing of RASn, CASn, WEn and ALTChn must be with respect to two clock signals CLK1 and CLK2. Note that CLK1 and CLK2 will be connected to the systems LCLK1 and LCLK2. Furthermore figure 13.11 shows how the row and column addresses must be multiplexed in time and how the address counter that supplies the linear address to the input of the DRAM controller must be enabled and disabled (i.e. when it must count).

![Figure 13.11: Functional DRAM timing](image)

In figure 13.11 it is seen that the address counter and therefore the other two counters in the reader must count on the falling edge of CLK2. Remember that the counters themselves are positive edge triggered. Therefore they must be clocked with the inverted CLK2 signal. Figure 13.12 shows that RASn, like the counters changes with the falling edge of CLK2. It can be seen that the counters must start counting on the falling edge of CLK2 when RASn is low and that they stop counting when RASn is high. Given the fact that the counters are enabled by an active high signal an important conclusion is that the inverted RASn signal can be used in the reader to enable or disable the three counters.

Due to the fact that both CLK1 and CLK2 are involved in the clocking of the DRAM controller two statemachines will be designed to further specify this controller. One state machine will be based on CLK1 rising and will be treated in 13.6.2. and one state machine will be based on CLK2 falling which will be described in 13.6.3. The resulting VHDL behavioral model for the DRAM controller will therefore contain these two clocked processes. Finally the functional timing diagram shows that CASn, WEn and ALTChn can be created by combinatorial functions of the clocks and RASn. The VHDL model will therefore contain Boolean equations that specify these signals. (Remember this is the reason that CLK1INT and CLK2INT were added to the DRAM controller).
13.6.2. Address multiplexing
From figure 13.11 it is seen that the multiplexing of the linear address is clocked by CLK1. When RASn is high the row address is output every rising edge. When RASn is low the column addresses are output every rising edge. Furthermore the 5 MSB's of the 27 bit linear address are output on CLK1 rising when RASn is high. From this observation the VHDL code for address multiplexing is derived. In the file rascas.vhd (appendix D.4), which describes the behavior of the DRAM controller, this code is implemented as a clocked process with name addrmux. On CLK1 rising, this process executes one if statement that checks the logic level of a signal called rasint.

Rasint is generated by a second process from rascas.vhd which will be treated in 13.6.3. Rasint is an internal signal in the DRAM controller of which among others RASn will be derived (RASn will be equal to rasint). Based on the level of rasint the process addrmux outputs the proper address bits of the input bus ADRIN on RCA[10:0] and A[4:0]. The process in fact describes a multiplexer with clocked outputs.

13.6.3. Generation of RASn
The state machine that is clocked by the falling edge of CLK2 generates rasint as an output based on the inputs RST, GRANTn and ENDOFSC. Furthermore rasint is determined by the 8 LSB's from the linear address on the ADRIN input bus. Namely, when all these bits are one this indicates that the next address will lie in another 256 page. Rasint must then go high in order to end the current page mode sequence and to allow for the address multiplexer to output a new row address. After one clock cycle rasint can go low again. It should be clear that rasint is directly assigned to the RASn output. Figure 13.12 shows the state diagram.

The VHDL code for this state machine can be found in rascas.vhd within the process rasgen.

13.6.4. Combinatorial assignments
For the outputs of the bitmap reader's DRAM controller the following equations are used:

\[ \text{RASn} = \text{rasint} \]
\[ \text{CASn} = \text{rasint} \text{ OR NOT CLK2INT} \]
\[ \text{WE} = \text{CLEAR OR rasint OR CLK1INT OR NOT CLK2INT} \]
AL TCH = rasint
DDIN = '0'
DDOUT = '1'

RASn and ALTCHn are equal to rasint that is generated by the VHDL process rasgen. For CASn rasint is combined with the CLK21NT signal. WEn is determined by rasint, CLEARn (from the reader’s control register) and the two CLK signals. CLEARn = '0' means that the DRAM controller has to perform read/write cycles. Finally DDIN and DDOUT are at a constant level that will inactivate the RIP unit’s transceiver bothways.

Within the ARCHITECTURE part of the file rascas.vhd the Boolean equations are specified as concurrent signal assignments, just before the description of the processes addrmux and rasgen.

13.6.5. Simulation result of the DRAM controller
The DRAM controller block has been functionally simulated. The results are in appendix E.3. It is seen that the VHDL description functionally realizes the same timing as the DRAM timing from figure 13.10.

13.7. FIFO controller

For the bitmap reader to be able to control the writing to and reading from the two FIFO’s in the RIP unit a functional block has been defined. This FIFO controller is shown in figure 13.13.

![Figure 13.13: Symbol for the fifo controller](image)

Like the requester and the RASn generating part of the DRAM controller, the FIFO controller is clocked by the falling edge of CLK2. Upon a detected active high reset (RST) the FIFO controller comes in an inactive state where RS1n and RS2n are ‘1’, FL1 and FL2 are ‘0’ and no write pulses are issued to either FIFO on W1n and W2n, nor does the controller supply any clock pulses on SOCP1 and SOCP2. IMDA is inactive high.

Upon START high the controller will keep RS1n low for one CLK2 clock cycle to reset FIFO 1. FL1, being the inverse of RS1n (see chapter 10) will then be high. After this reset cycle the controller will supply write pulses on W1n by combining the RASn and CLK2INT inputs according to:
W1n = RASn OR NOT CLK2INT

which in effect are pulses equal to CASn from the DRAM controller. During this activity no SOCP pulses are supplied to the FIFOs. What the FIFO controller is doing is simply allowing FIFO 1 to be filled with the data that is read from the DRAMs by the DRAM controller. This filling ends when an ENDOFSC high is detected. The controller will come in the same state as after reset.

Next, the FIFO controller waits for SOP to go low and subsequently SOS to go low. When it is synchronized to SOP low and SOS low the controller resets FIFO 2 with RS2n = '0' and FL2 = '1' for one CLK2 period. After this serial output clock pulses are supplied to FIFO 1 by inverting the BURST input:

SOCP1 = NOT BURST

Furthermore write pulses are supplied to FIFO 2:

W2n = RASn OR NOT CLK2INT

The FIFO controller is thus allowing FIFO 2 to be filled and FIFO 1 to be read by the printer engine.

When SOS goes high again (SOS is low for about 1 microsecond [PECSPEC]) the controller maintains its state but checks whether ENDOFPG becomes high or SOS becomes low again. Upon ENDOFPG = '1' the controller ends the filling of FIFO 2 but still enables reading from FIFO 1 until SOS becomes low. Then the controller returns to the initial state.

If ENDOFPG doesn’t become ‘1’ and SOS becomes ‘0’ reading from FIFO 1 is disabled and reading from FIFO 2 is enabled. FIFO 1 is reset again. After the one CLK2 period reset writing to FIFO 1 is enabled.

After SOS = ‘1’ the controller again checks for ENDOFPG. If ENDOFPG becomes high before SOS becomes low writing to FIFO 1 is disabled and reading FIFO 2 is enabled until SOS is detected to be low. Then the FIFO controller returns to the initial state.

When ENDOFPG remains to be low after SOS = ‘1’ and SOS is detected to be low the controller resets FIFO 2 and enables reading from FIFO 1. FIFO 2 can then be filled again.

It is clear that the FIFO controller, after having filled FIFO 1 due to a START command, in turns enables reading from FIFO 1 and filling FIFO 2, and then reading from FIFO 2 and filling FIFO 1. Upon ENDOFPG the controller returns to the initial state. During these actions the controller always is synchronized with the SOP and SOS signals.

### 13.7.1. State diagram and combinatorial signal assignments

The FIFO controller has been designed by combining a state machine and combinatorial assignments. The state machine is clocked with CLK2 falling and has the input vector (START, SOS, SOP, ENDOFSC, ENDOFPG). The outputs are combined in a 6 bits wide vector, with name outputs.

Outputs(0) and outputs(1) are used for enabling or disabling writing to FIFO 1 and 2 respectively. These enables are active-low. Outputs(2) and outputs(3) are for enabling or disabling reading of FIFO 1 and 2. They are active-high. Finally outputs(4) and outputs(5) are for the generation of the controller’s outputs RS1n, FL1 and RS2n, FL2 respectively. The combinatorial assignments to the outputs of the FIFO controller, based on the state machines vector outputs and the controller’s inputs RASn, BURST, and CLK2INT are:
W1n = RASn OR NOT CLK2INT OR outputs(0)
W2n = RASn OR NOT CLK2INT OR outputs(1)
SOCP1 = NOT BURST AND outputs(2)
SOCP2 = NOT BURST AND outputs(3)
RS1n = outputs(4)
RS2n = outputs(5)
FL1 = NOT outputs(4)
FL2 = NOT outputs(5)

Furthermore IMDA (see paragraph 12.7: design notes) is formed according to:

IMDA = SO1 AND SO2

The state machine diagram is given in figure 13.14. The interested reader can check the functionality, specified by this diagram, to the verbally described functionality in this paragraph.
The state diagram plus the combinatorial assignments are used to compose synthesizable VHDL, which can be found in the listing of the file FIFOctl.vhd (app. D.5). The VHDL model has been simu-
lated with Viewsim. Simulation results are in appendix E.4.

It can be seen that the controller issues the SOCP1 and 2 and W1n and W2n as desired. Also the functional timing of RS1n and RS2n and FL1 and 2 is right.

13.8. The architecture schematic

With the reader decoder, the address counter, scanlength counter, nrofscan counter, requester, DRAM controller and FIFO controller symbols a schematic has been created with Workview. This schematic can be found in appendix C.5. In order to have tristate outputs on the bitmap reader, tristate buffers from the Xilinx LCA 3000 series library have been added. To control these buffers the tristate controller that was used as an example in 13.3.4., has been added. Furthermore input and output buffers have been added to the input lines and remaining output lines. This is necessary when one wants to enter the implementation trajectory for Xilinx LCA’s, see chapter 14.

So now the bitmap reader is fully specified by it’s interfaces at the top level (figure 11.3), the decomposition into functional blocks, the interfaces of these functional blocks and either the VHDL specification of the blocks or the further schematic decomposition of the blocks into library macro’s. Since the reader is fully specified it can be simulated. Although the separate blocks have been verified on their functional behavior, the question remains if the reader as a whole behaves functionally correct, as showed in the flowchart of figure 11.2.

13.8.1. Verification by simulation

In order to do the verification a simulation has been carried out in which the reader is first reset and then programmed with start address $301_{16}$, scanlength $FF_{16}$ and number of scans $3_{16}$. The stimuli on the program interface of the reader are specified in a command file. Also the stimuli on all the other inputs of the reader are stated in this file.

The result of the simulation is given in a functional timing diagram in appendix E.5. This diagram comprises the total simulation. It has been verified that the reader indeed functions correctly by inspecting the waveforms not only from the total view in appendix E.5, but also by zooming and panning through the waveforms. The Viewlogic program Viewwave has been used for this verification.

A few remarks on this verification. It has been verified that the bitmap reader, after a start dummy write on the program interface makes a request and it resets FIFO 1. When the request has been granted the signals for the DRAM interface come out of tristate and the reader starts to perform DRAM accesses. Accordingly write pulses are supplied for FIFO 1. After exactly 256 accesses (in which it has been verified that the reader correctly handles overflow of the page mode page size) PAV is activated and the REQn is inactivated. When GRANTn is inactivated the outputs of the DRAM interface go into high impedance.

Until now BURST pulses, SOS and SOP were ignored. It has been verified that the bitmap reader after PAV correctly synchronizes on SOP and SOS. The bitmap reader correctly issues write pulses and serial output clock pulses (based on BURST) to the right FIFOs further on. IMDA is the correct combination of 501 and 502.

The bitmap reader activates the interrupt and inactivates PAV when three FIFOs of data, FIFO1, then FIFO 2 and then FIFO 1 again, have been passed on IMDA. At this point FIFO 2 contains data of a fourth scanline but this data will never be read since the reader ignores subsequent SOS pulses. After an interrupt acknowledge (dummy write on program interface) the interrupt line is inactivated again. A subsequent new start command causes the reader to start it’s actions all over again.

Based on the verification work it can be concluded that the bitmap reader specification is functionally correct. The next chapter will examine a possible implementation trajectory for the bitmap reader
based on logic synthesis for Xilinx LCA 3000 series technology and automatic placement and routing of the synthesized design into a real LCA.
14. Implementation trajectory

In this final chapter the experiences with a possible manner of implementation of the specified bitmap reader are treated. The parts that have been described in VHDL will first be translated into gates by logic synthesis. Then, after simulation of the bitmap reader at gate level it will be tried to place and route the design into a Xilinx LCA. Results and final recommendations will then be given.

14.1. Logic synthesis using the XI3000 library

With Viewlogic’s VHDLDES program the VHDL models have been translated into gate level designs. Since the implementation trajectory concentrates on the 3000 series LCA’s from Xilinx the library used by VHDLDES was the XI3000 library. After the synthesis schematic layouts of the resulting designs have been created with Viewlogic’s Viewgen. These schematics can be found in appendices F1 through F5. Inspection shows that in these schematics flipflops are generated for the outputs of the state machines that are described in the VHDL code and separate flipflops are generated for the states themselves. For instance the requester has 8 possible states that are encoded in three bits. The design shows three internal flipflops for these states.

As a result of the rules that have been followed for the VHDL descriptions, see paragraph 13.3, the synthesized logic is concise and without extra unnecessary flipflops. Other ways of composing the VHDL may lead to logic that does contain unnecessary flipflops [EVAL]. It is seen however that the designs contain unnecessary inverters. This is the result of the limited set of gate types the synthesizer uses. During further implementation steps the Viewlogic software can optimize the designs in this respect.

The synthesized designs have been incorporated in the hierarchy of the bitmap reader design. The VHDL blocks at the bitmap reader’s top level have been changed from modular to composite and the designs are added one level down from each block. The resulting design has been verified with Viewsim to be still correct. At this point the bitmap reader design is fully specified to the gate level and ready for porting to an LCA. Figure 14.1. shows the synthesis flow

![Synthesis flow diagram](Design with XI3000 macro library elements and functional VHDL blocks)

- Synthesize the VHDL models using XI3000 library
- Incorporate the synthesized logic into the hierarchy of the design

Figure 14.1: Synthesis flow
14.2. Porting to an XC3064-100 LCA

The Xilinx LCA's are programmable logic devices that consist of an array of socalled Configurable Logic Blocks (CLBs). At the moment Xilinx offers three families of LCAs, the 2000, the 3000 and the 4000 series. For devices of the 3000 series one CLB can be programmed with two combinational logic funtions with five inputs each. Furthermore one CLB contains two flipflops. Array sizes vary from 64 CLBs for the XC3020 (2000 gates) to 320 CLBs for the XC3090 (9000 gates). For every member of the family there are versions with 50, 70 and 100 MHz toggle rate [XIL]. From the Viewlogic software the total bitmap reader design has been estimated to be 189 CLBs large. Therefore an XC3064-100 LCA (224 CLBs, 6400 gates, toggle rate 100 MHz) will be used for the implementation study.

To place and route the design into the XC3064 the following path has been followed. See figure 14.2.

![Diagram](attachment:diagram.png)

After the execution of TOXNF the Viewlogic environment is left to enter the Xilinx environment for placement and routing. When the placement and routing has been successful the placed and routed LCA-file can be back annotated to the Viewlogic environment. To do this a wire file can be created from the LCA-file that can be used to create a simulation model for Viewsim. With viewsim the design can then be simulated again with the important notion that worst case timing delays, that resulted from placement and routing, can now be checked.

14.3. Result

To place and route the bitmap reader design about twenty times a try was made with APR. Most of the time the bitmap reader design could be successfully placed and routed into the XC3064 LCA. In these cases all CLBs could be placed and there were no unrouted nets. Problems arised however during the simulation sessions with the back annotated design. None of the placed and routed designs were able to meet the limits on timing. The designs showed longer worst case delays on some nets than allowed by the clock period, resulting in erroneous behavior of the state machines in the design. When the worst case delays were halved (a possibility of Viewsim) most of the designs functioned correctly.
In most cases the errors during simulation occurred due to ENDOFSC or ENDOFPG not changing in time. In other words, the delay for ENDOFSC and ENDOFPG that the counters supply to the other blocks of the reader was larger than the CLK2 period that clocks these counters and most of the blocks of the reader. It is believed that most of the problems arise from the fact that the bitmap reader incorporates such large counters. The counters occupy around half of the total of CLBs in the design. In the resulting placed and routed design the counters are not able to operate fast enough.

14.4. Recommendations

From 14.3 it is clear that straightforward implementation on an LCA, following the easiest, most automated flow, was not successful. However there are other ways of yielding a correct implementation which can be followed when one has the wish to use an LCA for the bitmap reader.

First of all it could be tried to follow the described implementation flow (fig 14.1 and 14.2) for the larger XC3090 LCA. Since this device has more CLBs it might be possible to automatically place the various blocks in a way that the routing yields shorter connections. Then the delays would be smaller. Furthermore the possibilities of using a 4000 series device could be investigated, provided the software belonging to the 4000 series is available.

Another possibility is to let the APR program create an implementation that meets the specifications as closely as possible and to then manually edit this implementation. This manual placement and routing can be done with Xilinx software and allows the designer to further optimize the implementation.

A third alternative is to not project all the functional blocks of the bitmap reader design on an LCA. Since the counters take up much resources of the LCA and cause the timing problems for ENDOFSC and ENDOFPG one could consider to leave them out of the LCA. The counters could be implemented with discrete components, for instance with TTL 74-series devices and the remaining part of the reader could be successfully placed and routed into an LCA.

Finally it should be realized that the method of specification of the reader, using VHDL for most parts, allows for lots of alternative implementation trajectories. For example one could decide to realize the different functional blocks with a mix of discrete components and programmable logic devices other than LCA’s. The specification still allows the designer this freedom.
15. Conclusions

From the evaluation of the TMS34020 it can be concluded that this processor is suitable for application in the Raster Image Processing unit. This is due to the strong graphics capabilities of the 34020. The processor's hardware supports easy DRAM design and the design of multiprocessing architectures. The graphics instructions allow the processor to be powerful in rasterization activities. General purposes characteristics complete the processor’s ability to be well used in the RIP unit design.

After a broad study of the requirements for the Raster Image Processing unit and the possible memory architectures the conclusion is that the system is best designed as a multiprocessing system. The processing entities are the TMS34020 and the bitmap reader, which share one contiguous memory. Scanline buffering enhances the system's performance.

The bitmap reader has been fully specified. The specification uses hierarchy and description of functional blocks with state machines, Boolean equations and VHDL models. The functionality has been verified by simulation to be correct. Logic synthesis, using Xilinx LCA technology completes the specification at gate level. From the described implementation trajectory for LCAs it can be concluded that straightforward implementation still meets some problems where timing is concerned. A more important conclusion however is that the functional specification is in fact technology independent and suitable for use with alternative ways of implementation on different PLD technologies.

The general conclusion is that the results of this final project form a solution to the stated problem. A top level Raster Image Processing unit architecture has been designed which is based on the 34020 and meets the functional requirements for the unit. To come to this architecture, the 34020, memory architectures and system bus designs have been evaluated and studied thoroughly. Specifically, the bitmap reader has been fully specified and the specification is suited for implementation with use of programmable logic. As a whole this report forms a framework for further implementation activities.
Literature


[PONS] Ponsor Corporation; "Portable TMS34020 Debugger user manual"; Ponsor Corp. 1990

[PECSPEC] R. Beeren; "FE-PE Video Interface Specification"; Internal report Océ

[KOO] Prof. Ir. J. Koomen; "College Systeemtechnologie"; Course Syllabus: Digital Systems section (EB), Faculty of Electrical Engineering, Eindhoven University of Technology


[LCADES] Viewlogic; "LCA design kit (Logic Cell Array)"; Viewlogic Systems Inc., Version A, June 1989


[EVAL] R. van Acquoij, E. Lambers, A. van Lier; "Evaluatie Viewlogic EDA systeem"; Internal report Océ

[XIL] Xilinx; "The programmable gate array data book"; Xilinx Inc. 1988

[HEUV] P. Heuvelings, G. Vons; "Parade; parallele rasterisatie, scanline interleaving"; Internal report Océ


[TOSH] Toshiba Corporation; "MOS Memory Data Book"; Toshiba Corp. 1990
[IDT] Integrated Device Technology Data Book


[VEEN1] Ir. C.A. van Veenendaal; “Grafische ontwerpen van toestandsmachines”; Elektronica Nr. 21, 1989, p. 49-55

[VEEN2] Ir. C.A. van Veenendaal; “Met toestandsmachines ontwerpen in PLD en ASIC”; Elektronica Nr. 9, 1990, p. 35-39


[MEY] E. Meyer; “VHDL opens the road to top-down design”; Computer Design, February 1989, p. 57-62


[ZENG] A.J. Zengerink; “In huis ontwikkelen van gate-arrays”; Electronica Nr. 8, 1990, p. 54-63

[ELEC] ELECTRONICA; “Themanummer: Programmeerbare Logica”; Electronica Nr. 17, 1990

Appendix A: Source code for the thick line benchmark

A1: C-sources

A2: Optimized Draw_trap function with inline assembly
#include <c:\usr\qpst\gsptools\stdlib.h>
#include <c:\usr\qpst\gsptools\float.h>
#include <c:\usr\qpst\gsptools\ctype.h>
#include <c:\usr\qpst\gsptools\errno.h>
#include <c:\usr\qpst\gsptools\time.h>

main()

    int Index, counter;

    for( counter = 0 ; counter <= 2000 ; counter++ ){
        for( Index = 0 ; Index <= 285 ; Index = Index + 19 ){
            DrawLine( 4, 4, 289, (Index + 4), 7 );
            DrawLine( 10, 114, (Index + 10), 399, 7 );
        }

        for( Index = 0 ; Index <= 600 ; Index = Index + 25 ){
            DrawLine( 330, 10, (Index + 330), 310, 3 );
            DrawLine( 930, 10, (930 - Index), 310, 3 );
        }

        for( Index = 0 ; Index <= 200 ; Index = Index + 20 ){
            DrawLine( 5, 425, 955, (Index + 425), 3 );
        }

        for( Index = 0 ; Index <= 950 ; Index = Index + 25 ){
            DrawLine( 5, 430, (Index + 5), 635, 3 );
        }

        /*
        * Only for test purposes,
        * Remove the next Procedure call for Benchmarking!
        */
        SaveBitMap();
    }
#include <math.h>
#include "line_con.h"

/* Global Variables */
unsigned long BitMap[ BM_HEIGHT * BM_WIDTH ];

DrawLine( xb, yb, xe, ye, width )
int xb, yb, xe, ye, width;
{
    int Ax, Ay, Bx, By, Cx, Cy, Dx, Dy, Fx, Fy;
    int dx, dy, swap;
    double length;
    if( xe<xb ) {
        swap = xe;
        xe = xb;
        xb = swap;
        swap = ye;
        ye = yb;
        yb = swap;
    }
    dx = xe - xb;
    dy = ye - yb;
    if( dx==0 ) {
        Draw_Vertline( xb, yb, ye, width );
    }
    else if( dy==0 ) {
        Draw_Horline( xb, xe, yb, width );
    }
    else {
        length = sqrt( (double)( (dx*dx)+(dy*dy) ) );
        Fx = (int)(((double)(width*dx)/(length+length))+0.5);
        if( dy>=0 ) {
            Fy = (int)(((double)(width*dy)/(length+length))+0.5);
        }
        else {
            Fy = (int)(((double)(width*dy)/(length+length))-0.5);
        }
        Ax = xb - Fy;
        Ay = yb + Fx;
        Bx = xb + Fy;
        By = yb - Fx;
        Cx = Bx + dx;
        Cy = By + dy;
        Dx = Ax + dx;
        Dy = Ay + dy;
        if( Ax==Bx ) {
            Draw_Trap( Ax, Dx, By, ey, Ay, Dy );
        }
        else if( dy>0 ) {
            if( Ax==Dx ) {
                Draw_Trap( Ax, Bx, Ay, By, Dy, ey );
            }
            else if( Dx>Bx ) {
                Do_Lower_Upper( Ax, Ay, Bx, By, Dx, Dy, Cx);
            }
            else{
                Do_Upper_Lower( Ax, Ay, Dx, Dy, Bx, By, Cx );
            }
        }
    }
}
Draw_Trap( Bx, Dx, Cy, Dy, By, Ay );

    }
else if( Cx<=Ax ) {
        Do_Lower_Upper( Bx, By, Cx, Cy, Ax, Ay, Dx);
    }
else {
        Do_Upper_Lower( Bx, By, Ax, Ay, Cx, Cy, Dx);
    }
}

lraw_Vertline( vx, vyb, vye, hwidth )
    int vx, vyb, vye, hwidth;

    int x, xleft, xright, swap;
    unsigned long *ScanlineAdress;

    xleft = vx-(hwidth/2);
    xright = vx+(hwidth/2);
    if( vyb>vye ) {
            swap=vyb;
            vyb=vye;
            vye=swap;
    }
    ScanlineAdress = BitMap + ( xleft * BM_HEIGHT ) ;
    x = xleft;
    while( x<=xright ) {
            RasterLine( ScanlineAdress, x, vyb, vye );
            ScanlineAdress = ScanlineAdress + BM_HEIGHT ;
            x++;}

lraw_Horline( hx, hxe, hy, vwidth )
    int hx, hxe, hy, vwidth;

    int x, yu, yl;
    unsigned long *ScanlineAdress;

    yu = hy+(vwidth/2);
    yl = hy-(vwidth/2);
    ScanlineAdress = BitMap + ( hx * BM_HEIGHT ) ;
    for( x=hx; x<=hxe; x++ ) {
            RasterLine( ScanlineAdress, x, yl, yu );
            ScanlineAdress = ScanlineAdress + BM_HEIGHT ;
    }
#include "line_con.h"

/*
 * x1,y1------/ 
 * \-----\x4,y4 */

/* Global Variables */
extern unsigned long BitMap[];

void Lower_Upper( x1, y1, x2, y2, x3, y3, x4 )
.

int x1, y1, x2, y2, x3, y3, x4;

int sdx, sdy, ddx, ddy;
int two_sdx, two_sdy, two_ddx, two_ddy;
int sdydx, sdydx, ddydx, ddydx;
int sdydx_plus1, sdydx_minus2sdx;
int ddydx_minus1, ddydx_plus2ddx;
int x, y1, yu, d1, d2;

unsigned long *ScanlineAdress = BitMap + ( x * BM_HEIGHT ) ;

sdx = x3 - x1;
sdy = y3 - y1;
ddx = x2 - x1;
ddy = y2 - y1;

two_sdx = sdx+sdx;
two_sdy = sdy+sdy;
two_ddx = ddx+ddx;
two_ddy = ddy+ddy;

sdydx = (int)(sdy/sdx);
sdydx = two_sdy - (two_sdx*sdyx);
ddydx = (int)(ddy/ddx);
ddydx = two_ddy - (two_ddx*ddydx);

sdyx_plus1 = sdydx + 1;
sdydx_minus2sdx = sdydx - two_sdx;
ddydx_minus1 = ddydx - 1;
ddydx_plus2ddx = ddydx + two_ddx;

x = x1; /* initialze x */
ScanlineAdress = BitMap + ( x * BM_HEIGHT ) ;

/* init on x1,y1 */
yu = ((two_sdx*y1)+((sdy-sdx)+(two_sdx-l)))/two_sdx;
/* upperline */
d1 = (two_sdy - (two_sdx*(((yu-y1)+sdyx)) + (sdy-sdx));
/* init on x1,y1 */
y1 = ((y1*two_ddx) + (ddx+ddy))/two_ddx;
/* lowerline */
d2 = (two_ddy - (two_ddx*(((y1-y1)+ddydx)) + (ddx+ddy));

while( x<x2 ) {
else{
    yu = yu + sdydxi;
    d1 = d1 + sdydx;
}
if( d2<0 ) {
    yl = yl + ddydxi_minus1;
    d2 = d2 + ddydf_plus2ddx;
} else{
    yl = yl + ddydxi;
    d2 = d2 + ddydx;
}
x++;
ScanlineAdress = ScanlineAdress + BM_HEIGHT ;
}
if( x==x2 ) {
    RasterLine( ScanlineAdress, x, y2, yu );
    if( dl>0 ){
        yu = yu + sdydxi_plus1;
        d1 = d1 + sdydxf_minus2sdx;
    } else{
        yu = yu + sdydxi;
        d1 = d1 + sdydx;
    }
    x++;
    ScanlineAdress = ScanlineAdress + BM_HEIGHT ;
}
/* init op onderste lijn */
yl = ((two_sdx*y2)+(sdy-sdx)+(two_sdx))/two_sdx;
d2 = ((two_sdy) - (two_sdx*((yl-y2)+sdydxi)) + (sdy-sdx);
while( x<x3 ) {
    RasterLine( ScanlineAdress, x, yl, yu );
    if( dl>0 ){
        yu = yu + sdydxi_plus1;
        d1 = d1 + sdydxf_minus2sdx;
    } else{
        yu = yu + sdydxi;
        d1 = d1 + sdydx;
    }
    if( d2>=0 ) {
        yl = yl + sdydxi_plus1;
        d2 = d2 + sdydf_minus2sdx;
    } else{
        yl = yl + sdydxi;
        d2 = d2 + sdydx;
    }
    x++;
    ScanlineAdress = ScanlineAdress + BM_HEIGHT ;
}
if( x==x3 ) {
    RasterLine( ScanlineAdress, x, yl, y3 );
    if( d2>=0 ) {
        yl = yl + sdydxi_plus1;
        d2 = d2 + sdydx_minus2sdx;
    } else{
        yl = yl + sdydxi;
        d2 = d2 + sdydx;
    }
}
\[ y = \frac{(\text{two}\_\text{ddx}\ast y3) + ((\text{two}\_\text{ddx}-1) + (\text{ddy} - \text{ddx}))}{\text{two}\_\text{ddx}}; \]
\[ d1 = \frac{(\text{two}\_\text{ddy}\ast 2) - (\text{two}\_\text{ddx}((y4 - y3) + \text{ddy} - \text{ddx})) + (\text{ddx} - \text{ddy})}{\text{two}\_\text{ddx}}; \]

```c
while( x<=x4 ) {
    RasterLine( ScanlineAdress, x, yl, yu );
    if( d1<=0 ) {
        yu = yu + ddy - ddx;
        d1 = d1 + ddy - ddx;
    } else{
        yu = yu + ddy;
        d1 = d1 + ddy;
    }
    if( d2>=0 ) {
        yl = yl + sdy;
        d2 = d2 + sdy;
    } else {
        yl = yl + sdy;
        d2 = d2 + sdy;
    }
    x++;    
    ScanlineAdress = ScanlineAdress + BM\_HEIGHT ;
}
```
#include "line_con.h"

/*
  \( x_1, y_1 \)------\---/  \( x_3, y_3 \) */
/*  / \    /----\      */
/*  \-----/ \-----/     */
/*  \( x_2, y_2 \) \-----/ */
/*  \( x_4, y_4 \) */

/* Global Variables */
extern unsigned long BitMap[] ;

o_Upper_Lower( x1, y1, x2, y2, x3, y3, x4 )
  int x1, y1, x2, y2, x3, y3, x4 ;
  int sdx, sdy, ddx, ddy ;
  int two_sdx, two_sdy, two_ddx, two_ddy ;
  int sdydxi, sdydx, ddydx, ddydx ;
  int sdydxi_plus1, sdydx_minus2sdx ;
  int ddydx_minus1, ddydx_plus2ddx ;
  int x, y1, yu, dl, d2 ;

  unsigned long *ScanlineAdress ;

  sdx = x2 - x1;
  sdy = y2 - y1;
  ddx = x3 - x1;
  ddy = y3 - y1;

  two_sdx = sdx+sdx;
  two_sdy = sdy+sdy;
  two_ddx = ddx+ddx;
  two_ddy = ddy+ddy;

  sdydxi = (int)(sdy/sdx);
  sdydx = two_sdy - (two_sdx*sdydxi);
  ddydx = (int)(ddy/ddx);
  ddydx = two_ddy - (two_ddx*ddydx);

  sdydxi_plus1 = sdydxi + 1;
  sdydx_minus2sdx = sdydx - two_sdx;
  ddydx_minus1 = ddydx - 1;
  ddydx_plus2ddx = ddydx + two_ddx;

  x = x1 ;  /*init x*/
  ScanlineAdress = BitMap + ( x * BM_HEIGHT ) ;

  /* init on x1,y1 */
  yu = ((two_sdx*y1)+((sdy-sdx)+(two_sdx-l)))/two_sdx;
  /* upperline */
  dl = (two_sdy - (two_sdx*((yu-yl)+sdydxi))) + (sdy-sdx);

  /* init on x1,y1 */
  yl = ((yl*two_ddx) + (ddx+ddy))/two_ddx;
  /* lowerline */
if (d2<0) {
    yl = yl + ddydxi_minus1;
    d2 = d2 + ddydxf_plus2ddx;
} else {
    yl = yl + ddydxi;
    d2 = d2 + ddydxf;
}
x++;
ScanlineAdress = ScanlineAdress + BM_HEIGHT;

if (x==x2) {
    RasterLine(ScanlineAdress, x, y3, y2);
    x++;
    ScanlineAdress = ScanlineAdress + BM_HEIGHT;
} else {
    if (x==x2) {
        RasterLine(ScanlineAdress, x, y1, y2);
        if (d1<=0) {
            yu = yu + ddydxi_minus1;
            d1 = d1 + ddydxf_plus2ddx;
        } else {
            yu = yu + ddydxi;
            d1 = d1 + ddydxf;
        }
    }
    if (d2<0) {
        yl = yl + ddydxi_minus1;
        d2 = d2 + ddydxf_plus2ddx;
    } else {
        yl = yl + ddydxi;
        d2 = d2 + ddydxf;
    }
    x++;
    ScanlineAdress = ScanlineAdress + BM_HEIGHT;
}

while (x<x3) {
    RasterLine(ScanlineAdress, x, y1, yu);
    if (d1<=0) {
        yu = yu + ddydxi_minus1;
        d1 = d1 + ddydxf_plus2ddx;
    } else {
        yu = yu + ddydxi;
        d1 = d1 + ddydxf;
    }
    if (d2<0) {
yl = yl + ddydx1;
d2 = d2 + ddydx2;
}
x++;
ScanlineAdress = ScanlineAdress + BM_HEIGHT ;
}

if( x==x3 ) {
  RasterLine( ScanlineAdress, x, y3, yu );
  if( d1<=0 ) {
    yu = yu + ddydx1_minus1;
    d1 = d1 + ddydxf_plus2ddx;
  } else{
    yu = yu + ddydx1;
    d1 = d1 + ddydxf;
  }
x++;
  ScanlineAdress = ScanlineAdress + BM_HEIGHT ;
}

yl = ((two_sdx*y3) + (sdy + sdx))/two_sdxi;
d2 = ((two_sdy*2) - (two_sdx*((yl-y3)+sdydxi))) - (sdx+sdy);

while (x<=x4) {
  RasterLine( ScanlineAdress, x, yl, yu );
  if( d1<=0 ) {
    yu = yu + ddydx1_minus1;
    d1 = d1 + ddydxf_plus2ddx;
  } else{
    yu = yu + ddydx1;
    d1 = d1 + ddydxf;
  }
  if( d2>=0 ) {
    yl = yl + sdydxi_plus1;
    d2 = d2 + sdydxf_minus2sdx;
  } else{
    yl = yl + sdydxi;
    d2 = d2 + sdydxf;
  }
x++;
  ScanlineAdress = ScanlineAdress + BM_HEIGHT ;
}


```c
#include "line_con.h"

/* Global Variables */
extern unsigned long Bitmap[];

raw_Trap( x1, x3, y2, y4, y1, y3 )
int x1, x3, y2, y4, y1, y3 ;
{
    int dx, dy, two_dx, two_dy, dydxi ;
    int dydx, dydx_plus1, dydx_minus1 ;
    int dydx_minus2dx, dydx_plus2dx ;
    int x, yu, yl, d1, d2 ;
    unsigned long *ScanlineAdress ;
    dx = x3 - x1 ;
    dy = y3 - y1 ;
    two_dx = dx + dx ;
    two_dy = dy + dy ;
    dydxi = (int)(dy / dx) ;
    dydx = two_dy - (two_dx * dydxi) ;
    if( dy >= 0 ) {
        dydx_plus1 = dydx + 1 ;
        dydx_minus2dx = dydx - two_dx ;
        x = x1 ;
        ScanlineAdress = Bitmap + ( x * BM_HEIGHT ) ;
        yu = (two_dx * y1) + ((dy - dx) + (two_dx - 1)) / two_dx ;
        d1 = (two_dy - (two_dx * ((yu - y1) + dydxi))) + (dy - dx) ;
        RasterLine( ScanlineAdress, x, y2, yu ) ;
        if( d1 > 0 ) {
            yu = yu + dydx_plus1 ;
            d1 = d1 + dydx_minus2dx ;
        } else {
            yu = yu + dydxi ;
            d1 = d1 + dydx ;
        }
        x++ ;
        ScanlineAdress = ScanlineAdress + BM_HEIGHT ;
        yl = (two_dx * y2) + (dy - dx) + (two_dx) / two_dx ;
        d2 = (two_dy - (two_dx * ((yl - y2) + dydxi))) + (dy - dx) ;
        while( x < x3 ) {
            RasterLine( ScanlineAdress, x, yl, yu ) ;
            if( d1 > 0 ) {
                yu = yu + dydx_plus1 ;
                d1 = d1 + dydx_minus2dx ;
            } else {
                yu = yu + dydxi ;
                d1 = d1 + dydx ;
            }
            if( d2 >= 0 ) {
                yu = yu + dydx_plus1 ;
                d2 = d2 + dydx_minus2dx ;
            } else {
                yu = yu + dydxi ;
                d2 = d2 + dydx ;
            }
        }
    } else {
        yu = yu + dydxi ;
        d1 = d1 + dydx ;
        if( d2 >= 0 ) {
            yu = yu + dydx_plus1 ;
            d2 = d2 + dydx_minus2dx ;
        } else {
            yu = yu + dydxi ;
            d2 = d2 + dydx ;
        }
    }
}
```
d2 = d2 + dydxf;
}
x++;  
ScanlineAdress = ScanlineAdress + BM_HEIGHT;
}  
RasterLine(ScanlineAdress, x, y1, y3);
}
else{

dydxi_minus1 = dydx - 1;
dydxf_plus2dx = dydxf + two_dx;

x = x1;
ScanlineAdress = BitMap + ( x * BM_HEIGHT );

y1 = (((y2 * two_dx) + (dx + dy)) / two_dx;
d2 = (((two_dy) - (two_dx * ((y1 - y2) + dydx))) + (dx + dy);

RasterLine(ScanlineAdress, x, y1, y1);
if ( d2 < 0 ) {
    y1 = y1 + dydx_minus1;
d2 = d2 + dydxf_plus2dx;
}
else{
    y1 = y1 + dydx;
d2 = d2 + dydxf;
}
x++;  
ScanlineAdress = ScanlineAdress + BM_HEIGHT;

yu = (((two_dx * y1) + ((two_dx - 1) + (dy - dx))) / two_dx;
d1 = (((two_dy) - (two_dx * ((yu - y1) + dydx))) + (dy + dx);

while ( x < x3 ) {
    RasterLine(ScanlineAdress, x, y1, yu);
    if ( d1 <= 0 ) {
        yu = yu + dydx_minus1;
d1 = d1 + dydxf_plus2dx;
    }
    else {
        yu = yu + dydx;
d1 = d1 + dydxf;
    }
    if ( d2 < 0 ) {
        y1 = y1 + dydx_minus1;
d2 = d2 + dydxf_plus2dx;
    }
    else {
        y1 = y1 + dydx;
d2 = d2 + dydxf;
    }
    x++;  
    ScanlineAdress = ScanlineAdress + BM_HEIGHT;
}
RasterLine(ScanlineAdress, x, y4, yu);

/*
 * Deze functie vult een segment in de scanline
 * op met een vulpatroon
 */

scanline_adr : adres eerste word in scanline.
* x : x-coordinaat van bitmap
* yl : y-coordinaat waar vullen moet beginnen.
* yu : y-coordinaat waar vullen eindigt.
*/

asterLine( scanline_adr, x, yl, yu )
unsigned long *scanline_adr ;

static unsigned long lowmask[32] = {
    0xFFFFFFPP, 0xFFFFFFPE, 0xFFFFFFFC, 0xFFFFFFF8,
    0xFFFFFFFO, 0xFFFFFFE0, 0xFFFFFFC0, 0xFFFFFFF0,
    0xFFFFFFEO, 0xFFFFFE00, 0xFFFFFC00, 0xFFFFFF00,
    0xFFFFFO00, 0xFFFFE000, 0xFFFFC000, 0xFFFFF000,
    0xFFFO000, 0xFFE0000, 0xFFC0000, 0xFF00000,
    0xF000000, 0xE000000, 0xC000000, 0x8000000 
};

static unsigned long highmask[32] = {
    0x00000001, 0x00000003, 0x00000007, 0x0000000F,
    0x0000001F, 0x0000003F, 0x0000007F, 0x000000FF,
    0x000001FF, 0x000003FF, 0x000007FF, 0x00000FFF,
    0x00001FFF, 0x00003FFF, 0x00007FFF, 0x0000FFFF,
    0x0001FFFF, 0x0003FFFF, 0x0007FFFF, 0x000FFFFF,
    0x001FFFFF, 0x003FFFFF, 0x007FFFFF, 0x00FFFFFF,
    0x01FFFFFF, 0x03FFFFFF, 0x07FFFFFF, 0x0FFFFFFF,
    0x1FFFFFFF, 0x3FFFFFFF, 0x7FFFFFFF, 0xFFFFFFF 
};

/*
 * yl en yu zijn pixel nummers.
 * first_word en last_word zijn de word nummers
 * waarin deze pixel(s) liggen.
 */
int first_word, last_word;

/*
 * index is offset tov scanline_adr
 */
unsigned long *index;

/*
 * pattern = ZWART !!!!!
 * color is mogelijk hiermee
 */
unsigned long pattern;

unsigned long mask, old;
int count;

pattern = BLACK_VALUE;

/* met 32 bits per word is het word number;...*/
last_word = yu >> 5;

index = scanline_adr + first_word;

if ( last_word == first_word )
{
    /* patroon in een woord */
    mask = highmask[ yu & REM32 ] & lowmask[ yl & REM32 ];
    old = *index & (~mask);
    *index = (pattern & mask) | old;
}

if ( last_word == (first_word + 1) )
{
    /* patroon in twee woorden */
    /* first word */
    mask = lowmask[ yl & REM32 ];
    old = *index & (~mask);
    *index = (pattern & mask) | old;

    /* second word */
    mask = highmask[ yu & REM32 ];
    old = *(index + 1) & (~mask);
    *(index + 1) = (pattern & mask) | old;
}

if ( last_word == (first_word + 2) )
{
    /* patroon in drie woorden */
    /* first word */
    mask = lowmask[ yl & REM32 ];
    old = *index & (~mask);
    *index = (pattern & mask) | old;

    /* third word */
    mask = highmask[ yu & REM32 ];
    old = *(index + 2) & (~mask);
    *(index + 2) = (pattern & mask) | old;

    /* second word */
    *(index + 1) = pattern;
}

if ( last_word > (first_word + 2) )
{
    /* patroon in meerdere woorden */

    /* first word */
    mask = lowmask[ yl & REM32 ];
    old = *index & (~mask);
    *index = (pattern & mask) | old;

    /* last word */
    mask = highmask[ yu & REM32 ];
    old = *(index + (last_word-first_word) ) & (~mask);
    *(index + (last_word-first_word) ) = (pattern & mask) | old;

    /* 'tussen' word */

    for ( count = 1; count < (last_word-first_word); count++ )
    {
        *(index + count) = pattern;
    }
}
A.2 Optimized Draw_trap function with inline assembly
#include "line_con.h"

/* Global Variables */
extern unsigned long BitMap[];

Draw_Track( x1, x3, y2, y4, y1, y3 )
int x1, x3, y2, y4, y1, y3 ;
{
  asm(" .copy c:\usr\qpst\takehome\sdb20.equ ");
  asm(" mmtm SP,B0,B1,B2,B7,B10,B11,B12,B13,B14,B3,B4,B9 ");
  asm(" move FP,B14 ");
  asm(" movi 0280h,DPTCH,1 ");
  asm(" setcdp ");
  asm(" mwait ");
  asm(" movi 0001h,A0,1 ");
  asm(" move AO,@OC0000150h,1 ");
  asm(" movi 0000000h,OFFSET,1 ");
  asm(" movi 0000000h,COLO1,1 ");
  asm(" move *B14(-96),SADDR,1 ");
  asm(" sll 16,SADDR ");
  asm(" movi 0000000h,COLOR1,1 ");
  asm(" move *B14(-160),DYDX,1 ");
  /* asm(" inc DYDX "); */
  asm(" sll 16,DYDX ");
  asm(" move *B14(-32),MPTCH,1 ");
  asm(" move *B14(-128),SPTCH,1 ");
  asm(" sll 16,SPTCH ");
  asm(" move *B14(-192),B13,1 ");
  /* asm(" inc B13 "); */
  asm(" sll 16,B13 ");
  asm(" move *B14(-64),B12,1 ");
  /* asm(" inc B12 "); */
  asm(" sub SADDR,SPTCH ");
  asm(" sub DYDX,B13 ");
  asm(" sub MPTCH,B12 ");
  asm(" divs B12,SPTCH ");
  asm(" divs B12,B13 ");
  asm(" move B13,MADDR ");
  asm(" sll 16,MPTCH ");
  asm(" loop: ");
  asm(" tfill XY ");
  asm(" dsjs B12,loop ");
  asm(" mmfm SP,A0 ");
  asm(" mmfm SP,B0,B1,B2,B7,B10,B11,B12,B13,B14,B3,B4,B9 ");
}
Appendices

Appendix B: Bitmap reader flow chart

Flow chart for global behavior of the bitmap reader
Appendix C: Schematics

C.1: Top level RIP unit architecture
C.2: 32 bits address counter
C.3: 8 bits scanlength down counter
C.4: 16 bits number of scans down counter
C.5: Decomposed bitmap reader architecture
Appendix C.3

Design: 8 bits Scanlength counter

date: April 1991

DRAWN BY: PSE Steemers
Appendix D: Synthesizable VHDL descriptions

D.1: Tristate controller (tstctl.vhd)
D.2: Bitmap reader decoder (readdecode.vhd)
D.3: Requester (requester.vhd)
D.4: DRAM controller (rascas.vhd)
D.5: FIFO controller (fifoclt.vhd)
-- This is a VHDL file that describes the
-- tristate controller of the bitmap reader.

-- design name: tstctl.vhd
-- designer: P.S.E. Steemers
-- date: march 14th 1991
-- update: april 26th 1991

-- Interface declaration:

ENTITY tstctl IS
    PORT(
        RST, GRANT: IN vlbit;
        CLK1: IN vlbit;
        TST: OUT vlbit;
        GI: OUT vlbit);
END tstctl;

ARCHITECTURE behavior OF tstctl IS

TYPE state_vec IS array (0 to 1) OF vlbit;

SIGNAL cur_state: state_vec := ('0', '0');

BEGIN

control:PROCESS

VARIABLE next_state: state_vec;
CONSTANT SO: state_vec := ('0', '0');
CONSTANT S1: state_vec := ('0', '1');
CONSTANT S2: state_vec := ('1', '0');
CONSTANT unknown: state_vec := ('X', 'X');

BEGIN

WAIT UNTIL CLK1 = '1';

-- implement reset logic

IF rst = '1' then
    next_state := SO; tst <= '1'; GI <= '1';
ELSIF bitx(rst) then
    next_state := unknown; tst <= 'X'; GI <= 'X';

-- state encoding and output derivation

ELSIF cur_state = SO THEN
    IF GRANT = '0' THEN
        next_state := S1; tst <= '0'; GI <= '0';
    ELSE
        next_state := SO; tst <= '1'; GI <= '1';
    END IF;
ELSIF cur_state = S1 THEN
    IF GRANT = '1' THEN
        next_state := S2; tst <= '0'; GI <= '1';
    ELSE
        next_state := S1; tst <= '0'; GI <= '0';
    END IF;
ELSIF cur_state = S2 THEN
    next_state := SO; tst <= '1'; GI <= '1';
ELSE
    next_state := unknown; tst <= 'X'; GI <= 'X';

END control;
-- This is a VHDL file that describes the
-- bitmapreader's decoder.

-- Interface declaration:

ENTITY readdecode IS
  PORT(
    RCS, STROBE, RDWR: IN vlbit;
    SELECT: IN vlbit_vector(3 downto 0);
    SAS0,SAS1,SAS2,SAS3: OUT vlbit;
    SLS,NOS0,NOS1,START,CTL,INTACK: OUT vlbit);
END readdecode;

ARCHITECTURE behavior OF readdecode IS

BEGIN
AS0 <= not RCS and not STROBE and not RDWR and not SELECT(3) and not SELECT(2) and not SELECT(1) and not SELECT(0);
AS1 <= not RCS and not STROBE and not RDWR and not SELECT(3) and not SELECT(2) and not SELECT(1) and SELECT(0);
AS2 <= not RCS and not STROBE and not RDWR and not SELECT(3) and not SELECT(2) and SELECT(1) and not SELECT(0);
AS3 <= not RCS and not STROBE and not RDWR and not SELECT(3) and not SELECT(2) and SELECT(1) and SELECT(0);
SLS <= not RCS and not STROBE and not RDWR and not SELECT(3) and SELECT(2) and not SELECT(1) and not SELECT(0);
NOS0 <= not RCS and not STROBE and not RDWR and not SELECT(3) and SELECT(2) and not SELECT(1) and SELECT(0);
NOS1 <= not RCS and not STROBE and not RDWR and not SELECT(3) and SELECT(2) and SELECT(1) and not SELECT(0);
TL <= not RCS and not STROBE and not RDWR and not SELECT(3) and SELECT(2) and SELECT(1) and SELECT(0);
START <= not RCS and SELECT(3) and not SELECT(2) and not SELECT(1) and not SELECT(0); -- S
INTACK <= not RCS and SELECT(3) and not SELECT(2) and not SELECT(1) and SELECT(0);
END behavior;
-- This is a VHDL file that describes the requester

design name: requester.vhd
designer: P.S.E. Steemers
date: february 21th 1991
update: april 26th 1991

-- Interface declaration:
ENTITY requester IS
  PORT(
    RST, CLK2, START, ENDOFSC, ENDOFFG, SOP, SOS, INTACK: IN vlbit;
    REQ, SLPE, NOSPE, PAV, INT: OUT vlbit);
END requester;

-- Description of behavior
ARCHITECTURE behavior OF requester IS
  TYPE state_vec IS array (0 to 2) OF vlbit;
  SIGNAL outputs: vlbit_vector(0 to 4);
  SIGNAL cur_state: state_vec := ('0', '0', '0');
BEGIN
  REQ <= outputs(0);
  SLPE <= outputs(1);
  NOSPE <= outputs(2);
  PAV <= outputs(3);
  INT <= outputs(4);

  request: PROCESS
  VARIABLE next_state: state_vec;
  CONSTANT S0: state_vec := ('0', '0', '0');
  CONSTANT S1: state_vec := ('0', '0', '1');
  CONSTANT S2: state_vec := ('0', '1', '0');
  CONSTANT S3: state_vec := ('0', '1', '1');
  CONSTANT S4: state_vec := ('1', '0', '0');
  CONSTANT S5: state_vec := ('1', '0', '1');
  CONSTANT S6: state_vec := ('1', '1', '0');
  CONSTANT S7: state_vec := ('1', '1', '1');
  CONSTANT unknown: state_vec := ('X', 'X', 'X');

  CONSTANT 00: vlbit_vector(0 to 4) := ('0', '0', '0', '0', '1');
  CONSTANT 01: vlbit_vector(0 to 4) := ('1', '1', '1', '0', '1');
  CONSTANT 02: vlbit_vector(0 to 4) := ('1', '0', '0', '1', '1');
  CONSTANT 03: vlbit_vector(0 to 4) := ('0', '0', '0', '0', '0');
  CONSTANT 04: vlbit_vector(0 to 4) := ('X', 'X', 'X', 'X', 'X');

  BEGIN
    WAIT UNTIL CLK2 = '0';

    -- implement reset logic
    IF rst = '1' then
      next_state := S0; outputs := 01;
  END request;
END behavior;
ELSIF bitx(rst) then
  next_state := unknown; outputs <= 06;
-- state encoding and output derivation

ELSIF cur_state = S0 THEN
  IF START = '1' THEN
    next_state := S1; outputs <= 00;
  ELSE
    next_state := S0; outputs <= 01;
  END IF;
ELSIF cur_state = S1 THEN
  IF ENDOFSC = '1' THEN
    next_state := S2; outputs <= 02;
  ELSE
    next_state := S1; outputs <= 00;
  END IF;
ELSIF cur_state = S2 THEN
  IF SOP = '1' THEN
    next_state := S3; outputs <= 02;
  ELSE
    next_state := S2; outputs <= 02;
  END IF;
ELSIF cur_state = S3 THEN
  IF SOP = '0' THEN
    next_state := S4; outputs <= 02;
  ELSE
    next_state := S3; outputs <= 02;
  END IF;
ELSIF cur_state = S4 THEN
  IF SOS = '1' THEN
    next_state := S5; outputs <= 02;
  ELSE
    next_state := S4; outputs <= 02;
  END IF;
ELSIF cur_state = S5 THEN
  IF SOS = '0' THEN
    next_state := S6; outputs <= 03;
  ELSE
    next_state := S5; outputs <= 02;
  END IF;
ELSIF cur_state = S6 THEN
  IF ENDOFPG = '1' THEN
    next_state := S7; outputs <= 05;
  ELSIF ENDOFSC = '1' THEN
    next_state := S5; outputs <= 02;
  ELSE
    next_state := S6; outputs <= 04;
  END IF;
ELSIF cur_state = S7 THEN
  IF INTACK = '1' THEN
    next_state := S0; outputs <= 01;
  ELSE
    next_state := S7; outputs <= 05;
  END IF;
ELSE
  next_state := unknown; outputs <= 06;
END IF;

END PROCESS request;

END behavior;
-- This is a VHDL file that describes the DRAM controller
-- of the bitmap reader.

-- design name: rascas.vhd
-- designer: P.S.E. Steemers
-- date: february 19th 1991
-- update: march 29th 1991

-- Interface declaration:

ENTITY rascas IS
    PORT(
        rst: in vlbit; --synch reset
        Interface to control
        ADRIN: IN vlbit_vector(26 downto 0);
        ENDOFSC: IN vlbit;
        CLEAR: IN vlbit;
        Interface to clocks
        CLK1, CLK2, CLK1INT, CLK2INT: IN vlbit;
        Interface to memory
        RCA: OUT vlbit_vector(10 downto 0);
        RAS: OUT vlbit;
        CAS, WE: OUT vlbit;
        ALTCH, DDIN, DDOUT: OUT vlbit; -- In this design ALTCH will be the sa
        Interface to arbiter
        GRANT: IN vlbit;
        Interface to decoder
        A: OUT vlbit_vector(4 downto 0));
END rascas;

-- Description of behavior

ARCHITECTURE behavior OF rascas IS

TYPE state_vec IS array (0 to 1) OF vlbit;

IGNAL rasint: vlbit := '1';
IGNAL cur_state: state_vec := ('0', '0');
BEGIN
    AS <= rasint;
    AS <= rasint OR NOT CLK2INT;
    WE <= CLEAR OR rasint OR CLK1INT OR NOT CLK2INT;
    DIN <= '0'; -- ddin is always inactive
    DOUT <= '1'; -- ddout is always inactive
    LTCH <= rasint;

    addrmux: PROCESS
      BEGIN
        WAIT UNTIL CLK1 = '1';
        IF rasint = '1' THEN
          RCA <= ADRIN(21 downto 11);
          A <= ADRIN(26 downto 22);
          ELSIF rasint = '0' THEN
            RCA <= ADRIN(10 downto 0);
          END IF;
      END PROCESS addrmux;

    rasgen: PROCESS
      BEGIN

VARIABLE next_state: state_vec;
CONSTANT page: vlbit_vector(7 downto 0) := ('1','1','1','1','1','1','1','1');
CONSTANT S0: state_vec := ('0','0');
CONSTANT S1: state_vec := ('0','1');
CONSTANT S2: state_vec := ('1','0');
CONSTANT unknown: state_vec := ('X','X');  -- unknown state

BEGIN
  WAIT UNTIL CLK2 = '0';

  -- implement reset logic
  IF rst = '1' then
    next_state := S0; rasint <= '1';
  ELSIF bitx(rst) then
    next_state := unknown; rasint <= 'X';
  END IF;

  -- implement next state encoding and output derivation
  ELSIF cur_state = S0 THEN
    IF GRANT = '0' THEN
      next_state := S1; rasint <= '0';
    ELSE
      next_state := S0; rasint <= '1';
    END IF;
  ELSIF cur_state = S1 THEN
    IF ENDOFSC = '1' THEN
      next_state := S2; rasint <= '1';
    ELSIF GRANT = '1' THEN
      next_state := S0; rasint <= '1';
    ELSE
      next_state := S1; rasint <= '0';
    END IF;
  ELSIF cur_state = S2 THEN
    IF GRANT = '1' THEN
      next_state := S0; rasint <= '1';
    ELSE
      next_state := S2; rasint <= '1';
    END IF;
  ELSE
    next_state := unknown; rasint <= 'X';
  END IF;
  cur_state <= next_state;
END PROCESS rasgen;
END behavior;
This is a VHDL file that describes the FIFO controller of the bitmap reader.

-- design name: fifoetl.vhd
-- designer: P.S.E. Steemers
-- date: march 7th 1991
-- update: april 26th 1991

-- Top level interface declaration:

ENTITY fifoetl IS
PORT (
  RST, CLK2,CLK2INT, START,ENDOFSC,ENDOFPG,SOP,SOS,RAS,BURST,SO1,SO2: IN vlb bit;
  RS1,RS2,FL1,FL2,W1,W2,SOCP1,SOCP2,IMDA: OUT vlbit);
END fifoetl;

-- Description of behavior

ARCHITECTURE behavior OF fifoetl IS

TYPE state_vec IS array (0 to 3) OF vlbit;

SIGNAL outputs: vlbit_vector(O to 5);
SIGNAL cur_state: state_vec := ('0','0','0','0');

BEGIN

control: PROCESS

VARIABLE next_state: state_vec;

CONSTANT S0: state_vec := ('0','0','0','0');
CONSTANT S1: state_vec := ('0','0','0','1');
CONSTANT S2: state_vec := ('0','1','0','0');
CONSTANT S3: state_vec := ('0','1','0','1');
CONSTANT S4: state_vec := ('0','1','1','0');
CONSTANT S5: state_vec := ('0','1','1','1');
CONSTANT S6: state_vec := ('1','0','0','0');
CONSTANT S7: state_vec := ('1','0','0','1');
CONSTANT S8: state_vec := ('1','0','1','0');
CONSTANT S9: state_vec := ('1','0','1','1');
CONSTANT S10: state_vec := ('1','1','0','0');
CONSTANT S11: state_vec := ('1','1','0','1');
CONSTANT unknown: state_vec := ('X','X','X','X');

CONSTANT 01: vlbit_vector(0 to 5) := ('1','1','0','0','1','1');
CONSTANT 02: vlbit_vector(0 to 5) := ('1','1','0','0','0','1');
CONSTANT 03: vlbit_vector(0 to 5) := ('1','1','1','0','1','1');
CONSTANT 04: vlbit_vector(0 to 5) := ('1','1','1','0','0','1');
CONSTANT 05: vlbit_vector(0 to 5) := ('1','0','0','0','1','1');
CONSTANT 06: vlbit_vector(0 to 5) := ('1','0','0','1','1','1');
CONSTANT 07: vlbit_vector(0 to 5) := ('0','0','0','1','1','1');
CONSTANT 08: vlbit_vector(0 to 5) := ('1','1','0','0','0','1');
CONSTANT 09: vlbit_vector(0 to 5) := ('0','0','0','0','0','1');
CONSTANT 10: vlbit_vector(0 to 5) := ('X','X','X','X','X','X');

BEGIN
  WAIT UNTIL CLK2 = '0';

-- implement reset logic
IF \( rst = '1' \) then
next_state := S0; outputs <= 01;
ELSIF \( \text{bitx}(rst) \) then
next_state := unknown; outputs <= 010;

-- implement state encoding

ELSIF \( \text{cur_state} = S0 \) THEN
  IF \( \text{START} = '1' \) THEN
    next_state := S1; outputs <= 02;
  ELSE
    next_state := S0; outputs <= 01;
  END IF;

ELSIF \( \text{cur_state} = S1 \) THEN
  IF \( \text{ENDOFSC} = '1' \) THEN
    next_state := S2; outputs <= 01;
  ELSE
    next_state := S1; outputs <= 09;
  END IF;

ELSIF \( \text{cur_state} = S2 \) THEN
  IF \( \text{SOP} = '1' \) THEN
    next_state := S3; outputs <= 01;
  ELSE
    next_state := S2; outputs <= 01;
  END IF;

ELSIF \( \text{cur_state} = S3 \) THEN
  IF \( \text{SOP} = '0' \) THEN
    next_state := S4; outputs <= 01;
  ELSE
    next_state := S3; outputs <= 01;
  END IF;

ELSIF \( \text{cur_state} = S4 \) THEN
  IF \( \text{SOS} = '1' \) THEN
    next_state := S5; outputs <= 01;
  ELSE
    next_state := S4; outputs <= 01;
  END IF;

ELSIF \( \text{cur_state} = S5 \) THEN
  IF \( \text{SOS} = '0' \) THEN
    next_state := S6; outputs <= 03;
  ELSE
    next_state := S5; outputs <= 01;
  END IF;

ELSIF \( \text{cur_state} = S6 \) THEN
  IF \( \text{SOS} = '1' \) THEN
    next_state := S7; outputs <= 05;
  ELSE
    next_state := S6; outputs <= 05;
  END IF;

ELSIF \( \text{cur_state} = S7 \) THEN
  IF \( \text{ENDOFPG} = '1' \) THEN
    next_state := S11; outputs <= 04;
  ELSIF \( \text{SOS} = '0' \) THEN
    next_state := S8; outputs <= 08;
  ELSE
    next_state := S7; outputs <= 05;
  END IF;

ELSIF \( \text{cur_state} = S8 \) THEN
  IF \( \text{SOS} = '1' \) THEN
    next_state := S9; outputs <= 07;
  ELSE
    next_state := S8; outputs <= 07;
  END IF;

ELSIF \( \text{cur_state} = S9 \) THEN
  IF \( \text{ENDOFPG} = '1' \) THEN
    next_state := S10; outputs <= 06;
ELSIF S0S = '0' THEN
next_state := S6; outputs <= O3;
ELSE
next_state := S9; outputs <= O7;
END IF;
ELSIF cur_state = S10 THEN
IF S0S = '0' THEN
next_state := S0; outputs <= O1;
ELSE
next_state := S10; outputs <= O6;
END IF;
ELSIF cur_state = S11 THEN
IF S0S = '0' THEN
next_state := S0; outputs <= O1;
ELSE
next_state := S11; outputs <= O4;
END IF;
ELSE
next_state := unknown; outputs <= O10;
END IF;

cur_state <= next_state;
END PROCESS control;

W1 <= RAS or not CLK2INT or outputs(0);
W2 <= RAS or not CLK2INT or outputs(1);
SOCP1 <= not BURST and outputs(2);
SOCP2 <= not BURST and outputs(3);
RS1 <= outputs(4);
RS2 <= outputs(5);
FL1 <= not outputs(4);
FL2 <= not outputs(5);
IMDA <= sol and s02;

END behavior;
Appendix E: Simulation results

| E.1: | Reader decoder |
| E.2: | Requester |
| E.3: | DRAM controller (two sheets) |
| E.4: | FIFO controller |
| E.5: | Bitmap reader |
Appendix E.1: Simulation of the reader decoder

- RCS
- STROBE
- RDWR
- SELECT
- SAS0
- SAS1
- SAS2
- SAS3
- SLS
- NOS0
- NOS1
- CTL
- START

<table>
<thead>
<tr>
<th>Time (ns)</th>
<th>0</th>
<th>118</th>
<th>194</th>
<th>2166</th>
<th>2390</th>
<th>2614</th>
<th>2838</th>
<th>3062</th>
<th>3286</th>
<th>3510</th>
<th>3734</th>
<th>3958</th>
<th>4182</th>
<th>4406</th>
<th>4630</th>
</tr>
</thead>
</table>
Appendix E.2: Simulation of the requester

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RST</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>START</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SOP</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SOS</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ENDOFSC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ENDOFPG</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>INTACK</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>REQ</td>
<td>X</td>
<td>2560 7680 13312 19456 25600 31744 37888 44032 50176 56320 62464 68608 74752</td>
</tr>
<tr>
<td>SLPE</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>NOSPE</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>PAV</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>INT</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>
Appendix E.4: Simulation result of the FIFO controller
Appendix F: Resulting schematics after logic synthesis

F.1: Reader decoder schematic
F.2: Requester schematic
F.3: DRAM controller schematic
F.4: FIFO controller schematic
F.5: Tristate controller schematic