Master's thesis:

TRANSPUTER
DESIGN AND SIMULATION
IN IDASS

Ing. M.C.J.M. van Hassel

Coach : Dr. ir. A.C. Verschueren
Supervisor : Prof. ir. M.P.J. Stevens
Period : September 1993 - August 1994
**ABSTRACT**

This master thesis is the conclusion of my graduation period at Eindhoven University of Technology. This report describes the design and implementation in IDaSS of a transputer, based on the IMS T800 from INMOS.

Transputers are microprocessors that can be connected directly with each other via on-chip link interfaces. In this way a network of transputers can easily be build. This enables parallel computing. A transputer also contains an internal process scheduler. This enables parallel processing.

IDaSS is an interactive design and simulation environment for digital circuits. It is developed by my coach Dr. Ir. A.C. Verschueren.

In IDaSS a transputer is implemented with four link interfaces, two timers, an event channel and a process scheduler. The floating-point unit and the dynamic RAM interface are not implemented because of time constraints. But these blocks already exist in the IDaSS design library. The goal was to implement a transputer core, based on the INMOS T800. Most typical transputer features are implemented. So the goal has been reached to a large extent.

Furthermore during this graduation period I have written an AWK tool for stripping IDaSS documentation files from not useful text. Because my coach and other IDaSS users were very interested, I adjusted this tool to the recommendations of my coach and spread it. The usage and working of this tool is briefly described in this report. This tool can only strip documentation files from IDaSS version V0.08m. The next IDaSS version will make this tool superfluous.
PREFACE

This report is the master thesis for my graduation at the Eindhoven University of Technology. It concludes the three years I have spent at this university to acquire the title 'Ir' in the field of Information Technology in continuation of the highschool of technology in Breda.

In order to reduce the size of appendices, I used my AWK tool to shrink the loads of documentation generated by IDaSS.

If anyone would like to obtain the original IDaSS design files and/or the AWK tool, then contact Dr. Ir. A.C. Verschueren at the Digital Systems Group (EB). He can also be reached via email at: verschue@eb.ele.tue.nl.

Here I would like to take the opportunity to thank some people who made my graduation possible and joyful:

• Ad Verschueren for the very good coaching.
• All students and co-workers at the Digital Systems Group for helping me anyhow and for making my graduation period enjoyable.
• Mark Venbrux for sharing his house with me for two very joyful years.
• The Walhalla and the AOR for cheap beer.
• My parents and my girlfriend Lianda de Jong for pushing me to carry on and for giving me the opportunity to study.
• Minister Ritzen for waiting to restrict the study freedom and for waiting to economize on the scholarship and free travelling after I finished studying.
• Everybody else who contributed in any way to the success of my study. Thanks!

Marco van Hassel
# TABLE OF CONTENTS

1. INTRODUCTION ................................................................. i
   1.1 History ..................................................................... 1
   1.2 Multiprocessor systems ............................................. 2
   1.3 Applications .............................................................. 4
   1.4 Comparison with other multiprocessor systems ............. 5
       1.4.1 TMS320C40 ............................................... 5
       1.4.2 iWarp ....................................................... 6
       1.4.3 POOL and DOOM .......................................... 6

2. OCCAM ........................................................................... 7

3. IMS T800 ........................................................................ 9
   3.1 System services ......................................................... 10
   3.2 External memory interface ......................................... 11
   3.3 Internal RAM ............................................................ 11
   3.4 CPU ........................................................................ 12
       3.4.1 Registers ....................................................... 12
       3.4.2 Instructions ..................................................... 13
       3.4.3 Scheduler ......................................................... 13
       3.4.4 Communication through channels .................... 15
   3.5 Floating point unit ..................................................... 17
   3.6 Links ..................................................................... 17
   3.7 Events .................................................................... 19
   3.8 Timers .................................................................... 19

4. IDASS ............................................................................. 23
   4.1 Converting an IDaSS design to silicon ....................... 24

5. IMPLEMENTATION OF THE TRANSPUTER ....................... 25
   5.1 Interfacing with the outside world ............................ 26
   5.2 Internal memory interfacing ..................................... 27
   5.3 The Z-bus protocol .................................................. 28
   5.4 Timing of the links and timers ................................. 29

6. TEST SYSTEM .................................................................. 31

7. IMPLEMENTATION OF RAM AND ROM ......................... 33
   7.1 Internal and external RAM ........................................ 33
   7.2 ROM ................................................................. 34

8. IMPLEMENTATION OF THE BUS MULTIPLEXER ............ 35
   8.1 Selection ............................................................... 36
   8.2 Arbitration ............................................................. 37
   8.3 Multiplex unit .......................................................... 38

9. IMPLEMENTATION OF SYSTEM SERVICES ..................... 41
10. IMPLEMENTATION OF THE EXTERNAL MEMORY INTERFACE .............................................. 45

11. IMPLEMENTATION OF THE TIMERS ..................................................................................... 47
   11.1 Timer 0 ......................................................................................................................... 47
   11.2 Timer 1 ......................................................................................................................... 48

12. IMPLEMENTATION OF THE LINKS .................................................................................... 51
   12.1 The bus multiplexer ....................................................................................................... 52
   12.2 Link control .................................................................................................................. 53
   12.3 Link interface ................................................................................................................ 53
      12.3.1 The bus multiplexer ............................................................................................... 54
      12.3.2 Link output ............................................................................................................. 56
      12.3.3 Link input ............................................................................................................... 59

13. IMPLEMENTATION OF THE EVENT CHANNEL ................................................................. 63

14. IMPLEMENTATION OF THE CPU ....................................................................................... 65
   14.1 The fetcher .................................................................................................................... 65
   14.2 The decoder .................................................................................................................. 67
   14.3 The load/store unit ....................................................................................................... 68
   14.4 The workspace cache ................................................................................................... 70
   14.5 The Z-bus arbiter ......................................................................................................... 71
   14.6 The scheduler ............................................................................................................... 71
      14.6.1 Arbitration of the requests ....................................................................................... 73
      14.6.2 State control .......................................................................................................... 74
      14.6.3 Scheduling a process .............................................................................................. 76
      14.6.4 Extracting a process ............................................................................................... 77
   14.7 The executer .................................................................................................................. 78
      14.7.1 State number and instruction control ..................................................................... 81
      14.7.2 Stack manipulations ............................................................................................... 82
      14.7.3 Loading and storing ............................................................................................... 84
      14.7.4 Z-bus communications .......................................................................................... 84

15. EXECUTION OF INSTRUCTIONS ....................................................................................... 87

16. PERFORMANCE .................................................................................................................... 97

17. STRIPPING IDASS DOCUMENT FILES .......................................................................... 105
   17.1 Usage of the stripper ..................................................................................................... 105
   17.2 Explanation of the AWK program ................................................................................. 106

18. CONCLUSIONS .................................................................................................................. 107

19. RECOMMENDATIONS ........................................................................................................ 109

REFERENCES ........................................................................................................................ 111
INDEX ..................................................................................................................................... 115

available as separate documentation:
APPENDICES : Design documentation
1. INTRODUCTION

1.1 History

The first idea for the transputer stems from 1975, when Ian Barron had the idea that it should be possible to build a processor on a single chip which could be used as a building block for larger parallel computers, or even supercomputers. Along with this processor a high level language had to be provided to use all of its possibilities.

In 1984 the technology had advanced far enough for the first transputer to built. It became possible to place a processor with memory (not much) and communication facilities on a single chip. The first transputer was built by the firm INMOS and was for testing purposes only. In September 1985 the first commercial transputer was made by the same firm. This was the T414a.

The T414a became successful when INMOS produced an adapter for the IBM’s Personal Computer and released the language OCCAM 1, to program the transputer. Unfortunately, there were a number of flaws to the T414a and OCCAM 1. The T414a did not have a floating point unit and OCCAM 1 provided no alternative. OCCAM 1 did not support functions, nor the possibility to send more than one variable through a communication channel.

The T414b and OCCAM 2 solved most of those problems. The transputer was ready for serious applications, such as real time systems and as a building block for supercomputers. During that same period INMOS built a smaller and cheaper variation of the T414, the T212.

The T800, which was introduced in the second half of 1987, was the first transputer with a coprocessor. This coprocessor could be placed on-chip due to new advances in the VLSI technology. The latest news (News94) suggests that INMOS has released a new transputer, the T9000. It has about 10 times the performance of a T800, but the most important architecture improvement that the T9000 offers over its predecessors is that it implements a faster and more flexible communication system.

I will not discuss OCCAM in great depth in this report. This has been done many times before and I don’t consider it essential to design and simulate a transputer.

As VLSI technology progresses, the requirements for computer systems keep increasing. These requirements increase so fast, the VLSI technology can’t keep the pace. To meet these new demands, the Von Neumann architecture has become insufficient. Therefore research in this field concentrates on other architectures.

Decisions have to be made concerning the grain size (i.e. what is the smallest unit that can
be executed in parallel). The transputer uses fine grain parallelism (distribution of instructions), but when communicating with other transputers, it uses medium grain parallelism (distribution of procedures). It is hard to classify the transputer into one particular class as defined by Flynn [Flynn66] (i.e. SIMD, MIMD, etc.). This depends on the environment in which the transputer is used. It can be used stand-alone (SISO) or in a wavefront approach (SPrMD: Single Program Multiple Data), etc. Although the transputer has a reduced instruction set, it cannot be considered as a RISC processor, because it has microcode for complex instructions (CISC) [DeLeeu92].

1.2 Multiprocessor systems

A transputer is a VLSI component, consisting of a processor, memory and communication links for direct point-to-point connections with other transputers. This enables transputers to be connected together to construct multiprocessor systems (see figure 1).

![Transputer network diagram](image)

Figure 1 Transputer network

Also transputers have the ability to execute many software processes, sharing its time
between them automatically, to create new processes rapidly, and to perform communication between processes within a transputer and between processes in different transputers. All of these capabilities are integrated into hardware of the transputer.

Currently INMOS offers the 16 bit T212 transputer, the 32 bit T414 and the T800, which is similar to the T414 but with an on-chip floating point unit. Very soon INMOS will release the T9000, which has a 32 bit pipelined, superscalar processor, a 64 bit floating point unit and a virtual channel processor.

Most of these transputers have 4 full duplex communication links. Except for the T9000, the communication protocol only allows communication between two directly connected transputers. To connect more than four transputers, INMOS made the C004, a transparent programmable link switch designed to provide a full crossbar switch between 32 link inputs and 32 link outputs. This switch is programmable via a separate link called the configuration link.

The T9000 uses a different communication protocol. Here each message is broken into packets, each of which contains a header, data and a packet terminator. This makes it possible for more than one process to use one physical link. The packets of different processes may be interleaved over a single physical link. Each process uses a so called virtual channel. When a packet is received by a transputer, the header is used to determine on which virtual channel the packet was communicated. In order that the packet and depacketisation can be performed without burdening the processor, the T9000 includes a virtual channel processor (VCP) to perform these functions. The VCP occupies about 8% of the chip area and enables the T9000 to support up to 64k virtual channels over its 4 serial links.

The T9000 link system also enables transputers to be connected via a network of C104 packet routers, as shown in figure 2. This allows virtual channels to be established from any transputer to any number of other transputers. The C104 operates by using a packet header to determine the routing needed to send a packet to its destination. The C104 provides routing between 32 links.
1.3 Applications

I have not spent much time on studying applications, because they are not necessary to understand the internal architecture of a transputer. But to broaden your horizon I added a list of applications where transputers could be and have been used:

- High speed multi processor systems
- Workstations and workstation clusters
- Supercomputers
- Real time processing
- Scientific and mathematical applications
- Digital signal processing
- Accelerator processors
- Distributed databases
- System simulation
- Telecommunications
- Microprocessor applications
- Industrial control
- Robotics
- Fault tolerant systems
- Medical instrumentation
- Graphics processing
- Image processing
- Pattern recognition
- Artificial intelligence
1.4 Comparison with other multiprocessor systems

The INMOS transputer is not the only parallel microprocessor. Manufacturers as Texas instruments, Intel and Philips also participate in this field.

1.4.1 TMS320C40

The TMS320C40 (C40 for short) of Texas Instruments is the first parallel-processing DSP [Simar91]. The C40 architecture combines six bidirectional communication ports for direct processor to processor communication, a six channel DMA coprocessor for concurrent I/O, a 32 bits CPU with floating point ALU, 512 bytes instruction cache, 8 Kbytes RAM, 16 Kbytes ROM and two 32-bit timers.

The communication ports consists of an 8-bit wide data bus and four control lines, providing a bandwidth of 20 Mbytes/sec each. Each communication port contains a 64 byte input FIFO buffer and a 64 byte output FIFO buffer. Transfers between processors are performed simply by reading and writing the communication ports at memory mapped addresses. The ports allow a wide variety of topologies to be implemented, like hexagonal grids, 3D grids and 6 dimensional hypercubes.

The DMA coprocessor can move data to and from off-chip memory, on-chip memory and the six communication ports while the CPU processes data. In a single cycle, the DMA coprocessor can perform up to three operations per cycle:

• a 32 bits data transfer
• update an internal address register
• update a transfer counter.

This leads to a maximum performance of 75 MOPS (Million Operations Per Second) with a 25 MHz clock.

The CPU can perform up to eight operations per cycle:

• floating-point or integer multiply
• floating-point or integer addition or subtraction
• two data accesses
• two address register updates
• zero-time branch and loop-counter update.

Special support is also provided for floating-point division, floating-point square-root, byte and half word manipulation and IEEE floating-point format conversions. With a 25 MHz clock this leads to a maximum performance of 200 MOPS.

The C40 has two 32 bits wide external memory interfaces supporting up to 16 Gbytes.
These are the global memory interface and the local memory interface. Each memory interface is capable of running at up to 100 Mbytes/sec [Texas92].

1.4.2 iWarp

iWarp is a multicomputer architecture being developed jointly by Intel and Carnegie Mellon University. An iWarp system is a 2D array of at most 1,024 iWarp cells. Each iWarp cell is composed of one iWarp microprocessor and its local memory [Kung90].

The iWarp microprocessor contains a computation agent and a communication agent. Both share the memory interface and register file.

The computation agent contains three independent, nonpipelined computational units:
- the floating-point adder (10 Mflops at 20 MHz)
- the floating-point multiplier (10 Mflops at 20 MHz)
- the integer/logic unit (20 MIPS at 20 MHz).

The communication agent allows an iWarp cell to connect to four neighbours via eight 40 Mbytes/sec buses (four input and four output).

Intel manufactured the first iWarp microprocessor in December 1989 after the first three and a half years of development. The chip holds more than 650,000 transistors and is fabricated in a 1 μm-CHMOS [Peters91].

1.4.3 POOL and DOOM

In the framework of Esprit project 415, Philips Research Laboratories performed a lot of research and design on the Parallel Object-Oriented Language POOL and the Decentralized Object-Oriented Machine DOOM [Bronne89]. In this language-first approach, DOOM is designed to execute programs in the language POOL.

In the language POOL, a program is subdivided into a large number of so-called objects, which communicate by sending messages. The language offers explicit parallelism and support for structuring large applications to be executed on DOOM.

DOOM is an experimental system for the exploration of parallelism in object-oriented programming. The DOOM architecture consists of a collection of self contained computers, comprising a CPU, local memory and a communication unit to connect these computers via a point-to-point packet switching network [Bronne87].
2. OCCAM

It is not necessary to completely understand the language OCCAM, nevertheless I will give a very short introduction. This will bring a little more understanding of the communication through internal and external channels and the process scheduling.

Transputers can be programmed in most high level languages, such as C, Pascal and Fortran. Where it is required to exploit concurrency, but still to use standard languages, OCCAM can be used as a harness to link modules written in the selected languages. To gain benefit from the transputer architecture, the whole system can be programmed in OCCAM. This provides all the advantages of a high level language, the maximum program efficiency and the ability to use the special features of the transputer.

OCCAM enables a system to be described as a collection of concurrent processes, which communicate with each other and with peripheral devices through channels. OCCAM programs are built from three primitive processes:

\[
\begin{align*}
\text{x := expr} & \quad \text{assign expression expr to variable x} \\
\text{chanl ! expr} & \quad \text{output expression expr to channel chanl} \\
\text{chan2 ? x} & \quad \text{input from channel chan2 to variable x}
\end{align*}
\]

The primitive processes are combined to form constructs:

- **Sequential**: execute processes one after another
- **Parallel**: execute processes concurrently
- **Alternative**: execute only the first ready process

A construct is itself a process, and may be used as a component of another construct (see examples at the end of this chapter).

Conventional sequential programs can be expressed with variables and assignments, combined in sequential constructs. IF, WHILE and CASE constructs are also provided.

OCCAM programs may be configured for execution on one or many transputers. It provides the necessary tools for correctly distributing a program configured for many transputers. Configuration does not affect the logical behaviour of a program. However, it does enable the program to be arranged to ensure that performance requirements are met.

A parallel construct may be configured for a network of transputers by using the PLACED PAR construct. Each component process (termed a placement) is executed by a separate transputer. The variables and timers used in a placement must be declared within each placement process.

On any individual transputer, the outermost parallel construct may be configured to
prioritize its components with the PRI PAR construct. Each process is executed at a separate priority. The first process has the highest priority, the last priority has the lowest priority. Lower priority components may only proceed when all higher priority components are unable to proceed.

Examples:

SEQ

\[
\begin{align*}
\text{c1} & \ ? \ x \\
x & := x + 1 \\
\text{c2} & \ ! \ x
\end{align*}
\]

Inputs a value on channel \texttt{c1}, increments it, and then output the result on channel \texttt{c2}.

PAR

\[
\begin{align*}
\text{c1} & \ ? \ x \\
\text{c2} & \ ! \ y
\end{align*}
\]

Inputs on channels \texttt{c1} and outputs on channel \texttt{c2} concurrently.

ALT

\[
\begin{align*}
\text{c1} & \ ? \ s \\
\text{counter} & := \text{counter} + 1 \\
\text{c2} & \ ? \ s \\
\text{SEQ} \\
\text{c3} & \ ! \ \text{counter} \\
\text{counter} & := 0
\end{align*}
\]

If input on channel \texttt{c1} is ready first, then increment \texttt{counter}, but if channel \texttt{c2} is ready first, then output \texttt{counter} on channel \texttt{c3} and reset \texttt{counter}. 
3. IMS T800

As a guideline to implement a transputer in IDaSS, the IMS T800 transputer is chosen. The T414 is similar to the T800 except for the floating point unit and the T212 is the 16 bit version of the T414. These three transputers have the same internal architecture, with the T800 as most extensive one. The T9000 is not chosen, because it has a completely different architecture and it uses a different communication protocol. Maybe the T9000 could be modelled and simulated in IDaSS in next graduation projects. But now, in this chapter the internal architecture of a T800 transputer will be described briefly, to understand what I used as a guideline to my implementation in IDaSS.

The IMS T800 transputer is a 32 bit CMOS microprocessor with a 64 bit floating point unit, 4 Kbytes on-chip RAM, an external memory interface, two 32 bit timers, four standard INMOS communication link interfaces and one event channel. The block diagram of the T800 is shown in figure 3.

Figure 3 T800 block diagram

Not drawn in this block diagram are the power supply pins, the reset input, the clock input and the processor clock output. These signals don’t need to be implemented in IDaSS.
3.1 System services

The internal clock frequency can be selected by the 3 bit input 'ProcSpeed0-2', according to table I. The internal clock is derived from the external clock, which is standard 5 MHz for all kind of transputers. Inclusion of a frequency selection in this table does not imply immediate availability.

<table>
<thead>
<tr>
<th>'ProcSpeed0-2'</th>
<th>internal clock frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20.0</td>
</tr>
<tr>
<td>1</td>
<td>22.5</td>
</tr>
<tr>
<td>2</td>
<td>25.0</td>
</tr>
<tr>
<td>3</td>
<td>30.0</td>
</tr>
<tr>
<td>4</td>
<td>35.0</td>
</tr>
<tr>
<td>5</td>
<td>invalid</td>
</tr>
<tr>
<td>6</td>
<td>17.5</td>
</tr>
<tr>
<td>7</td>
<td>invalid</td>
</tr>
</tbody>
</table>

The transputer can be bootstrapped either from a link or from external ROM. If input 'BootFromROM' is connected high, the transputer starts to execute code from address $7FFFFFFE_{16}$. This location should contain a backward jump to a program in ROM. If 'BootFromROM' is connected low the transputer will wait for the first bootstrap message to arrive on any of its links.

If 'Analyse' is taken high when the transputer is running, the transputer will halt at the next descheduling point. The registers will contain special values for analysis purposes. Input links will continue with outstanding transfers. Output links will not make another access to memory for data but will transmit only those bytes already in the link buffer.

The 'Error' pin carries the internal Error flag.
3.2 External memory interface

The memory is word aligned on four byte boundaries (databus ‘Data’ is 32 bits wide). A read from memory (input ‘Read’ high) will always hand over a word of 4 bytes together. To extract one byte out of this word an extra instruction needs to be executed. When writing, four write strobes ‘Write0-3’ are used to determine which bytes within the word have to be written into the memory. So no read-modify-write is needed for writing part words. Slow memory can introduce wait states with input ‘Wait’.

Since the address bus ‘Address’ has a width of 30 bits, the address space is $2^{30}$ words = 1 Gwords = 4 Gbytes. A byte pointer has a width of 32 bit, where the two least significant bits determine the byte within the word (the least significant byte of a word is the lowest addressed byte). Word pointers have a width of 30 bits and are stored in the highest 30 bits of a 32 bit register. As a consequence the lowest 2 bits are free for other information.

The T800 supports direct memory access (DMA). This DMA enables an external device to read and write the external memory. DMA can be requested by taking input ‘DMAreq’ high. When the transputer transfers control of the bus, the ‘Address’, ‘Read’ and ‘Write’ outputs are tristated and output ‘DMAgranted’ is asserted high. The processor is still able to access its internal memory, however external memory is out of reach until the end of the DMA request.

The External Memory Interface (EMI) supports dynamic and static RAM as well as ROM and EPROM. EMI timing can be configured at reset to serve most memory types and speeds. There are 13 internal configurations which can be selected by a single pin connection. If none are suitable the user can configure the interface to specific requirements. I’m not going to describe these configurations, because such a memory interface has already been implemented in IDaSS by E.B.C. Daniëls and falls outside the scope of this graduation project. The implementation I have done, is a static memory interface with DMA control.

3.3 Internal RAM

The T800 has 4 Kbytes of internal static memory. Each internal memory access takes one processor cycle. With a 30 MHz internal clock the bandwidth is 30 Mwords/sec. = 120 Mbytes/sec.

Internal memory starts at the most negative address $80000000_{16}$ and extends to
80000000₁₆ through 8000006F₁₆ are used as auxiliary processor registers and the rest is free to use by the user. See figure 4 for a complete memory map.

![Figure 4 Memory map](image)

External memory space starts at 80001000₁₆ and extends up through 00000000₁₆ to 7FFFFFFF₁₆. Memory configuration data used by the External Memory Interface must be in the address space from 7FFFFF6C₁₆ to 7FFFFFFD₁₆. The bootstrapping code start at address 7FFFFFFE₁₆.

### 3.4 CPU

#### 3.4.1 Registers

The transputer has a small 32 bit register set, which consists of a workspace pointer \( W \), an instruction pointer \( I \), an operand register \( O \), and three registers forming an evaluation stack: \( A \), \( B \) and \( C \). The workspace pointer is a word pointer to an area of memory where local variables of the current process are held. Each process occupies an area of memory as his workspace. The priority of the current process is stored in the least significant bit of register \( W \) (0 = high, 1 = low priority). This workspace pointer together with the priority is called the *Workspace descriptor*. The instruction pointer is a byte pointer to the next instruction to be executed. The operand register is used in the formation of instructions.
3.4.2 Instructions

Each instruction consists of a single byte. The four most significant bits (upmost nibble) form the function and the four least significant bits (lowest nibble) contain data. All instructions are executed by loading the four data bits into the least significant four bits of the operand register, which is then used as the instruction’s operand (see figure 5).

![Figure 5 Instruction representation](image)

To load more data bits in the operand register, two instructions *pfix* (prefix) and *nfix* (negative prefix) can be executed. The *pfix* instruction loads its four data bits into the operand register and then shifts the operand register up four places. The *nfix* instruction is similar, except that it complements the operand register before shifting up. All other instructions clear the operand register after execution.

In this representation only 16 instructions can be defined, which is not enough to use all features of a transputer. To get more than 16 instructions, one instruction *opr* (operate) causes the operand register to be interpreted as the function. This provides a potential number of $2^{32} = 4$ Giga instructions. But of course this number of instructions has not been defined, yet. Most instructions only need the least significant 8 bits of the operand register, which makes them 2 byte instructions (a *pfix* and an *opr*).

3.4.3 Scheduler

The scheduler enables any number of concurrent processes to be executed together, sharing the processor time. This removes the need for a software kernel. At any time a process is in one of the next states:
**Active**  
- Being executed.  
- Waiting to be executed in the *process queues*.

**Inactive**  
- Waiting for an internal channel to become ready to communicate.  
- Inputting or outputting a message on a link.  
- Waiting until a specified time in one of the timers.  
- Waiting until a specified time in one of the *timer queues*.  
- Waiting for an external event to occur.

The scheduler operates in such a way that inactive processes do not consume any processor time. A process in execution that has to wait for something, is descheduled (it becomes inactive). Then the first process from the process queues can be taken in execution. For process switching the current process has to be taken out of execution and added to the end of the process queues (it stays active). When an inactive process is able to continue the scheduler schedules it, which means add it to the end of the process queues. At last a new process can be added to the process queues. These transitions are graphically represented in figure 6.

![Figure 6 Process state transitions](image)

Active processes which are waiting to be executed are held in two process queues, one of high priority processes and one of low priority processes. These process queues are implemented as linked lists of the workspaces of the processes. The scheduler holds in register *FPtr0* the pointer to the head and in register *BPtr0* the pointer to the tail of the linked list of active high priority processes. Likewise the scheduler holds the registers *FPtr1* and *BPtr1* for the active low priority processes as shown in figure 7.
Every process has saved his instruction pointer (Iptr) at address workspace pointer - 1. Address workspace pointer - 2 contains the workspace pointer of the next process (next Wptr) or the value $80000000_{16}$ as end mark. When the queue is empty $FPtr$ holds $80000000_{16}$. The process which is in execution by the processor is not in the process queues. The processor holds its workspace pointer in register $W$.

### 3.4.4 Communication through channels

Communication between processes is achieved by means of channels. An internal channel is implemented by a single word in memory, an external channel is implemented by a link between two transputers. The instructions *input message* and *output message* use the address of the channel to determine whether the channel is internal or external. The following addresses are reserved for external channels:

- $80000000_{16}$ Link 0 Input
- $80000004_{16}$ Link 0 Output
- $80000008_{16}$ Link 1 Input
- $8000000C_{16}$ Link 1 Output
- $80000010_{16}$ Link 2 Input
- $80000014_{16}$ Link 2 Output
- $80000018_{16}$ Link 3 Input
- $8000001C_{16}$ Link 3 Output
Both instructions *input message* and *output message* use three parameters: a channel address, the number of bytes to transfer and the message pointer, that is a byte pointer to the message start. Communication takes place when both the inputting and outputting processes are ready. Consequently, the process which first becomes ready must wait until the second one is ready.

**Internal channel communication**

Before a word in memory can be used as an internal channel for the first time, it must be initialised to 80000000₁₆. The first process that becomes ready will write its workspace descriptor to this channel address, write the message pointer at address Wptr - 3, and the processor starts to execute the next process from the process queues.

When the second process becomes ready, it will notice that another process is waiting, because the channel address does not contain the value 80000000₁₆. The second process reads the workspace descriptor of the first process from the channel address and the first message pointer from that workspace pointer - 3. Then it copies a number of bytes from the second message pointer to the first message pointer or the other way around if the second process is an inputting process. It does not matter whether the inputting or the outputting process becomes ready first.

The second process specifies the length of message actually transferred. If the second process is an outputting process with longer message length than the inputting process, then the input message buffer will overflow (data after the message buffer is overwritten). If in this case the length is shorter, then the output message will not completely fill the input message buffer. If the second process in an inputting process with longer message length, then the input buffer will get undefined data from addresses after the output message. If in this case the length is shorter, then the input buffer will not get the output message completely.

**External channel communication**

When the channel address points to an external channel the processor delegates to an autonomous link interface the job of transferring the message. That is the processor writes the workspace pointer of the process, the number of bytes to transfer and the message pointer into registers of the link interface and deschedules the process. When the message has been transferred the link interface causes the scheduler to reschedule the process. This allows the processor to continue the execution of other processes whilst the external message transfer is taking place. The protocol ensures that it does not matter which process in which transputer first becomes ready.
The length problem also occurs in external channel communication, but has different consequences. The process with the shorter message length will successfully transfer data, while the other process will hang, waiting to send or receive more data. Still the processor of that transputer can execute other processes, but that link interface stays occupied. Next communication over that link can release the hanging process, but will result in new hanging processes in both transputers, and so forth. In short, all communications over this link after the length error will fail. However the error flag of the processor will not indicate these errors. This problem can be solved by defining a new protocol above the existing. For example, the first four bytes to transfer indicate the length of the message. The T9000 protocol solves the length problem.

### 3.5 Floating point unit

The 64 bit FPU provides single and double length arithmetic to floating point standard ANSI-IEEE 754-1985 [IEEESt85]. It is able to perform floating point arithmetic concurrently with the CPU, sustaining in excess of 2.25 Mflops on a 30 MHz device. All data communication between memory and the FPU occurs under control of the CPU.

The FPU consists of a microcoded engine with a three deep floating point evaluation stack for manipulation of floating point numbers. These stack registers are \( FA, FB \) and \( FC \), each of which can hold either 32 bit or 64 bit data; an associated flag, set when floating point value is loaded, indicates which. The stack behaves in a similar manner to the CPU stack.

I didn’t spend any time on this block, because such a FPU has already been implemented in IDaSS. On this project have been working F.J.A. Korsten, G.H. Hotho, M. van Balen and A.C. Verschueren.

### 3.6 Links

The T800 has four identical bi-directional serial links. Each link comprises an input channel and an output channel. In figure 8 process A outputs a message on link interface 0 of transputer 1 to process Q via link interface 3 of transputer 2. At the same time process B inputs a message on that same link from process P. Every data byte sent on a link is acknowledged on the input of the same link, thus each signal line carries both data and control information. In figure 8 process A sends a data byte to process Q, and process Q must send an acknowledge back to process A for being ready to receive another data byte. These acknowledges are interleaved with the data bytes sent by process P.
The quiescent state of a link output is low. Each data byte is transmitted as a high start bit followed by a one bit (to discriminate between a data packet and an acknowledge packet) followed by eight data bits followed by a low stop bit (see figure 9). The least significant data bit is transmitted first. An acknowledge consists of a high start bit followed by a low stop bit.

![Figure 8 Link configuration](image)

Figure 8 Link configuration

The sending link reschedules the sending process only after the acknowledge for the final byte of the message has been received. It is allowed to send an acknowledge packet before the data packet has been fully received, which is called overlapped acknowledge.

![Figure 9 Data and acknowledge packets](image)

Figure 9 Data and acknowledge packets

The standard communication speed is 10 Mbits per second, but in addition 5 and 20 Mbits per second can be used. The link speed can be selected by the pins 'LinkSpecial', 'LinkOSpecial' and 'Link123Special' (see table II). The link 0 speed can be set independently.
<table>
<thead>
<tr>
<th>Link Special</th>
<th>Linkn Special</th>
<th>Mbits/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

Links are not synchronised with the processor clock. Thus links from independently clocked systems may communicate, providing only that the clocks are nominally identical and within specification.

### 3.7 Events

When an external event takes input pin ‘EventReq’ high the event channel is made ready to communicate with a process. When both the event channel and the process are ready the processor takes the output pin ‘EventAck’ high and the process, if waiting, is scheduled. ‘EventAck’ is removed after ‘EventReq’ goes low.

Only one process may use the event channel at any time. If no process requires an event to occur ‘EventAck’ will never be taken high. Although ‘EventReq’ triggers the channel on a transition from low to high, it must not be removed before ‘EventAck’ is high.

Setting a high priority process to wait for an event input is a way of interrupting a transputer program.

### 3.8 Timers

The transputer has two 32 bit timers, a high priority timer $T_0$ and a low priority timer $T_1$. This means that timer $T_0$ is accessible only to high priority processes and timer $T_1$ only to low priority processes. Timer $T_0$ is incremented every microsecond, cycling completely in approximately 1 hour and 12 minutes. Timer $T_1$ is incremented every 64 microsecond, which makes a full period of approximately 76 hours.
Each timer has three 32 bit registers:

<table>
<thead>
<tr>
<th>Timer $T0$</th>
<th>Timer $T1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Clock0$</td>
<td>$Clock1$</td>
</tr>
<tr>
<td>$TNextReg0$</td>
<td>$TNextReg1$</td>
</tr>
<tr>
<td>$TPtrLoc0$</td>
<td>$TPtrLoc1$</td>
</tr>
</tbody>
</table>

- $Clock0$ is the current value of the timer
- $TNextReg0$ is the next time to deschedule a waiting process
- $TPtrLoc0$ is the pointer to the first process on the timer queue
- $Clock1$ is the current value of the timer
- $TNextReg1$ is the next time to deschedule a waiting process
- $TPtrLoc1$ is the pointer to the first process on the timer queue

The current value of the current priority timer can be read by executing a *load timer* instruction. A process can wait for a specified time with the *timer input* instruction. If this time is in the 'past' then the process continues immediately. If the time is in the 'future' the process is descheduled. When the specified time is reached the process is scheduled again.

When the timer value reaches the most positive integer ($7FFFFFFF_{16}$) its value wraps round to the most negative integer ($80000000_{16}$). When comparing a specified time with the current time $Clockx$, times between $Clockx$ and $Clockx + \text{unsigned} \ 80000000_{16}$ are considered to be in the future and all other values of the specified time are considered to be in the past.

All processes waiting for a time are linked in a sorted list, the *timer queue*. For both timers there is a timer queue and a pointer to the first item in the timer queue is stored in the register $TPtrLocx$. This pointer points to the workspace of the first waiting process. At workspace - 5 the time to wait for is stored and at workspace - 4 the workspace pointer to the next waiting process in the linked list is stored (see figure 10).
Figure 10 Timer queue
At the Digital Systems Group of the Faculty of Electrical Engineering at Eindhoven University of Technology, a project named 'Structured Analysis and Structured Design' (SASD) was started a few years ago to develop tools, to design and simulate Ultra Large Scale Integrated (ULSI) circuits. The 'Interactive Design and Simulation System for Ultra Large Scale Integration' (IDaSS for ULSI) is one of the tools resulting from this SASD project [Versch90].

IDaSS is a software tool written by Dr. Ir. A.C. Verschueren in Smalltalk and runs on IBM PC/AT-compatibles. It allows the user to graphically design complex logic systems at a very high level where there are no limitations posed by the silicon technology in which the design has to be ultimately implemented. The designer can build and interactively simulate any logic system using building blocks like registers, memories (RAM, ROM, FIFO, LIFO and CAM), operator blocks (which can describe any asynchronous logic) and state machine controllers.

The current 'Object Oriented (Hardware) System Design' (OOSD) project concentrates on the creation of an IDaSS design library of re-usable processor cores and input/output controllers. Within this library, building a custom processor to perform specific tasks is eased to the point that only a few library components must be picked from the library, interconnected with IDaSS and converted to silicon.

To be able to interconnect the elements of this library, strict specifications for the interfaces must be followed. Specifications of the following interfaces are defined:

- (internal) memory interface
- Input/output interface
- Interrupt interface

The current IDaSS design library contains several microprocessors: Motorola 6801 and 68000 and Intel 8051 and 8085, a bus switch to connect a number of processors with a number of memories, static/dynamic memory interface, caches, IEEE floating point processor, DES (Data Encryption Standard) processor, DSP's: TMS32010 and Motorola 96002, Ethernet controller, token data processor (NEC 7281), etc. At this moment work is done on a fuzzy logic processor, a superscalar processor and of course a T800 transputer. To fit the transputer in this library it must obey the interface specifications.
4.1 Converting an IDaSS design to silicon

A finished IDaSS design can be transformed into logic gates (synthesis) that can in turn be transformed into a silicon layout (compilation). For automatic synthesis a number of ASA tools are provided by Sagantec. At this moment the synthesis poses a number of problems, because not all IDaSS features can be translated into equivalent logic gates.

If the designer restricts his design it is possible to create a silicon layout. I tried to take into account these restrictions, but I’m not sure my design is fully synthesizable, because I never came to synthesis. After all, these problems may solve in the future, because the Digital Systems Group is working on a compiler from IDaSS to VHDL.

One of the problems is the asynchronous ‘IDaSS RAM’, which will be translated into a synchronous ‘ASA RAM’. In order to simulate the behaviour of the ‘ASA RAM’ a substitution schematic is provided by A.C. Verschueren. In my design I used this substitution schematic as the RAM building block. Before synthesis these ASA RAMs must be replaced by the IDaSS RAMs.

1 The IDaSS RAM outputs the read data asynchronously, while the ASA RAM outputs the read data after one clock period. Writing is done synchronously in both types of RAM.
5. IMPLEMENTATION OF THE TRANSPUTER

This chapter provides an overview of the implementation of the internal architecture of the transputer. Furthermore implementation considerations at this level are given.

In the block schematic of the transputer (see figure 11) the blocks of the IMS T800 can be found again. I did this on purpose to ease comparison. The block ‘Serv’ contains the system services, ‘RAM’ is the internal RAM, ‘EMI’ is the external memory interface, ‘mux4x2’ is a bus multiplexer, ‘CPU’ is the central processing unit including the executor and the scheduler, ‘Timers’ consists of two timers, ‘Links’ contains the four link interfaces and ‘Event’ is the event channel.

Figure 11 Block schematic of the T800
5.1 Interfacing with the outside world

The interface with external memory consists of the following connectors:

- 'a' = 30 bits address output
- 'd' = 32 bits data input/output
- 'c' = 5 bits control output
- 'h' = 1 bit handshake input.

Where the control output 'c' carries the read and write strobes:

- bit 0: read a 32 bits word
- bit 1: write byte 0 (least significant byte of the 32 bits word)
- bit 2: write byte 1
- bit 3: write byte 2
- bit 4: write byte 3 (most significant byte of the 32 bits word).

Handshake input 'h' becomes high in the last clock cycle of a read or write cycle.

Direct memory access (DMA) can be requested by taking input 'DReq' high. When the transputer transfers control of the bus the address and control outputs are tristated and output 'DGra' (DMA Granted) is asserted high.

The four inputs 'Un' and the four outputs 'LOn' are the serial inputs and outputs of the four link interfaces. They answer the INMOS communication protocol. Their speed can be selected with the 8 bits input 'LS':

- bit 0..1 = LOS (speed of link 0)
- bit 2..3 = L1S (speed of link 1)
- bit 4..5 = L2S (speed of link 2)
- bit 6..7 = L3S (speed of link 3).

This is unlike the IMS T800, where the speed of link 1, 2 and 3 are always the same. In my design the speed of each link can be selected independently. I defined the link speed selection according to table III, where LnS is a two bits part of input 'LS' concerning link n.
5. IMPLEMENTATION OF THE TRANSPUTER

<table>
<thead>
<tr>
<th>LnS</th>
<th>speed of link n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5 Mbits/sec</td>
</tr>
<tr>
<td>1</td>
<td>10 Mbits/sec</td>
</tr>
<tr>
<td>2</td>
<td>20 Mbits/sec</td>
</tr>
<tr>
<td>3</td>
<td>invalid</td>
</tr>
</tbody>
</table>

When an external event takes input ‘EReq’ (Event request) high the event channel is made ready to communicate with a process. When both the event channel and the process are ready the processor takes ‘EAck’ (Event acknowledge) high and the process, if waiting, is scheduled. ‘EAck’ is removed after ‘EReq’ goes low.

Input ‘PS’ (= Processor Speed) selects the internal clock frequency. This is defined exactly like the input ‘ProcSpeed0-2’ of the IMS T800 (see table I). The inputs ‘Ana’ (= Analyse) and ‘BFR’ (= BootFromRom) are not connected yet. The implementation of their functionality is not essential. It is just additional hardware. Output ‘E’ (= Error) carries the error flag of the CPU.

5.2 Internal memory interfacing

All internal memory interfaces are the same as the external memory interface. Generally a memory interface consist of four connectors, called ‘a’, ‘d’, ‘c’ and ‘h’, which stands for respectively address, data, control and handshake. The bus multiplexer ‘mux4x2’ connects four processors with two memories. This bus multiplexer is based on the bus switch designed by Mark Megens during his practical period. The CPU has three interfaces, one for the fetcher, one for the load/store unit and one for the scheduler.

The four links share one memory interface. This is sustainable, because the link interfaces do not perform memory requests frequently. I will estimate the maximum frequency: in full duplex mode a link interface sends and receives data packets of 11 bits and acknowledge packets of 2 bits interlaced. At 20 Mbits/s with a 20 MHz clock every 11 + 2 = 13 clock cycles one byte can be send and received with an acknowledge. A link interface loads and stores four bytes together. This leads to one store action every 4 * 13 = 52 clock cycles and one load action every 52 clock cycles. In other words, one memory request every 52 / 2 = 26 clock cycles. If all four links work concurrently in full duplex mode the shared memory interface could reach an average speed of one memory request every 26 / 4 = 6.5 clock cycles. This is still sustainable, when a read or write cycle takes at most six clock cycles. Of course usually the speed of memory requests is lower.
5.3 The Z-bus protocol

Because inactive processes do not consume any processor time, the scheduler must be able to add processes to the process queues without interrupting the executer. Inactive processes are not in the process queues, but their workspace descriptors are held in registers in the link interfaces, the timers and the event channel. If an inactive process becomes ready, these blocks requests the scheduler for scheduling the process again. These schedule requests go through the buses ‘rT’, ‘rL’ and ‘rE’ to the scheduler. The scheduler must fetch the workspace descriptor and add it to the process queues. When a timer was ready the scheduler must write the next timeout of the timer queues into a register of the appropriate timer. The executer can also request for scheduling new processes. Therefore the scheduler must be able to communicate with the executer, links, timers and event channel.

Furthermore the executer must be able to read and write registers in the scheduler, link interfaces, timers and event channel. For example to start a link process, the number of bytes to send or receive and the message pointer must be written into registers of a link interface. Some instructions ask for reading the clock register of a timer, or the process queue pointers in the scheduler. Furthermore when the executer deschedules a process it writes the workspace descriptor of the process into a register of the appropriate block. All these tasks ask for communication between the executer and all these blocks.

For all these communications I designed one bus architecture, consisting of two buses: the U-bus and Z-bus, together called the ‘Z-bus’ for short. The Z-bus is a 32 bits bidirectional data bus, controlled by the U-bus. The U-bus is a 8 bits unidirectional control bus, driven by the executer or the scheduler. The U-bus holds a block address, a register address and a read/write signal. Handshaking is not needed, because the data at the Z-bus is always valid within one clock cycle.

The U-bus represents:
- bit 0 1 = Read, 0 = Write
- bit 1..3 register address
- bit 4..7 block address.

The block addresses are:

- 0 = used as rest value
- 1 = Executer (inside the CPU)
- 2 = Scheduler (inside the CPU)
- 3 = Event
- 4 = Timer 0
- 5 = Timer 1
- 6 = Link 0 In
- 7 = Link 0 Out
- 8 = Link 1 In
- 9 = Link 1 Out
- 10 = Link 2 In
- 11 = Link 2 Out
- 12 = Link 3 In
- 13 = Link 3 Out
- 14 = Executer (inside the CPU)
- 15 = unused.
The definition of the register addresses is dependent on the block type. These register addresses will be defined in the concerning block descriptions, later in this report. In each block the workspace register has register address 0.

The reason that the executer has two block addresses is that it can perform two different tasks. Block address 1 is used when the executer wants to schedule a process, while block address 14 is used when the executer wants to start the next ready process from the process queues. This enables the scheduler to order all requests so that the request number equals the block address. Besides the order of block addresses determines the order of priority of their requests (the lowest number has the highest priority).

The scheduler and the executer can both be master, which means that they are able to control the U-bus. Therefore they must conquer for it via an arbiter. All blocks can be slave, including the executer and the scheduler. This enables the executer and the scheduler to exchange information. The U-bus addresses a register in another block. If the read bit is set the register must output its value on the Z-bus, to enable the master to read the register. If the read bit is not set the register should load the value on the Z-bus, to enable the master to write to the register.

5.4 Timing of the links and timers

Because IDaSS cannot define absolute clock frequencies, some suboptimal solution has been made. In IDaSS there is only one system clock and no other clocks can be defined. Every step you make, the complete system will be stepped forward. To slow down some parts of the system, a counter can be used. In this way the system clock can be divided by whole numbers.

As we saw before the system clock is defined by the input 'PS'. To increment timer 0 every microsecond, a precounter is used. The reset value of this precounter is carried by output 'CT' of the block 'Serv'. Timer 1 can easily be derived from timer 0.

The links can input or output one bit per clock cycle or less. The number of clock cycles between the bits is determined by output 'CL' of block 'Serv'. This signal carries three values, one for each link speed (5, 10 and 20 Mbits/s).
6. TEST SYSTEM

This chapter shows the test system, that has been used to load and execute programs, to test the link interfaces and other functions.

To load and execute programs a ROM is connected with the transputer. After reset the transputer will fetch instructions from the ROM. Also a RAM is connected to test loading and storing data in external memory. Because the transputer has only one memory interface, extra hardware (the 'selector') is needed to connect both ROM and RAM with this interface (see figure 12).

![Figure 12 Test system](image)

The address space of the 8 Kbytes ROM is situated in the upmost memory address space (from address 7FFFFFF00 to 7FFFFFFF), while the address space of the 4 Kbytes RAM is situated just above the internal memory (from address 80001000 to 80001FFF). The number of wait states\(^2\) of these memories is adjustable from 0 to 3.

The selector passes the control bus on to the memory selected by the address output of the transputer. This selector holds the control bus of the unselected memory inactive. The handshake of the selected memory is passed on to the transputer. The data buses are directly connected with each other. The selector passes the address buses on after being adjusted to their width.

\(^2\) Here wait states are defined as the number of clock cycles before ending a read or write cycle with a high handshake signal.
The link interfaces can be tested by connecting them with other link interfaces. Normally link interfaces of different transputers are connected with each other. But this implies in IDaSS that the transputer block must be copied. This doubles the memory demand of the simulator. Although this would be possible with a 8 Mbytes RAM computer, the simulation time increases disastrously. An alternative is to connect two link interfaces of the same transputer with each other. This does not really increase the memory demand and the simulation time. In this test system link 0 is connected with link 1 and link 2 is connected with link 3. This makes a lot of test situations possible.

As you can see in figure 12, register 'LS' drives the link speed selection input 'LS'. In this case the speed of links 0, 1, 2 and 3 are set to respectively 10, 10, 20 and 20 Mbits/sec (see table III). Obviously links that are connected with each other must use the same speed.

The event input 'EReq' can be taken high by setting register 'EReq' to one. A viewer shows the value of output 'EAck'. Manually a request can be simulated by setting register 'EReq' and resetting it after output 'EAck' becomes high. Of course a process must ask for an event, otherwise output 'EAck' will never become high.

Registers 'Ana' and 'BFR' drive the corresponding inputs, but they don’t care because their functionality is not implemented yet. A viewer shows the value of the error flag at output 'E'. The internal clock frequency is selected by register 'PS'. In this case I have chosen for 20 MHz (see table I). The reason for this choice is that all link speeds can be derived from this clock frequency. Obviously you should not change the registers 'PS' and 'LS' dynamically.

The DMA request can be simulated by setting register 'DReq' to one. Output 'DGra' will become high as soon as possible. You should manually reset this register after some time to end the DMA request.
7. IMPLEMENTATION OF RAM AND ROM

This seems to be trivial, but it is not. This RAM must be able to write partial words, and both RAM and ROM must generate a handshake signal after some wait states.

7.1 Internal and external RAM

With the asynchronous IDaSS RAM it would be possible to read, modify and write a word within one clock cycle. But as said before, to make this design synthesizable, the synchronous ASA RAM must be used. To be able to write partial words with ASA RAMs I divided the 32 bit wide RAM into four ASA RAMs of 8 bits wide each (see figure 13).

Figure 13 RAM

The blocks 'M0', 'M1', 'M2' and 'M3' are four ASA RAMs. The operator 'SW' divides the 32 bits data input 'i' into four parts of 8 bits data outputs 'o0' through 'o3'. The four write strobes in the control input 'c' are distributed over the four ASA RAMs. So only the selected RAMs write a byte and the others don't. Because reading always asks for the whole 32 bits word, the read strobe of input 'c' goes to all the RAMs. The 32 bits data output 'o' is simply a concatenation of the four data inputs 'i0' through 'i3'. The three-state data output 'o' is enabled as long as the read strobe is active.

The handshake must become high after a number of clock cycles (wait states). This number is counted with the 2 bits register 'hR'. Operator 'SW' increments this register as long as input 'c' has active read or write strobes. When the register reaches a certain number the handshake output 'h' becomes high. This also resets register 'hR' via its control input, which overrides the incrementing. The reset value of this counter is adjustable from 0 to 3. For the internal RAM this number must always be 1.
7.2 ROM

A ROM is simpler to implement than a RAM, because it cannot write. So the problems with writing part words disappear in this case. The only problem is generation of the handshake signal. But this can be done in the same way as in the RAM. The schematic of the ROM is shown in figure 14.

![Figure 14 ROM](image)

The read strobe of input 'c' enables the 32 bits three-state data output 'rd' of the ROM. Operator 'CH' increments the 2 bits handshake register 'hR' only when the read strobe is active. Obviously the write strobes have no effect at all. Furthermore the handshaking goes the same as in the RAM.
8. IMPLEMENTATION OF THE BUS MULTIPLEXER

The bus multiplexer is an important part of the transputer, because it enables communication with the internal and external memory at the same moment.

The bus multiplexer in this design connects two memories with four processors (see figure 15).
The internal memory is connected with interface ‘m0’ (consisting of the connectors ‘am0’, ‘dm0’, ‘cm0’ and ‘hm0’) and external memory with interface ‘m1’.

This bus multiplexer uses fixed priority in order of processor interface number:
- low priority interface ‘p0’: fetcher of the CPU
- interface ‘p1’: load/store unit of the CPU
- interface ‘p2’: links
- high priority interface ‘p3’: scheduler of the CPU.

In this context I will call these four memory requesters ‘processors’.

The considerations for this choice are given here. The fetcher and the load/store unit will perform a lot of memory requests. To prevent starvation of the other requests, they are given lower priority. The load/store unit must have higher priority than the fetcher, because the execution of an instruction directly waits for this. If all instructions in the CPU are executed, the load/store unit will stop requesting for memory, so the fetcher can fetch the next instructions. The scheduler must have the highest priority, because it must be able to schedule processes that became ready.

The blocks ‘ARBO’ and ‘ARB1’ are the arbitration blocks, ‘MUX00’ through ‘MUX31’ are the multiplex units and ‘S0’ through ‘S3’ are selection operators.

Because the four processors can request for memory communication at the same moment, arbitration is needed. The multiplex units place requests at inputs ‘req0’ through ‘req3’ of the arbiters. The arbiter grants one of the requests access with an acknowledge (one of the output ‘ack0’, ‘ack1’, ‘ack2’ or ‘ack3’ goes high). The acknowledged multiplex unit will connect the processor with the memory.

This bus multiplexer is based on the bus switch designed by Mark Megens during his practical period. His design is a very universal solution for bus switching. I minimized his design for this application. For example, I removed the address selection from the multiplex units and put it into a selection operator. Also I changed the arbitration from rotating priority into fixed priority. I am not going to describe all adaptations, because that would describe a lot of complicated logic I did not use.

### 8.1 Selection

The selection operator select one of the two memories dependent on the address if a read or write request is active. If the address input ‘a’ carries a value between 80000000\text{H} and 80000FFF\text{H}, the internal memory is selected with output ‘S0’, else the external memory is selected with output ‘S1’. These 2 bits outputs not only select a memory, but also pass on the decoded read or write requests:
00₂ not selected
01₂ selected for reading
10₂ selected for writing
11₂ invalid.

The selection operator performs a second task. If the processor does not request for reading or writing the handshake outputs from the multiplex units are three-stated. Most processors cannot handle an undefined handshake signal. Therefore this operator forces the handshake output 'hp' low if no request is performed, else output 'hp' follows input 'h'.

8.2 Arbitration

Both arbiters 'ARBO' and 'ARB1' have exactly the same contents, shown in figure 16.

![Figure 16 Arbiter](image)

Operator 'AO' describes the actual arbitration logic. The arbitration is relatively simple compared with the design of Mark Megens, because it uses fixed priority. The order of priority is (from high to low) 'req3', 'req2', 'req1' and 'req0'.

The 3 bits status register holds the number of the processor in service:

- bit 0 : 1 = serving a processor, 0 = no processor is in service
- bit 1..2 : processor number being served.

The memory ends a read or write cycle by driving the handshake input 'hm' high in the last
clock cycle of a read or write cycle. This signal resets the status register to zero, meaning no processor is in service. This enables new requests to come in service.

Because memory must always have a defined control bus, the arbiter drives this bus if no request is performed. Therefore operator ‘CO’ enables three-state output ‘cm’, which is always zero.

8.3 Multiplex unit

The multiplex unit is the real connection between a processor and a memory. All multiplex units are the same, except that the units for the internal memory narrow the address bus to 10 bits. Figure 17 shows the contents of a multiplex unit.

![Multiplex unit diagram]

The operator ‘CTRL’ makes output ‘req’ high when it is selected by input ‘sel’. Asynchronously or after some clock cycles input ‘ack’ will become high. When the memory is busy with another processor, the acknowledge only comes after some clock cycles. During this time the switch is placed in the ‘Hold’ state. Output ‘c’ follows input ‘sel’ and adds the Hold state:

- **00** no request
- **01** Read: requested and acknowledged for reading
- **10** Write: requested and acknowledged for writing
- **11** Hold: requested but not acknowledged yet.

Operator ‘SWITCH’ simply connects all buses, except in the Hold state output ‘hp’ is
forced low. All three-state outputs are default disabled. In the Read state the three-state outputs 'am', 'dpo', 'cm' and 'hp' are enabled. In the Write state the three-state outputs 'am', 'dmo', 'cm' and 'hp' are enabled. In the Hold state only the three-state output 'hp' is enabled.
9. IMPLEMENTATION OF SYSTEM SERVICES

The implementation of system services is very simple in IDaSS. No power supply pins, clock input and reset input is needed. What I did implement describes this chapter.

The block ‘Serv’ gives timing parameters for the timers and links dependent on input ‘PS’ and passes on the error flag (see figure 18).

![Block diagram of Serv](image)

Figure 18 Contents of block ‘Serv’

Output ‘CT’ carries the reset value for the precounter of timer 0 according to table IV. This table is implemented as a rom, named ‘CTROM’. ‘PS’ is the 3 bits address input and ‘CT’ is the 6 bits data output.

<table>
<thead>
<tr>
<th>‘PS’</th>
<th>clock freq.</th>
<th>‘CT’</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20.0 MHz</td>
<td>19</td>
</tr>
<tr>
<td>1</td>
<td>22.5 MHz</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>25.0 MHz</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>30.0 MHz</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>35.0 MHz</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>invalid</td>
<td>unknown</td>
</tr>
<tr>
<td>6</td>
<td>17.5 MHz</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>invalid</td>
<td>unknown</td>
</tr>
</tbody>
</table>

This does not always result in running timer 0 at exactly 1 MHz (incrementing every microsecond). When the internal clock frequency is 22.5 MHz, timer 0 runs at 0.978 MHz and with 17.5 MHz at 0.972 MHz. With all other internal clock frequencies timer 0 runs at exactly 1 MHz.

With partial frequency control it is possible to run timer 0 at exactly 1 MHz with a clock
of a non multiple frequency. For example with a clock of 17.5 MHz, the precounter should reset one time at 17, and the other time at 16. This does not result in an incrementation interval of 1 microsecond, but the incrementation speed is exactly 1 MHz. I did not use this technique, but it is recommendable.

The 12 bits output ‘CL’ carries three reset values for the precounters of the links, one value for each link speed:
- bits 0..3 reset value for 5 Mbits/s
- bits 4..7 reset value for 10 Mbits/s
- bits 8..11 reset value for 20 Mbits/s

These reset values are dependent on input ‘PS’ according to table V. This table is also implemented as a rom, named ‘CLROM’.

### Table V Reset values for link speeds

<table>
<thead>
<tr>
<th>‘PS’</th>
<th>clock freq.</th>
<th>5 Mbits/s</th>
<th>10 Mbits/s</th>
<th>20 Mbits/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20.0 MHz</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>22.5 MHz</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>25.0 MHz</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>30.0 MHz</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>35.0 MHz</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>invalid</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>6</td>
<td>17.5 MHz</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>invalid</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
</tr>
</tbody>
</table>

This also does not always result in exactly the wanted link speed. In table VI the real link speeds are calculated. Notice that only at internal clock frequency 20 MHz all link speeds can be simulated exactly. In this case partial frequency control is not recommendable. The clock of the INMOS links are asynchronous with the processor clock. This cannot be simulated in this IDaSS version, so I do not recommend to spend a lot of hardware on this task.
### Table VI Real link speeds

<table>
<thead>
<tr>
<th>'PS'</th>
<th>clock freq.</th>
<th>real link speeds versus wanted link speeds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>5 Mbits/s</td>
</tr>
<tr>
<td>0</td>
<td>20.0 MHz</td>
<td>5 Mbits/s</td>
</tr>
<tr>
<td>1</td>
<td>22.5 MHz</td>
<td>4.5 Mbits/s</td>
</tr>
<tr>
<td>2</td>
<td>25.0 MHz</td>
<td>5 Mbits/s</td>
</tr>
<tr>
<td>3</td>
<td>30.0 MHz</td>
<td>5 Mbits/s</td>
</tr>
<tr>
<td>4</td>
<td>35.0 MHz</td>
<td>5 Mbits/s</td>
</tr>
<tr>
<td>5</td>
<td>invalid</td>
<td>unknown</td>
</tr>
<tr>
<td>6</td>
<td>17.5 MHz</td>
<td>4.38 Mbits/s</td>
</tr>
<tr>
<td>7</td>
<td>invalid</td>
<td>unknown</td>
</tr>
</tbody>
</table>

The error flag of the CPU at input 'Ei' is simply connected with output 'E' of the transputer. The connectors 'Ana' and 'BFR' are not shown in figure 18, because their functionality is not implemented.
10. IMPLEMENTATION OF THE EXTERNAL MEMORY INTERFACE

This implementation of the external memory interface is a static RAM and ROM interface with DMA control. It should support dynamic RAM as well, but I have not implemented this functionality. Such a memory interface has already been implemented in IDaSS by E.B.C. Daniëls. Besides, this is not necessary logic to simulate the typical features of a transputer. This chapter describes the implementation of a static memory interface with DMA control.

Figure 19 shows the schematic of the implementation of the external memory interface.

Normally the two memory interfaces are simply connected, comprising a static memory interface. But if a DMA request takes input 'DReq' high, operator 'DMA' waits for a moment that it can interrupt the communication between the external memory and the transputer. For simplicity it waits until the control bus 'c' becomes zero. Then output 'DGra' becomes high to give a DMA acknowledge (assuming that the DMA request is still high). This signals the buffers 'aB' and 'cB' to disable their outputs and operator 'hC' holds the handshake inside the transputer low. Register 'rR' stores the DMA acknowledge to retain that a DMA request is in service.
11. IMPLEMENTATION OF THE TIMERS

The T800 contains two 32 bit timers. Timer 0 increments every microsecond and timer 1 every 64 microseconds. These timers are able to schedule processes that are waiting for a certain time. High priority processes can only wait for timer 0 and low priority processes for timer 1. Timer 0 also gives a time slice signal every 1024 increments. The executor will switch low priority processes after two time slices.

The implementation of these timers is shown in figure 20.

![Timers Diagram](image)

The Z-bus and U-bus are connected with both timers. Operator ‘ba’ defines the block addresses of timer 0 and 1. Operator ‘rO’ just concatenates the schedule requests ‘rT0’ and ‘rT1’ to output ‘rT’, with ‘rT0’ the least significant bit. Input ‘CT’ carries the reset value for the precounter of timer 0. This enables incrementation of timer 0 every microsecond independent on the internal clock frequency. Furthermore timer 0 drives the time slice output ‘ts’ high every 1024 increments. I made the incrementation rate of timer 1 dependent on timer 0, because timer 0 is already synchronised. Every time timer 0 increments, ‘P0’ gives a signal to timer 1. This signal is precounted 64 times. This should lead to incrementation of timer 1 every 64 microseconds.

11.1 Timer 0

Figure 21 represents the implementation of timer 0.
The executer and the scheduler can read and write to the workspace registers 'WR', the clock register 'CR' and the time out register 'TR'. 'WR' has register address 0, 'CR' has address 1 and 'TR' has address 2.

The precount register 'PR' increments every clock cycle. Operator 'PO' resets this register if its value equals the value at input 'CT'. This reset signal also goes to operator 'TO' and to timer 1 via connector 'PO'. Operator 'TO' increments register 'CR' every time the precounter resets. If 'CR' equals 'TR' and 'WR' does not equal $80000000_{16}$ (indicating that a process is waiting for a time), then operator 'TO' sets the schedule request register 'rR'. When the scheduler reads the workspace register via the Z-bus, operator 'TO' resets the request register.

If the first 9 bits of register 'CR' are zero, then operator 'TO' drives the time slice output 'ts' high for one clock cycle. This happens every 1024 increments of timer 0.

11.2 Timer 1

Timer 1 has about the same implementation as timer 0 (see figure 22).
The time slice output disappeared and the precounting is different from timer 0. The precount register 'PR' does not increment every clock cycle, but only when input 'PO' (from timer 0) becomes high. This should occur every microsecond. If the value of register 'PR' is zero, then operator 'PO' drives incrementation output 'i' high for one clock cycle. Operator 'TO' will increment the clock register 'CR' at this signal. Because the precount register is 6 bits wide, timer 1 increments every 64 ticks of timer 0.
12. IMPLEMENTATION OF THE LINKS

The links are the most characteristic of transputers and therefore the most interesting to implement.

The block 'links' of the T800 contains the four link interfaces, link control and a bus multiplexer as depicted in figure 23.

The operator 'rO' just concatenates the schedule requests of the link interfaces to one output 'rL'.
12.1 The bus multiplexer

Each link interface has one memory interface. These are multiplexed to one memory interface. The implementation of this bus multiplexer is represented in figure 24. Actually this bus multiplexer is a simplified version of the bus multiplexer ‘mux4x2’.

No address selection has to be performed, because there is only one memory interface for all link interfaces. The multiplex units ‘MUX0’ through ‘MUX3’ are the same as those in the bus multiplexer ‘mux4x2’, except for holding the handshake output ‘hp’ low as long as control input ‘cp’ is low. The arbiter ‘ARB’ is exactly the same. So it uses fixed priority in order of request number: (from high to low) ‘req3’, ‘req2’, ‘req1’ and ‘req0’. This means that link interface 3 has highest priority and link interface 0 the lowest priority. As mentioned before this should not cause starvation. If someone could prove starvation, then the solution is simple: use rotating priority.
12.2 Link control

This block controls the speed of all link interfaces. The implementation is shown in figure 25.

![Figure 25 Link control](image)

The speed of each link can be selected with the input 'LS'. See paragraph 5.1 for the definition of this input. Input 'CL' carries three reset values, one for each link speeds. See chapter 9 for the definition of this input. Operator 'LSO' selects a reset value for each link and places them at the outputs 'r0' through 'r3'. It selects one of the reset values at input 'CL' dependent on the link speed selection input 'LS'. The precounter registers 'C0' through 'C3' increment by default. Operators 'O0' through 'O3' are the comparators. They drive the link control output 'LC' high when the precounter register equals the reset value. This signal causes a link interface to input or output the next bit. Of course this signal also resets the precounter register.

12.3 Link interface

The four link interfaces of the transputer are exactly the same, so it will be sufficient to describe one link interface. The implementation of a link interface is shown in figure 26.
The link interface is divided into the three blocks 'Out', 'In' and 'mux2x1'. Block 'Out' serves an output process and block 'In' serves an input process. The block 'mux2x1' is a bus multiplexer to make the input process and the output process share one memory interface. The block addresses are defined by operator 'ba'. Operator 'ro' just concatenates the schedule requests 'rLO' and 'rLI', with 'rLI' at the least significant position.

The 'Out' block is able to transmit a message serially on the link output 'LO'. Likewise the 'In' block is able to receive a message serially on the link input 'LI'. The link control input 'LC' signals both to input and to output the next bit. The input and output block handshake with the signals 'Tack', 'Tar' and 'Rar'. The input block requests the output block to transmit an acknowledge by setting signal 'Tack'. The output block returns the 'Rar' signal when it is ready sending the acknowledge. When the input block received an acknowledge it reports this to the output block via signal 'Rar'.

12.3.1 The bus multiplexer

The bus multiplexer 'mux2x1' is not a full bus multiplexer, because it is dedicated to one read only interface and one write only interface. The output process (interface 0) only reads data from memory and the input process (interface 1) only writes data to memory. This makes it possible to simplify the bus multiplexer. The implementation of this block is depicted in figure 27.
Operator 'ARB' is a fixed priority arbiter. It gives interface 1 higher priority than interface 0. So the input process has higher priority than the output process. From control input 'c0' only the read request is accepted and from control input 'c1' only the write requests are accepted.

The status register 'sR' keeps up the following information:
- bit 0: 1 = serving a request, 0 = no serving
- bit 1: number of the interface that is in service.

The handshake signal resets this register.

Operator 'MUX' uses the status output of the arbiter to switch the signals. The address output 'a' and the control output 'c' follow the corresponding inputs of the interface in service. When there is no request the control output 'c' still equals input 'c1', in order to hold the control bus inactive. Likewise the address output 'a' follows input 'a1' by default. Handshake output 'h0' only goes high when interface 0 is in service and the handshake input 'h' goes high. Handshake output 'h1' is controlled likewise. Data output 'd0' always follows data input 'di' and is always enabled, because interface 0 can only read from memory. Data output 'd1' always equals data input 'd1', but is only enabled when interface 1 is in service.
12.3.2 Link output

Figure 28 shows the implementation of the link output block ‘Out’.

The block ‘OC’ is able to send data byte packets and acknowledge packets serially. The rest of this schematic reads data words from memory, handles Z-bus communication and generates the schedule request ‘rLO’.

The 32 bits registers ‘WR’, ‘AR’ and ‘CR’ hold respectively the workspace descriptor, the number of bytes to transmit and the message pointer. Operator ‘ZO’ can write the value from the Z-bus into these registers and operator ‘ZI’ can read these registers, if addressed with the U-bus. Register ‘AR’ is decremented and register ‘CR’ is incremented every time a data byte is transmitted.

If the byte selector of ‘CR’ overflows, operator ‘DO’ sets register ‘rR’. This means that
12. IMPLEMENTATION OF THE LINKS

reading of another data word from memory is needed. Then operator 'RO' sets the read
bit at the control bus. Operator 'aO' outputs the word pointer of the message pointer 'CR'
at the address bus. The handshake input 'h' loads the data bus value into register 'DR' and
reset register 'rR' in order to end the reading.

Operator 'TO' selects one byte of the data word in register 'DR' dependent on the byte
selector of the message pointer 'CR'. The selected byte is handed over to the transmit
block 'OC' at output 'Td' with the handshake signal 'Treq'.

If the transmitting of the data byte is ready, the block 'OC' reports this by setting the
handshake signal 'Tdr'. Operator 'SO' stores this signal in the 2 bits status register 'SR',
together with the receive acknowledge ready signal 'Rar' from the link input block 'In'.
Because these signals are pulsed signals (high during one clock cycle), operator 'SO' holds
these status bits high when set once. The control output 'c' of operator 'CO' resets the
status register.

This schematic does not contain a state machine controller, but operator 'CO' performs
a comparable task. Register 'TR' holds the state number and the 8 bits control output 'c'
controls nearly all operators in this schematic. It carries the following information:

- bit 0..1  'T': register 'TR': state number
- bit 2  'Tdr': bit 0 of register 'SR'
- bit 3  'Rar': bit 1 of register 'SR'
- bit 4  'r': register 'rR': reading from memory
- bit 5  'A1': register 'AR' > 1: more than one byte to transmit
- bit 6  'A0': register 'AR' = 0: no bytes to transmit
- bit 7  'W8': register 'WR' = 80000000: no workspace descriptor.

The state numbers represent:
- 0  'Standby': no output process is active
- 1  'Read': reading a data word from memory
- 2  'Transmit': transmitting data bytes and receiving acknowledges.

State transitions are graphically depicted in a state transition diagram in figure 29.
The reset value of register 'TR' is zero, so the startup state is 'Standby'. As soon as register 'WR' contains a workspace descriptor and 'AR' is greater than zero, operator 'DO' will set register 'rR' and the next state is 'Read'. In this state is waited until the reading is finished by resetting register 'rR'. Then operator 'TO' transmits the first data byte and the next state is 'Transmit'. Transmission of a data byte only ends when both 'Tdr' and 'Rar' are high. Notice that it is possible that the acknowledge has already been received before the data byte is completely transmitted, in case of an overlapped acknowledge. Every time transmission of a data byte ends, register 'SR' resets, operator 'ZO' decrements 'AR' and increments 'CR', and operator 'DO' sets register 'rR' when the word selector of 'CR' equals 3 and 'AR' > 1 (before decremented). If this is the case a state transition to 'Read' occurs, in order to wait for the reading before the next data byte can be transmitted.

Operator 'CO' sets the request output 'rLO' when 'AR' = 0, 'WR' holds a workspace descriptor, 'Tdr' = 1, 'Rar' = 1 and the state is 'Transmit'. The scheduler will answer this request by reading the workspace descriptor via the Z-bus. Operator 'ZO' will then automatically reset register 'WR' to 80000000₁₆. This will remove the request and cause a state transition to 'Standby'.

Output control

The block 'OC' performs the task of transmitting data and acknowledge packets serially. Its implementation is shown in figure 30.
If the transmit data request input 'Treq' goes high, register 'dR' loads the data byte from input 'Td'. Because this request is a pulsed signal, it is kept in register 'rR'. In the same way the transmit acknowledge request 'Tack' is kept in register 'aR'. Register 'cR' counts the bit position in the transmitted packet including the start and stop bits. As long as the link control input 'LC' is low, output 'LO' equals its previous value, loaded in register 'OR'. If input 'LC' goes high, then the next bit is transmitted at output 'LO' and the counter is incremented.

If a data or acknowledge request is received at counter = 0, a start bit is transmitted and the counter is incremented. Notice that at this point there is no difference between a data packet and an acknowledge packet. If the acknowledge request is set at counter = 1, a stop bit is transmitted and the acknowledge ready output 'Tar' goes high. Else a one bit is transmitted to identify a data packet. At the following counter values a data bit of register 'dR' is transmitted. The value counter - 2 is the number of the selected bit. In this way the least significant bit is transmitted first, as defined by INMOS. At position 10 a stop bit is transmitted, the counter resets and the data ready output 'Tdr' goes high.

12.3.3 Link input

Figure 31 shows the implementation of the link input block 'In'.
Figure 31 Link input

This schematic looks like the link output, but is different at some points. The block ‘IC’ is able to receive data byte packets and acknowledge packets serially. Output ‘Rd’ carries a received data byte if the data ready output ‘Rdr’ goes high. When an acknowledge is received output ‘Rar’ goes high. In order to make overlapped acknowledge possible, output ‘Rdd’ goes high as soon as the received packet is identified as a data packet.

Operator ‘RO’ merges the received byte into the 32 bits register ‘dR’ at the byte position equal to the byte selector of ‘CR’. Operator ‘RO’ also sets the corresponding write bit in the 4 bits register ‘bR’. Operator ‘ZO’ decrements ‘AR’ and increments ‘ZO’ each time a data byte is received. If the byte selector of ‘CR’ overflows, operator ‘DO’ will set register ‘rR’. This makes operator ‘WO’ drive the control bus with the write bits in register ‘bR’. Operator ‘aO’ drives the address bus with the word pointer of ‘CR’. The handshake input ‘h’ resets the write bits in register ‘bR’ and resets register ‘rR’ in order to end the writing.

Operator ‘SO’ stores three signals in status register ‘SR’:
• bit 0  'Rdr': receive data ready
• bit 1  'Rdd': receive data detected
• bit 2  'Tar': transmit acknowledge ready.

Because these signals are pulsed signals (high during one clock cycle), operator 'SO' holds these status bits high when set once. It resets the status when a new data byte is detected, that is when 'Rdd' goes high, while bit 1 of the status register is already set.

The link input also does not contain a state machine controller and even no state register, because it is always in the same state. Nevertheless operator 'CO' has control of nearly all operators in this schematic with its 6 bits control output 'c'. This output carries the following information:

• bit 0..2  'S': register 'SR': status information
• bit 3  'A1': register 'AR' = 1: one byte to receive
• bit 4  'A0': register 'AR' = 0: no bytes to receive
• bit 5  'W8': register 'WR' = 80000000₁₆: no workspace descriptor.

Operator 'CO' requests for transmitting an acknowledge (set 'Tack'), as soon as the receiving of a data packet is detected ('Rdd' high) and of course 'WR' must hold a workspace descriptor and 'AR' not zero. In this way an overlapped acknowledges is generated. The transmit acknowledge request is removed as soon as bit 2 ('Tar') of the status register is set.

Operator 'CO' requests for scheduling the input process (set 'rLI') when 'AR' equals zero and 'WR' holds a workspace descriptor. This implies that to setup an input process the number of bytes must be written before the workspace descriptor. The scheduler will answer the schedule request by reading the workspace register 'WR'. This reading also resets 'WR' to 80000000₁₆, what automatically removes the schedule request.

Operator 'ZO' allows writing via the Z-bus to the registers 'WR', 'AR' and 'CR' as long as register 'WR' equals 80000000₁₆. In all other situations an input process has control of these registers. Reading via the Z-bus is still possible through operator 'ZI'. Mind out that reading the workspace register will reset it and therefore stop the input process!

Operator 'RO' accept the received data byte only when a process is active and 'AR' is not zero. Otherwise it will hold the registers 'dR' and 'bR'. Only if all bits of the status register are set, the receiving of a data byte has completely finished. Each time a data byte is completely received, the status register resets, operator 'ZO' will decrement 'AR' and increment 'CR' and operator 'DO' will check if the byte selector of 'CR' overflows. Of course also a process must be active and 'AR' must not equal zero. If 'AR' equals one just before decremented, the last data byte has been received. Then operator 'DO' must always set register 'rR', in order to write data register 'dR' for the last time.
Input control

Block 'IC' performs the task of receiving data and acknowledge packets serially. Its implementation is shown in figure 32.

Figure 32 Block 'IC'

Register 'cR' counts the bit position in the received packet. The 8 bits data register 'dR' saves the received data bits. While link control input 'LC' is low, nothing happens. That means operator 'IC' holds both registers and gives no signals. The actions described below presume that 'LC' is high.

If input 'LI' goes high while the counter is zero, then the counter is incremented for the first time. Afterwards the counter increments always, if not overruled by a reset. The first bit is a start bit and the second bit can be a one to identify a data packet or a zero stop bit of an acknowledge packet. If a zero bit is received while the counter is one, the packet was an acknowledge. Then the receive acknowledge ready output 'Rar' goes high and the counter resets. But if a one bit is received while the counter was one, the packet was a data packet. Then the receive data detected output 'Rdd' goes high and the counter increments. If the counter value lies between one and ten, the received bit at input 'LI' is shifted into register 'dR'. It is shifted right, because the least significant bit is transferred first. If the counter reaches 10, the receive data ready output 'Rdr' goes high and the counter resets.
13. IMPLEMENTATION OF THE EVENT CHANNEL

The event channel provides a handshake interface between an external event and an internal process. No message input or output is performed. One process can wait until an external event occurs.

Figure 33 shows the implementation of this event channel.

![Figure 33 Event channel](image)

Operator 'CO' performs all tasks of the event channel, like Z-bus communication, schedule request and event acknowledge. Operator 'ba' defines the block address of the event channel. The workspace register 'WR' has register address 0. Three-state buffer 'ZB' enables output the value of register 'WR' at the Z-bus. The registers 'SR', 'AR' and 'RR' load respectively the schedule request output 'rE', the event acknowledge output 'EAck' and the event request input 'EReq', defining the status of the event channel.

Two distinct situations are possible: The event can occur before or after a process is waiting. Let’s describe the first situation first, that is the event occurs before a process is waiting. When input 'EReq' goes high while the register 'WR' holds $80000000_{16}$, then the event channel changes this register to $80000001_{16}$ indicating that it is ready. Before the executer deschedules a process it will read the workspace register via the Z-bus. Then the executer knows that the event already occurred, so the process continues immediately without descheduling. The reading via the Z-bus will reset the workspace register, so the event channel is ready for another event. Additionally operator 'CO' sets the acknowledge output 'EAck', indicating to the outside world that an internal process has noticed the external event. Then the external logic should remove the event request. After 'EReq' goes low the acknowledge is removed.

The second situation is that the event occurs after or together with the descheduling of a process that waits for an event. The executer reads the workspace register via the Z-bus, which is $80000000_{16}$ indicating that no event has already occurred. So the executer deschedules the process, that is it writes the workspace descriptor of the process (not equal to $80000000_{16}$) in the workspace register 'WR' of the event channel via the Z-bus. If the event request input 'EReq' goes high at this point, the acknowledge output 'EAck' and the schedule request output output 'rE' go high both. The scheduler will read the workspace
register 'WR' via the Z-bus, which resets this register to $8000000_{16}$ and removes the schedule request. The acknowledge is removed after 'EReq' goes low.
14. IMPLEMENTATION OF THE CPU

The CPU is the heart of the transputer and the most complicated part. It fetches, decodes and executes instructions, loads and stores data, schedules processes and communicates with other blocks in the transputer via the Z-bus.

The schematic of the implementation of the CPU is shown in figure 34.


The CPU has three memory interfaces, one for the fetcher, one for the load/store unit (via the transparent workspace cache) and one for the process scheduler. The scheduler receives schedule requests from the timers, links and event channel through the inputs ‘rT’, ‘rL’ and ‘rE’. The Z-bus protocol uses the connectors ‘U’ and ‘Z’. Output ‘E’ carries the error flag of the executer. Timer 0 drives the time slice input ‘ts’ for switching low priority processes. The executer can request the scheduler with the ‘nEx’ signal for giving the next ready process. With the ‘rEx’ signal the executer requests the scheduler for scheduling a process, just like a link interface, timer and event channel. When a high priority process is scheduled while the current process has low priority, the scheduler interrupts the executer by making output ‘int’ high.

14.1 The fetcher

The fetcher will free the executer from fetching instructions. The executer controls the 32
bits instruction pointer at input 'Ip'. The fetcher will return four instructions at the 32 bits output 'Ir' with a valid bit at output 'v'. Because the memory is word aligned, reading from memory always fetches four instructions together. The fetcher does not arrange the instructions. It just passes on all four instructions pointed to by the word pointer of the instruction pointer (the upmost 30 bits). The decoder uses the byte selector of the instruction pointer (the lower 2 bits) to start decoding at the right instruction. The implementation of the fetcher is shown in figure 35.

Figure 35 The fetcher

This fetcher always attempts to prefetch 8 instructions. That means 4 instructions pointed to by the word pointer of the instruction pointer and the next 4 instructions. Therefore it uses two sets of registers. Each set consists of a valid register 'v1R'/v2R', a 32 bits instruction register 'I1R'/I2R' and a 30 bits instruction pointer register 'P1R'/P2R', that only contains a word pointer. Register 'sR' keeps up the status of the fetcher: 0 means no fetching, 1 means fetching for set 1 and 2 means fetching for set 2.

When the word pointer at input 'Ip' does not equal both 'P1R' and 'P2R' or both sets are invalid, then the fetcher starts reading from address 'Ip'. This address is stored in register 'aR', because the value at input 'Ip' could change, while the address output 'a' must remain constant during the read cycle.

Only after the handshake input 'h' goes high, a selection is made were to store the received instructions at input 'd'. If the address register 'aR' equals the word pointer of 'Ip', then the instruction are stored in register 'I1R'. If 'aR' equals the word pointer 'Ip' + 1, then the instructions are stored in register 'I2R'. Otherwise the instructions are thrown away, because 'Ip' has changed to much (when jumping for example).
It is also possible that set 1 does not contain the asked instructions but set 2 does. Then the fetcher shifts set 2 to set 1 in one clock cycle.

14.2 The decoder

This decoder translates at maximum four instructions at the 32 bits input ‘Ir’. Instructions are translated to one 4 bits function (output ‘f’) and a 32 bits operand (output ‘O’). The executer executes these functions and changes the instruction pointer in the last clock cycle of the execution. The translation starts with the instruction pointed to by the byte selector (lowest 2 bits) of the 32 bits instruction pointer input ‘Ip’. The implementation of the decoder is depicted in figure 36.

This decoder is able to execute ‘pfix’ and ‘nfix’ instructions. A series of instructions starts with zero or more ‘pfix’ and/or ‘nfix’ instructions and ends with one other instruction. The number of instructions in a series is minimal 1 and maximal 8. The decoder will asynchronously calculate the new contents of the operand register and the function of the last instruction. If not all instructions of a series are available at the input ‘Ir’, then the decoder will calculate the operand as far as possible and output as function the last available ‘pfix’ or ‘nfix’ instruction. The executer will move the instruction pointer, so that the fetcher will deliver the next four instructions at input ‘Ir’. Then the decoder can finish the series. For that reason the decoder saves the operand of the unfinished series in register ‘Oreg’.

The decoder also calculates the next instruction pointer (32 bits output ‘nI’). This pointer equals the old instruction pointer (‘Ip’) incremented with the number of instructions in a series that are available at input ‘Ir’. This number can be 1 through 4. The executer loads this pointer in the instruction pointer register in the last clock cycle of the execution and if the instruction does not perform a jump.
If input 'v' is low, indicating invalid instructions at input 'Ir', the decoder will output 'pfix' as the function and will not increment the next instruction pointer. This causes the executer to hold.

The executer controls the function of the decoder via the 2 bits input 'Ie':

- %00  Hold.
- %01  Decode.
- %10  Reset.

When the execution of an instruction takes more than one clock cycle, the executer puts the decoder in the 'Hold' function. Then the decoder holds its operand and does not increment the instruction pointer. If you would leave the decoder in the 'Decode' function, the data bits would shift in the operand register more than once.

When the execution becomes ready in the next clock cycle, the executer puts the decoder in the 'Decode' function. Then the decoder will decode the next instructions, which the executer will load in a register.

When the execution performs a jump, the executer will put the decoder in the 'Reset' function. Then the decoder will reset the operand, hold the instruction pointer and output a 'pfix' function. This function is important, because it performs the handshaking to hold the executer.

### 14.3 The load/store unit

This block frees the executer from waiting for a store action to become ready. The load/store unit saves the address, data and control information in registers and outputs this information at the memory interface until the memory gives a handshake. The executer can go on while the load/store unit performs the write cycle. The executer must wait until the busy output 'b' is low, before performing another read or write.

The load/store unit cannot free the executer from waiting for a load action to become ready, because the executer might need the data in the next instruction. Nevertheless the load/store unit stores the address and control information in registers and outputs this information to the memory until the memory gives a handshake. The handshake output 'h' loads data in one of the registers of the executer.

The implementation of the load/store unit is shown in figure 37.
Register ‘cR’ loads the control signal from the executer. Operator ‘hO’ resets this register when the handshake input ‘h’ goes high. Because of that register ‘cR’ always holds zero in the first clock cycle of a read or write cycle. Operator ‘cO’ compresses both 5 bits control signals ‘ci’ and ‘cR’ to one 4 bits output ‘s’:

- bit 0: the read bit of input ‘ci’ is set,
- bit 1: at least one write bit of input ‘ci’ is set,
- bit 2: the read bit of register ‘cR’ is set,
- bit 3: at least one write bit of register ‘cR’ is set.

This signal is used by all other block in this schematic. In the first clock cycle of a read or write cycle register ‘aR’ loads the address input ‘ai’. In the next clock cycles of a read or write cycle register ‘aR’ drives the address bus. Likewise register ‘dR’ loads the data in the first clock cycle of a write cycle and drives the data bus in the other clock cycles of a write cycle. The executer is not allowed to drive the data bus when its control signal ‘ci’ does not contain a write bit. Operator ‘cO’ is the only driver of the control bus, to be sure that this bus continuously carries a defined value. When register ‘cR’ holds any read or write bits, this value is passed on, otherwise the control input ‘ci’ is passed on. Likewise the executer must continuously define the control input ‘ci’. Operator ‘hO’ drives output ‘b’ high during a read or write cycle, but not in the last clock cycle (the handshake signal is high in the last clock cycle). This signal is piped through a register to reduce the asynchronous block length. The output of this register is the busy signal for the executer. This signal goes high if the load/store unit is busy performing a read or write cycle, that is not in the first clock cycle. The executer is not allowed to drive the address or data bus as long as the busy signal is high, because then the load/store unit drives these buses.
14.4 The workspace cache

Because the processor has little working registers, many times local variables in the workspace have to be loaded and stored. I designed a cache for the first 32 words in the workspace of the current process. To keep this cache simple, it cannot store partial words. Only if all write strobes are set the whole word is stored in the cache memory with one valid bit. Caching partial words would increase the number of hardware components a lot, but will not increase the performance a lot. This cache is transparent, meaning that it can be placed anywhere in the processor-memory chain and can be left out without any problems. The implementation of this cache is depicted in figure 38.

Register ‘WR’ continuously loads the workspace pointer at input ‘W’, in order to detect changes of the workspace pointer from the executer. The data is stored in the cache memory ‘CM’. This block is an IDaSS RAM for simulation, but for synthesis this block should be replaced by a register file of 32 registers of 32 bits wide each with address logic. The 32 bits register ‘vR’ keeps up the valid bits of each data word in the cache memory. When the address lies in the range between ‘WR’ and ‘WR’ + 31 during a read or write cycle, the data is stored in the cache RAM and the accompanying valid bit is set. This block passes on every write to memory in order to keep the memory and the cache consistent. A read from memory is only passed on if the cache does not contain the data. If the cache contains the valid data, it will output this data at the data bus and give a handshake signal at output ‘h’. The control output to the memory ‘co’ stays inactive, so the memory will not notice this read action. This reading from cache will take place in only one clock cycle as if the data comes from a register.
If the workspace pointer at input ‘W’ does not equal register ‘WR’, the cache will mark all 32 words invalid by resetting register ‘vR’. This even occurs when the change of the workspace pointer lies within the cache range. The alternative is to shift data which remains valid, but this will cost a lot of hardware and clock cycles. Besides small changes to the workspace pointer happens rarely. Normally the workspace pointer changes due to a process switch, generally causing a large movement.

14.5 The Z-bus arbiter

Because the executer and the scheduler can both be master of the Z-bus, they must first request for the Z-bus at the arbiter. Both masters request by setting the most significant bit of the 9 bits input ‘r0’ or ‘r1’. The arbiter acknowledges one of the requests with a high output ‘a0’ or ‘a1’. To be sure that the U-bus always carries a defined value, there is only one driver: output ‘U’ of the Z-bus arbiter. Both masters send their wanted U-bus value to the arbiter via the lowest 8 bits of the inputs ‘r0’ and ‘r1’. The arbiters passes on the appropriate value, so the master that owns the Z-bus determines the value at the U-bus. If no request is submitted, then the arbiter will output a value at the U-bus that addresses no block at all, to bring the bus in rest state.

Because a master owns the Z-bus for only one clock cycle, the arbiter does not have to retain the request number that is in service. Every clock cycle there is new arbitration of the Z-bus.

The arbiter uses fixed priority. The scheduler has higher priority than the executer, because the scheduler must provide the executer with processes. The scheduler serves requests for scheduling and switching processes. Serving a request takes only one clock cycle for Z-bus communication and several clock cycles for several memory read/write cycles. While the scheduler performs these memory read/write cycles, the executer can become master of the Z-bus. So starvation of the executer requests is impossible.

14.6 The scheduler

The process scheduler is able to append processes to the tail of the process queues (schedule) and to extract processes from the head of the process queues. In order to schedule a process this block reads the workspace pointer from the requesting block via the Z-bus.

The scheduler also has the task to update the timer queues. This means that if it schedules
a timer process, it additionally places the next timer process from the head of the sorted
timer queue into the timer and moves the head pointer of the timer queue. For clearness,
not the scheduler but the executer does the tedious job of searching through the sorted
timer queue, to insert a new timer process at the right place.

The implementation of the scheduler is depicted in figure 39. The updating of the timer
queues is regrettably not implemented yet. Therefore two registers should be added, to
hold the head pointers of the timer queues. Notice that the executer can also change the
timer queues, when it inserts a new timer processes. This could go wrong, when both the
executer and the scheduler change the same timer queue at the same moment. A solution
could be to inform the executer which changes the scheduler makes.

Figure 39 Scheduler

Registers ‘FOR’, ‘BOR’, ‘F1R’ and ‘B1R’ hold the front and back pointer of the high and low
priority process queues, equivalent to respectively ‘Fptr0’, ‘Bptr0’, ‘Fptr1’ and ‘Bptr1’.
These registers are only 30 bits wide, because they hold word pointers. Register ‘rR’ holds
the request number which is in service. Register ‘PR’ holds the priority of the process
currently in execution. Register ‘WR’ loads data from the Z-bus when the acknowledge
input ‘ack’ goes high. Operator ‘ba’ defines the block address of the scheduler. Register
‘IR’ drives the interrupt output to the executer. Operator ‘PO’ loads the pointer registers.
Implementation of the CPU

Operator 'ZO' outputs one of the pointers at the Z-bus. Operator 'DO' controls memory access. Operator 'ARB' arbitrates the incoming requests. The state machine controller 'TS' has control of all operators and registers in this schematic. Because in IDaSS a state machine controller cannot read bus values, but can read register values, register 'aR' loads the acknowledge signal and register 'hR' loads the handshake signal.

14.6.1 Arbitration of the requests

Operator 'ARB' loads a request number in register 'rR'. As long as a request is in service this register will be in the hold function, to disable new requests. Arbiter 'ARB' uses fixed priority to select a request. The priority is in order of request number, where the lowest number has the highest priority (see table VII).

<table>
<thead>
<tr>
<th>request number</th>
<th>input name with bit number</th>
<th>block name</th>
<th>purpose of request</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(zero)</td>
<td>(no block)</td>
<td>(no request)</td>
</tr>
<tr>
<td>1</td>
<td>'rEx'</td>
<td>executer</td>
<td>scheduling</td>
</tr>
<tr>
<td>2</td>
<td>(zero)</td>
<td>scheduler</td>
<td>(no request)</td>
</tr>
<tr>
<td>3</td>
<td>'rE'</td>
<td>event</td>
<td>scheduling</td>
</tr>
<tr>
<td>4</td>
<td>'rT' 0</td>
<td>timer 0</td>
<td>scheduling</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>timer 1</td>
<td>scheduling</td>
</tr>
<tr>
<td>6</td>
<td>'rL' 0</td>
<td>link 0 input</td>
<td>scheduling</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>link 0 output</td>
<td>scheduling</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>link 1 input</td>
<td>scheduling</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>link 1 output</td>
<td>scheduling</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>link 2 input</td>
<td>scheduling</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>link 2 output</td>
<td>scheduling</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>link 3 input</td>
<td>scheduling</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>link 2 output</td>
<td>scheduling</td>
</tr>
<tr>
<td>14</td>
<td>'nEx'</td>
<td>executer</td>
<td>extracting</td>
</tr>
</tbody>
</table>

In order to equal the request number to the block address, two zero requests are added.
Address 2 is the scheduler, which cannot request itself. Address 0 is not a block address, but is used as the rest value of the U-bus. If no request is submitted the arbiter will give 15 as the request number. To wait for a request, the state machine controller will put register ‘rR’ in the load function as long as its value is 15. This register is in the hold function by default, so the request number cannot change as long as it is in service.

The scheduler must first serve all schedule requests, before extracting a process for the executer. The executer cannot generate new processes, before it has extracted a process. Only the scheduler can generate a new timer process, every time a timer process is scheduled. This could take a while if many processes wait for the same time. But it makes sense that these processes are scheduled first. Because in practice the number of timer processes is finite, starvation of the extract request is impossible. Starvation of the schedule requests is also impossible, because they all need the extract request to start a new process. So before a process is switched, all schedule requests must first been served.

14.6.2 State control

The state machine controller can be in one of the three states ‘NextRequest’, ‘Load’ and ‘Store’. I have described the three states in flow diagrams, depicted in figures 40, 41 and 42. The ovals reflect the states. A state transition only happens when the clock performed a cycle.
Figure 40 Flow diagram of state 'NextRequest'
14.6.3 Scheduling a process

All requests, except for request 14, are schedule requests. First operator 'UO' requests for reading via the Z-bus the workspace register of the requesting block. When the state machine controller detects that the acknowledge register 'aR' goes high, then register 'WR' has loaded the workspace descriptor in the previous clock cycle.

If the process to schedule has high priority and the current priority is low, then the
executor is interrupted by setting register ‘IR’. Remember that bit 0 of a workspace descriptor determines the priority of the process.

If the process queue is empty, then operator ‘PO’ loads both the front and back pointer register with the workspace pointer (function ‘WtoFB’). If the process queue is not empty, operator ‘DO’ stores the workspace descriptor at address back pointer - 2, in order to link the process to the end of the process queue. Operators ‘PO’ and ‘DO’ determine the priority level (from bit 0 of register ‘WR’) by itself and update only the corresponding process queue. At last the back pointer must move to this new end of the process queue. Therefore operator ‘PO’ loads the back pointer with the workspace pointer (function ‘WtoB’). The operator makes this action just after the handshake input goes high, to prevent the back pointer to change before the storing at address back pointer - 2 is finished.

14.6.4 Extracting a process

The executer can request for the next ready process from the process queues. The executer only needs the workspace descriptor of the new process. The scheduler can write this data via the Z-bus to the workspace register of the executer. The scheduler must give the first process of the high priority queue if there exists one. Otherwise the first low priority process is given.

First operator ‘UO’ requests for writing via the Z-bus to the workspace register of the executer. If the high priority queue is empty, then operator ‘ZO’ puts the low priority front pointer at the Z-bus and the current priority register ‘PR’ is set to low priority. Otherwise operator ‘ZO’ puts the high priority front pointer at the Z-bus and register ‘PR’ is set to high priority. When the acknowledge register ‘aR’ goes high, the writing is finished. Notice that register ‘PR’ contains the priority of the extracted process.

The interrupt register ‘IR’ is always reset, because this request could have been an answer to an interrupt. If there was no interrupt, the resetting does no harm.

At last the front pointer must be moved. If the extracted process was the last one (front pointer equals back pointer), the queue must be emptied by resetting the pointers to $80000000_{16}$ (the 30 bits word pointer is $20000000_{16}$). Otherwise operator ‘DO’ loads the workspace pointer of the next process from address front pointer - 2. Operator ‘PO’ loads this workspace pointer into the front pointer register (function ‘LoadnewF’). Operators ‘DO’ and ‘PO’ determine the priority level (from register ‘PR’) and update only the corresponding process queue.
14.7 The executer

The executer is the part of the transputer that actually executes instructions. It obtains instructions from the decoder, loads and stores data via the load/store unit and communicates with the scheduler, links, timers and event channel via the Z-bus. The implementation of this complicated block is printed in figure 43.
This executeer contains a lot of registers. The 4 bits function and 32 bits operand from the
decoder are loaded respectively in register 'fR' and 'OR'. Register 'SR' counts the state number. Register 'IR' holds the instruction pointer and register 'WR' the workspace descriptor. The three registers 'AR', 'BR' and 'CR' form the evaluation stack. The data registers 'diR' and 'doR' hold temporarily data. The four write bits are kept in register 'wrR'. The error flag and the halt-on-error flag are kept respectively in registers 'ER' and 'HR'.

Operator 'SO' increments the state number register 'SR' if the execution takes more clock cycles, and resets this register if the execution is in the last clock cycle. These three registers form a 12 bits instruction code:

- bit 0..7 lower 8 bits of the operand if 'fR' = F₁₆ (indirect instruction)
  \[ F₀₁₆ + 'fR' \] else (direct instruction).
- bit 8..11 state number 'SR'.

To reduce the size of the instruction code, the direct instructions are inserted into the operand code from \[ F₀₁₆ \] to \[ F₁₆ \]. This is allowed, because there is no indirect instruction with an operand code in this range.

Normally operator 'CO' outputs this instruction code at output 'c', which controls the function of all other operators in the executer. But state machine controller 'CT' can change the function of 'CO' to change the instruction code. This is necessary when an instruction prescribes execution of different functions dependent on some register contents. The state machine controller can easily read all registers in this schematic and take decisions to change execution. Operator 'SO' will load the changed state number into register 'SR', but the values of registers 'fR' and 'OR' do not change during execution of one instruction.

Sometimes execution must hold for some clock cycles, because of waiting for memory loading or storing or waiting for Z-bus communication. Therefore a second control bus is used. This is a copy of the first control bus, expanded with a hold bit:

- bit 0..11 instruction code
- bit 12 hold bit (1 = hold, 0 = continue).

Operator 'RW' drives this control bus with output 'hc' and sets the hold bit if loading or storing needs more clock cycles. Operator 'ZO' can also set this hold bit (via output 'hZ') if communication with the Z-bus does not finish in this clock cycle. The state register 'SR' will not increment, load or reset as long as the hold bit is set. Also all other registers hold their value. Therefore registers 'WR' and 'wrR', schematic 'ST' and operators 'SO', '10' and 'do' use this control bus. The values in the control specifications only define 8 or 12 bits, where the missing bits are assumed to be zero. If the hold bit is set no value equals the control bus, so the default function is executed. The default function is always 'Hold' or performs holding the registers.

Operator 'ALU' normally uses two 32 bits input 'X' and 'Y' to calculate the 32 bits output
14. IMPLEMENTATION OF THE CPU

'Z'. These two inputs can carry one of the register outputs. The operator 'XS' and 'YS' selects one of the buffers to drive the bus. The output 'Z' of the 'ALU' can drive the internal Z-bus, which is separated from the external Z-bus by buffers. The internal Z-bus is used to load a new value into one of the registers.

In order to be able to store data in memory in one clock cycle, the address must be calculated and the data must be selected from one of the registers at the same moment. The 'ALU' can calculate the address and drive the address bus with the Z-bus. But to drive the data bus, I added the D-bus. Operator 'DS' selects a buffer to drive this bus, just like the operators 'XS' and 'YS'. Here an extra operator 'MDB' can output the special value 80000000₁₀ ('Mint' = Minimum integer) at the D-bus. Additionally this bus is used when loading data from memory. Then operator 'DS' select no driver, but the data from memory is put at this bus. This data can be loaded in most of the registers. Without loading or storing, the D-bus can be used to exchange register values mutually.

Operator 'IO' can load the Z-bus or D-bus value to perform a jump. The workspace register 'WR' can only load from the Z-bus, but that has not been a restriction during my designing period. Schematic 'ST' can push the Z-bus or D-bus value on the three stack registers. Operator 'do' can load register 'diR' and 'doR' with the value at the internal Z-bus or D-bus. Only the 'ALU' can change the four write bits in register 'wrR' to write partial words. Normally these bits are all set. The error flag can be set by the 'ALU', but also the control connector can set and reset this flag directly. The halt-on-error flag can only be set and reset via the control connector.

Operator 'IO' loads the next instruction pointer 'nl' from the decoder if the state register resets (input 'Sc' = %11). This only occurs if the execution ends in this clock cycle. If a jump must be taken, this operator loads the instruction pointer register with the value of the Z-bus or D-bus and resets the decoder to prevent loading the wrong instruction. The decoder will output a prefix instruction and a zero operand. A prefix or negative prefix instruction will only cause operator 'IO' to load the next instruction pointer 'nl'.

14.7.1 State number and instruction control

Operator 'SO' is default in the 'Reset' function. But in this function the state number register 'SR' only resets if the hold bit of the control input is not set, else this register holds its value. To increment the state number, the control connector must put this operator in function 'Inc'. Likewise the state number holds in this function if the hold bit is set. Else not the register is incremented, but the state number from the control input is incremented. This is necessary to carry through jumps in the state number. The 2 bits output 'Sc' controls register 'SR' as follows: 0 = hold, 1 or 2 = load, 3 = reset. Because the default function is 'Reset', only the instruction codes that need incrementation of the state number must be included in the control specification.
Operator 'IO' controls the registers 'fR', 'OR' and 'IR' and the decoder with the 2 bits output 'Ie' according to table VIII.

<table>
<thead>
<tr>
<th>'Ic'</th>
<th>'fR'</th>
<th>'OR'</th>
<th>'IR'</th>
<th>decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hold</td>
<td>hold</td>
<td>hold</td>
<td>Hold</td>
</tr>
<tr>
<td>1</td>
<td>load</td>
<td>load</td>
<td>load</td>
<td>Decode</td>
</tr>
<tr>
<td>2</td>
<td>load</td>
<td>load</td>
<td>load</td>
<td>Reset</td>
</tr>
</tbody>
</table>

The default function of operator 'IO' is 'Next'. This function is smart enough to load the next instruction pointer only if the execution of the current instruction ends. That is output 'Ic' = 1 and 'IO' = 'nl' when input 'Sc' = 3. Else the registers and the decoder hold (output 'Ic' = 0). This operator only has to change function when a jump must be taken. For this purpose two functions are provided: 'JumpZ' and 'JumpD'. The function 'JumpZ' loads the value from the Z-bus into the instruction pointer register 'IR' and obviously 'JumpD' loads the value from the D-bus. Just before a new instruction pointer is loaded into register 'IR', it still contains the next instruction pointer and the decoder decodes these instructions. To prevent loading the wrong instruction the decoder is reset by making output 'Ic' = 3. The decoder resets the operand ('O' = 0) and outputs the function 'pfix' ('f' = 2). Only in the next clock cycle register 'IR' contains the new instruction pointer and the decoder is allowed to decode again.

14.7.2 Stack manipulations

The manipulate the stack I added a little schematic with a lot of buffers and an operator to control the registers and buffers (see figure 44).
Operator 'CO' has ten functions to manipulate the stack, listed in table IX. I have decided to hold a register if it should be left undefined according to INMOS.

Table IX Stack manipulation functions

<table>
<thead>
<tr>
<th>functions</th>
<th>'AR'</th>
<th>'BR'</th>
<th>'CR'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Hold'</td>
<td>hold</td>
<td>hold</td>
<td>hold</td>
</tr>
<tr>
<td>'PushZ'</td>
<td>load Z-bus</td>
<td>load 'AR'</td>
<td>load 'BR'</td>
</tr>
<tr>
<td>'PushD'</td>
<td>load D-bus</td>
<td>load 'AR'</td>
<td>load 'BR'</td>
</tr>
<tr>
<td>'Pop'</td>
<td>load 'BR'</td>
<td>load 'CR'</td>
<td>hold</td>
</tr>
<tr>
<td>'Pop2'</td>
<td>load 'CR'</td>
<td>hold</td>
<td>hold</td>
</tr>
<tr>
<td>'RepZ'</td>
<td>load Z-bus</td>
<td>hold</td>
<td>hold</td>
</tr>
<tr>
<td>'RepD'</td>
<td>load D-bus</td>
<td>hold</td>
<td>hold</td>
</tr>
<tr>
<td>'PopRepZ'</td>
<td>load Z-bus</td>
<td>load 'CR'</td>
<td>hold</td>
</tr>
<tr>
<td>'Reverse'</td>
<td>load 'BR'</td>
<td>load 'AR'</td>
<td>hold</td>
</tr>
<tr>
<td>'Dup'</td>
<td>hold</td>
<td>load 'AR'</td>
<td>load 'BR'</td>
</tr>
</tbody>
</table>
14.7.3 Loading and storing

Operator 'RW' has three functions: 'Hold', 'Load' and 'Store'. In the 'Hold' function output 'c' = 0 and the hold bit in output 'hc' follows input 'hZ' from operator 'ZO'. The 'Load' function waits until the busy input 'b' goes low. Then the read strobe at output 'c' is set and buffer 'diB' drives the internal D-bus. Operator 'DS' must have disable all buffers at this moment. The hold bit goes low only if both the busy input is low and the handshake input goes high. Operator 'aO' enables its output as soon as one bit is set at the control bus. The 'Store' function passes on the write bits of register 'wrR' when the busy input is low. Buffer 'doB' enables its output when at least one write strobe is set at the control bus. The hold bit at output 'hc' goes low as soon as the busy input goes low.

14.7.4 Z-bus communications

Operator 'ZO' communicates with all other blocks in the transputer via the external Z-bus. In the 'Hold' function all outputs are zero. The function 'Schedule' sets output 'rEx'. The scheduler will answer this request by reading the workspace descriptor via the Z-bus. Operator 'ZO' waits as a slave of the Z-bus until it is addressed with the U-bus (block address is defined by operator 'ba'). Then it removes the request and it enables buffer 'ZOB' to put the value of the internal Z-bus at the external Z-bus. The internal Z-bus should carry the workspace descriptor of the process to schedule. The hold output 'hZ' is set as long as the executer is not addressed with the U-bus.

In order to get the next ready process, operator 'ZO' performs function 'Extract'. This function sets output 'nEx'. The scheduler will answer this request by writing the workspace descriptor of the next ready process via the Z-bus to the executer. Operator 'ZO' waits until it is addressed with the U-bus. Then it enables buffer 'ZiB' to put the value of the external Z-bus at the internal Z-bus. Of course output 'Z' of the 'ALU' must be disabled at this moment. Operator 'ZO' does not remove the request immediately, but it checks the value at the Z-bus first. If this value is 80000000_16 or 80000001_16 then no process is ready, in other words both process queues are empty. In this case operator 'ZO' holds the request 'nEx' high, to keep asking for a valid workspace descriptor. At this point is waited until a process in the timers, links or event channel becomes ready. Because the schedule requests have higher priority than the extract request, the scheduler will first schedule the ready process and then extract the process for the executer. The hold output 'hZ' follows the request output.

In order to read or write some register of other block in the transputer, operator 'ZO' has
a number of functions to request for the Z-bus and become master. Therefore output \textit{req} carries a request bit, a block address, a register address and a read/write bit. The Z-bus arbiter will answer this request by making input \textit{ack} high. Then the executer has the Z-bus for one clock cycle and determines the value of the U-bus. The acknowledge signal will enable buffer \textit{ZIB} if reading or buffer \textit{ZOB} if writing. The hold output \textit{hZ} remains high until the acknowledge input goes high. The pointer of the process queues in the scheduler can be read with the functions \textit{ReadF0}, \textit{ReadB0}, \textit{ReadF1} and \textit{ReadB1}, and can be written with the functions \textit{WriteF0}, \textit{WriteB0}, \textit{WriteF1} and \textit{WriteB1}. The values of clock register of the timers can be read and written with the functions \textit{ReadT0}, \textit{ReadT1}, \textit{WriteT0} and \textit{WriteT1}. For the link interfaces are not separated functions, because that would give a large number of functions. In this case the channel address at the D-bus is used to determine the block address. The ROM \textit{baROM} performs the translation, according to table X.

Table X Translation of channel address to block address

<table>
<thead>
<tr>
<th>bit 2..4 of channel address</th>
<th>block address 'ba'</th>
<th>block name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>Link 0 Out</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>Link 1 Out</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>Link 2 Out</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>Link 3 Out</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>Link 0 In</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>Link 1 In</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>Link 2 In</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>Link 3 In</td>
</tr>
</tbody>
</table>

The functions \textit{ReadLW}, \textit{WriteLW}, \textit{WriteLA}, \textit{WriteLC} use this block address input \textit{ba}. 
15. EXECUTION OF INSTRUCTIONS

In the previous chapter the functions of the operators in the executer are described, but not how all the different instructions use these functions. We saw that instructions use one or more instructions codes, which define the function of all operators in the executer. In this chapter a list of all instruction codes with the accompanying function of each operators is given. This chapter does not give the definition of the instructions, because these can be found in the reference books [INMOS88a] en [Mitche90]. I am only going to list the implementation of the instructions. The instruction codes can be found in the control specification of each operator, but you cannot get an overview as depicted in this chapter.

The instructions can be broken down in 9 groups:

- Addressing and memory access
- Arithmetic and logical
- Branching and program control
- Process scheduling and control
- Inter-process communication
- Timer control
- Flag control
- Additional instructions on T800 (compared with T414)
- Floating-point arithmetic (not implemented).
### Table XI Addressing and memory access instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>code</th>
<th>A</th>
<th>O</th>
<th>TS</th>
<th>SelX</th>
<th>Hold</th>
<th>Hold</th>
<th>Hold</th>
<th>hold</th>
<th>Reset</th>
<th>hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldl</td>
<td>F7₁₈</td>
<td>W</td>
<td>O</td>
<td>TS</td>
<td>Index</td>
<td>PushD</td>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stl</td>
<td>FD₁₈</td>
<td>W</td>
<td>O</td>
<td>A</td>
<td>Index</td>
<td>Pop</td>
<td>Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldnl</td>
<td>F₃₁₆</td>
<td>A</td>
<td>O</td>
<td>TS</td>
<td>Index</td>
<td>RepD</td>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stnl</td>
<td>FE₁₆</td>
<td>A</td>
<td>O</td>
<td>B</td>
<td>Index</td>
<td>Pop2</td>
<td>Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldip</td>
<td>F₁₁₈</td>
<td>W</td>
<td>O</td>
<td></td>
<td>Index</td>
<td>PushZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldnlp</td>
<td>F₅₁₆</td>
<td>A</td>
<td>O</td>
<td></td>
<td>Index</td>
<td>PushZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>abbr.</th>
<th>code</th>
<th>A</th>
<th>O</th>
<th>TS</th>
<th>SelX</th>
<th>Hold</th>
<th>Hold</th>
<th>Hold</th>
<th>hold</th>
<th>Reset</th>
<th>hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>lb</td>
<td>001₁₆</td>
<td>A</td>
<td></td>
<td>di</td>
<td>TS</td>
<td>SelX</td>
<td>RepZ</td>
<td>Load</td>
<td>Di</td>
<td>Inc</td>
<td></td>
</tr>
<tr>
<td>sb</td>
<td>03B₁₆</td>
<td>A</td>
<td>B</td>
<td>di</td>
<td></td>
<td>SetWr</td>
<td>SelX</td>
<td>Pop2</td>
<td>Store</td>
<td>Inc</td>
<td>load</td>
</tr>
<tr>
<td>wsub</td>
<td>0A₁₆</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bsub</td>
<td>0₂₁₆</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wcnt</td>
<td>03F₁₆</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>Sar2</td>
<td></td>
<td>PushZ</td>
<td></td>
<td>Inc</td>
<td></td>
</tr>
<tr>
<td>bcnt</td>
<td>34₁₆</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>Shl2</td>
<td></td>
<td>RepZ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ajw</td>
<td>FB₁₆</td>
<td>W</td>
<td>O</td>
<td></td>
<td></td>
<td>Index</td>
<td></td>
<td></td>
<td></td>
<td>load</td>
<td></td>
</tr>
<tr>
<td>gajw</td>
<td>03C₁₆</td>
<td>A</td>
<td>di</td>
<td>W</td>
<td>W</td>
<td>SelX</td>
<td>AddPr</td>
<td>RepD</td>
<td>Zi</td>
<td>Inc</td>
<td></td>
</tr>
<tr>
<td>ldp1</td>
<td>1B₁₆</td>
<td>A</td>
<td>I</td>
<td></td>
<td></td>
<td>AddE</td>
<td>RepZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It would go to far to describe all functions of the ‘ALU’ in this report. See the appendices for the function specifications of operator ‘ALU’. When loading from memory operator ‘DS’ must be in function ‘TS’ to three-states all buffers, so that memory can drive this bus. All other function names of operators ‘XS’, ‘YS’ and ‘OS’ are obviously. Operator ‘do’ loads the D-bus value in register ‘diR’ when performing function ‘Oi’ and the Z-bus value when performing function ‘Zi’.

The ‘move’ instruction also belongs to this group, but it is not included in table XI, because I have not implemented it. This instruction is able to copy a number of bytes in memory with the least number of read and write cycles. Because the source address and destination address may have different byte counters, this is not simply copying words. This instruction requires a lot of byte shifting, incrementing and decrementing, which is impossible to do with the current architecture of the executer. For instance it is impossible to increment register ‘CR’, without changing ‘AR’, ‘BR’, ‘diR’ and ‘doR’.

Because the functionality needed for this instruction looks like the functionality used in the
link interfaces, my coach came with the idea that the hardware of the link interfaces could be shared, so that the move instruction can be executed by this hardware. Because this means that the link interfaces must be completely rebuild and because of time lack, I did not come to this point. But I recommend to try this approach. Besides it saves a lot of 32 bits incrementers and decrementers that I use in this design.

### Table XII Single word arithmetic and logical instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>code</th>
<th>XS</th>
<th>YS</th>
<th>DS</th>
<th>ALU</th>
<th>ER</th>
<th>ST</th>
<th>do</th>
<th>SO</th>
</tr>
</thead>
<tbody>
<tr>
<td>rev</td>
<td>0O₁₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldc</td>
<td>F₄₁₀</td>
<td>O</td>
<td>A</td>
<td></td>
<td>SelY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mint</td>
<td>4₂₁₀</td>
<td></td>
<td></td>
<td></td>
<td>Mint</td>
<td>SelD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adc</td>
<td>F₈₁₀</td>
<td>A</td>
<td>O</td>
<td></td>
<td>AddE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>0₅₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>AddE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>0C₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>SubE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>5₃₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>MulE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>5₂₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>AddE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>diff</td>
<td>0₄₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>SubE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>prod</td>
<td>0₈₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>MulE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmul</td>
<td>7₂₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>FMulE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eqc</td>
<td>F₇₁₀</td>
<td>A</td>
<td>O</td>
<td></td>
<td>Comp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gt</td>
<td>0₉₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>GT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>csub0</td>
<td>1₃₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>GTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ccnt1</td>
<td>4D₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>GTOE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>4₆₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>AND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>4B₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>OR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>3₃₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>XOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>3₂₁₀</td>
<td>A</td>
<td></td>
<td></td>
<td>NOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shl</td>
<td>4₁₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>SHL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shr</td>
<td>4₀₁₀</td>
<td>A</td>
<td>B</td>
<td></td>
<td>SHR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The division instructions 'div', 'rem' and 'ldiff' are not in the tables XII and XIII, because they are not implemented yet. Division is not a standard operator in IDaSS. Therefore this difficult functionality must be described in IDaSS operators. Very likely other people have already solved this problem, but I did not apply myself to this field.
### Table XIII Double word arithmetic and logical instructions

<table>
<thead>
<tr>
<th>default</th>
<th>A</th>
<th>O</th>
<th>TS</th>
<th>SelX</th>
<th>hold</th>
<th>Hold</th>
<th>Hold</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>abbr.</td>
<td>code</td>
<td>XS</td>
<td>YS</td>
<td>DS</td>
<td>ALU</td>
<td>ER</td>
<td>ST</td>
</tr>
<tr>
<td>ladd</td>
<td>16₁₈</td>
<td>₂₈₁₈</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LAddE</td>
<td>load</td>
<td>RepZ</td>
</tr>
<tr>
<td>lsum</td>
<td>37₁₈</td>
<td>₂₈₁₈</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LAddE</td>
<td></td>
<td>RepZ</td>
</tr>
<tr>
<td>lsub</td>
<td>38₁₈</td>
<td>₂₈₁₈</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSubE</td>
<td>load</td>
<td>RepZ</td>
</tr>
<tr>
<td>ldif</td>
<td>4F₁₈</td>
<td>₂₈₁₈</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSubE</td>
<td></td>
<td>RepZ</td>
</tr>
<tr>
<td>Imul</td>
<td>03₁₆</td>
<td>₃₈₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LMul</td>
<td></td>
<td>RepZ</td>
</tr>
<tr>
<td></td>
<td>13₁₆</td>
<td>₃₈₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LMul</td>
<td></td>
<td>RepZ</td>
</tr>
<tr>
<td></td>
<td>23₁₆</td>
<td>₃₈₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LMul</td>
<td></td>
<td>RepZ</td>
</tr>
<tr>
<td>Isl</td>
<td>03₁₆</td>
<td>₃₈₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSHL</td>
<td></td>
<td>PushZ</td>
</tr>
<tr>
<td></td>
<td>13₁₆</td>
<td>₃₈₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSHL</td>
<td></td>
<td>PushZ</td>
</tr>
<tr>
<td></td>
<td>23₁₆</td>
<td>₃₈₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSHL</td>
<td></td>
<td>PushZ</td>
</tr>
<tr>
<td>Ishr</td>
<td>03₅₆</td>
<td>₃₅₃₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSHR</td>
<td></td>
<td>PushZ</td>
</tr>
<tr>
<td></td>
<td>13₅₆</td>
<td>₃₅₃₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSHR</td>
<td></td>
<td>PushZ</td>
</tr>
<tr>
<td></td>
<td>23₅₆</td>
<td>₃₅₃₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSHR</td>
<td></td>
<td>PushZ</td>
</tr>
<tr>
<td>Xdble</td>
<td>01D₁₆</td>
<td>₃₀₁₆</td>
<td>A</td>
<td>A</td>
<td>di</td>
<td>XDBLE</td>
<td></td>
<td>RepZ</td>
</tr>
<tr>
<td>Csgnl</td>
<td>4C₁₆</td>
<td>₂₈₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>CSNGLE</td>
<td>load</td>
<td>PopRepZ</td>
</tr>
<tr>
<td>Xword</td>
<td>3A₁₆</td>
<td>₁₈₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>XWOR</td>
<td>load</td>
<td>PopRepZ</td>
</tr>
<tr>
<td>Cword</td>
<td>56₁₆</td>
<td>₂₈₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>CWOR</td>
<td>load</td>
<td>Pop</td>
</tr>
<tr>
<td>Norm</td>
<td>01₉₁₆</td>
<td>₁₉₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSOne</td>
<td></td>
<td>PushZ</td>
</tr>
<tr>
<td></td>
<td>11₉₁₆</td>
<td>₁₉₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSOne</td>
<td></td>
<td>PushZ</td>
</tr>
<tr>
<td></td>
<td>21₉₁₆</td>
<td>₁₉₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSOne</td>
<td></td>
<td>PushZ</td>
</tr>
<tr>
<td></td>
<td>31₉₁₆</td>
<td>₁₉₁₆</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>LSOne</td>
<td></td>
<td>PushZ</td>
</tr>
</tbody>
</table>
## Table XIV Branching and program control instructions

<table>
<thead>
<tr>
<th>default</th>
<th>A</th>
<th>O</th>
<th>TS</th>
<th>SelX</th>
<th>Hold</th>
<th>Hold</th>
<th>Hold</th>
<th>hold</th>
<th>Next</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbr.</td>
<td>code</td>
<td>XS</td>
<td>YS</td>
<td>DS</td>
<td>ALU</td>
<td>ST</td>
<td>RW</td>
<td>do</td>
<td>WR</td>
<td>IO</td>
</tr>
<tr>
<td>cj</td>
<td>0FA&lt;sub&gt;16&lt;/sub&gt;</td>
<td>I</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JumpZ</td>
</tr>
<tr>
<td>j</td>
<td>F0&lt;sub&gt;16&lt;/sub&gt;</td>
<td>I</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JumpZ</td>
</tr>
<tr>
<td>lend</td>
<td>021&lt;sub&gt;16&lt;/sub&gt;</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>121&lt;sub&gt;16&lt;/sub&gt;</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>221&lt;sub&gt;16&lt;/sub&gt;</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>321&lt;sub&gt;16&lt;/sub&gt;</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>421&lt;sub&gt;16&lt;/sub&gt;</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>521&lt;sub&gt;16&lt;/sub&gt;</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>621&lt;sub&gt;16&lt;/sub&gt;</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>721&lt;sub&gt;16&lt;/sub&gt;</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>0F9&lt;sub&gt;16&lt;/sub&gt;</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1F9&lt;sub&gt;16&lt;/sub&gt;</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2F9&lt;sub&gt;16&lt;/sub&gt;</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3F9&lt;sub&gt;16&lt;/sub&gt;</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4F9&lt;sub&gt;16&lt;/sub&gt;</td>
<td>I</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>020&lt;sub&gt;16&lt;/sub&gt;</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>120&lt;sub&gt;16&lt;/sub&gt;</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcall</td>
<td>06&lt;sub&gt;16&lt;/sub&gt;</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. The state machine controller changes this instruction code to 1FA<sub>16</sub> if ‘AR’ = 0.
2. The state machine controller changes this instruction code to 721<sub>16</sub> if ‘AR’ = 0.
### Table XV Process scheduling instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>code</th>
<th>XS</th>
<th>YS</th>
<th>DS</th>
<th>ALU</th>
<th>ST</th>
<th>RW</th>
<th>WR</th>
<th>IO</th>
<th>SO</th>
<th>ZO</th>
<th>Hold</th>
<th>Next</th>
<th>Reset</th>
<th>Hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>startp</td>
<td>00D_{16}</td>
<td>A</td>
<td>B</td>
<td>I</td>
<td>A</td>
<td>AddPr</td>
<td>Reverse</td>
<td>Store</td>
<td>Inc</td>
<td>Schedule</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10D_{16}</td>
<td>W</td>
<td>I</td>
<td>A</td>
<td>AddE</td>
<td>RepZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20D_{16}</td>
<td></td>
<td></td>
<td></td>
<td>Dec</td>
<td>RepP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table XVI Process control instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>code</th>
<th>XS</th>
<th>YS</th>
<th>DS</th>
<th>ALU</th>
<th>ST</th>
<th>RW</th>
<th>do</th>
<th>SO</th>
<th>ZO</th>
<th>Hold</th>
<th>Next</th>
<th>Reset</th>
<th>Hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>sthf</td>
<td>18_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SelX</td>
<td></td>
<td>Pop</td>
<td>WriteFO</td>
</tr>
<tr>
<td>sthb</td>
<td>50_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SelX</td>
<td></td>
<td>Pop</td>
<td>WriteBO</td>
</tr>
<tr>
<td>stlf</td>
<td>1C_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SelX</td>
<td></td>
<td>Pop</td>
<td>WriteF1</td>
</tr>
<tr>
<td>stlb</td>
<td>17_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SelX</td>
<td></td>
<td>Pop</td>
<td>WriteB1</td>
</tr>
<tr>
<td>saveh</td>
<td>03E_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>disable</td>
<td>Store</td>
<td>Zi</td>
<td>Inc</td>
</tr>
<tr>
<td></td>
<td>13E_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SelX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>23E_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>disable</td>
<td>Store</td>
<td>Zi</td>
<td>Inc</td>
</tr>
<tr>
<td></td>
<td>33E_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IncWord</td>
<td>Pop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>savel</td>
<td>03D_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>disable</td>
<td>Store</td>
<td>Zi</td>
<td>Inc</td>
</tr>
<tr>
<td></td>
<td>13D_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SelX</td>
<td></td>
<td></td>
<td>Inc</td>
</tr>
<tr>
<td></td>
<td>23D_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>disable</td>
<td>Store</td>
<td>Zi</td>
<td>Inc</td>
</tr>
<tr>
<td></td>
<td>33D_{16}</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IncWord</td>
<td>Pop</td>
<td></td>
<td>Inc</td>
</tr>
<tr>
<td>ldpri</td>
<td>1E_{16}</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pr</td>
<td>PushZ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table XVII Inter-process communication instructions

<table>
<thead>
<tr>
<th>default</th>
<th>A</th>
<th>TS</th>
<th>SelX</th>
<th>Hold</th>
<th>Hold</th>
<th>Hold</th>
<th>hold</th>
<th>Hold</th>
<th>Reset</th>
<th>Hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbr.</td>
<td>code</td>
<td>XS</td>
<td>DS</td>
<td>ALU</td>
<td>ST</td>
<td>RW</td>
<td>do</td>
<td>WR</td>
<td>IO</td>
<td>SO</td>
</tr>
<tr>
<td>out</td>
<td>00B₁₆¹</td>
<td>B</td>
<td>W</td>
<td>DecX</td>
<td>Load</td>
<td>Di</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
</tr>
<tr>
<td></td>
<td>10B₁₆²</td>
<td>B</td>
<td>I</td>
<td>DecX</td>
<td>Store</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
</tr>
<tr>
<td></td>
<td>20B₁₆</td>
<td>W</td>
<td>C</td>
<td>Dec3Word</td>
<td>Store</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
</tr>
<tr>
<td></td>
<td>30B₁₆</td>
<td>W</td>
<td>C</td>
<td>Dec3Word</td>
<td>Disable</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
</tr>
<tr>
<td></td>
<td>40B₁₆</td>
<td>W</td>
<td>DecWord</td>
<td>Load</td>
<td>JumD</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
</tr>
<tr>
<td></td>
<td>50B₁₆</td>
<td>W</td>
<td>DecWord</td>
<td>Load</td>
<td>JumD</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
<td>Inc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>60B₁₆</th>
<th>move 'AR' bytes from address 'CR' to address ['diR'] - 3, reset channel and schedule the process with workspace descriptor in 'diR'</th>
</tr>
</thead>
<tbody>
<tr>
<td>70B₁₆</td>
<td>C</td>
</tr>
<tr>
<td>80B₁₆</td>
<td>A</td>
</tr>
<tr>
<td>90B₁₆</td>
<td>W</td>
</tr>
<tr>
<td>AOB₁₆</td>
<td>W</td>
</tr>
<tr>
<td>B0B₁₆</td>
<td>W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>607₁₆</th>
<th>move 'AR' bytes from address ['diR'] - 3 to address 'CR', reset channel and schedule the process with workspace descriptor in 'diR'</th>
</tr>
</thead>
<tbody>
<tr>
<td>707₁₆</td>
<td>C</td>
</tr>
<tr>
<td>807₁₆</td>
<td>A</td>
</tr>
<tr>
<td>907₁₆</td>
<td>W</td>
</tr>
<tr>
<td>A07₁₆</td>
<td>W</td>
</tr>
<tr>
<td>B07₁₆</td>
<td>W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>reset ch</th>
<th>A</th>
<th>Mint</th>
<th>RepD</th>
<th>Load</th>
<th>Zi</th>
<th>Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>012₁₆⁵</td>
<td>A</td>
<td>X</td>
<td>Store</td>
<td>Vis</td>
<td>Inc</td>
<td>ReadLW</td>
</tr>
<tr>
<td>112₁₆</td>
<td>A</td>
<td>X</td>
<td>Store</td>
<td>Vis</td>
<td>Inc</td>
<td>ReadLW</td>
</tr>
<tr>
<td>212₁₆</td>
<td>A</td>
<td>X</td>
<td>Store</td>
<td>Vis</td>
<td>Inc</td>
<td>ReadLW</td>
</tr>
</tbody>
</table>

1. The state machine controller changes this instruction code to 70B₁₆ if the channel address in 'BR' is a link address (80000000₁₆ through 8000001C₁₆).
2. The state machine controller changes this instruction code to 60B₁₆ to call the 'move' instruction (not implemented yet).
3. The state machine controller changes this instruction code to 707₁₆ if the channel address in 'AR' is a link address (see note 1).
4. The state machine controller changes this instruction code to 607₁₆ (see note 2).
5. The state machine controller changes this instruction code to 212₁₆ if the channel address in 'AR' is a link address (see note 1).

The 'out' and 'in' instructions are not completely implemented yet, because they need the
move functionality to perform the actual communication. The transputer has two extra instructions to transmit a byte or a word: ‘outbyte’ and ‘outword’. These instructions are very similar to the ‘out’ instruction. I did not implement these two instructions, because the ‘out’ instruction is not finished yet. This group also contains 11 ‘Alt’ instructions, which are not implemented yet. These are complicated instructions which make a lot of decision dependent actions. I have not got the time to devote myself to this functionality.

<table>
<thead>
<tr>
<th>Table XVIII Timer control instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>default code abbr.</td>
</tr>
<tr>
<td>default A SelX Hold</td>
</tr>
<tr>
<td>sttimer 054₁₀ 154₁₀</td>
</tr>
<tr>
<td>Idtimer 022₁₆ 122₁₆</td>
</tr>
</tbody>
</table>

1 The state machine controller changes this instruction code to 122₁₆ if the current process has low priority (bit 0 of register ‘WR’ is set).

Not displayed in this group is the ‘tin’ instruction. This instruction waits for a certain time. If the time to wait for has already passed, the process simply continues. Else this process inserts itself in the timer queues. Therefore this instruction must search through a sorted linked list of processes. This is not very simple to implement and that is why I left this instruction for a while. But this while is still going on.

<table>
<thead>
<tr>
<th>Table XIX Flag control instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>default code abbr.</td>
</tr>
<tr>
<td>seterr 10₁₆</td>
</tr>
<tr>
<td>testerr 29₁₆</td>
</tr>
<tr>
<td>clrhalterr 57₁₆</td>
</tr>
<tr>
<td>sethalter 58₁₆</td>
</tr>
<tr>
<td>testhalterr 59₁₆</td>
</tr>
</tbody>
</table>

This group has also a forgotten instruction: ‘testpranal’. This is a simple instruction that pushes the ‘Analyse’ flag on the stack. But the analyse functionality is not implemented, so there is no flag to push.
This group has a few vague instructions, which I have not implemented. The instructions ‘crcword’ and ‘crcbyte’ perform cyclic redundancy check on respectively a word or a byte. Further the instruction ‘bitrevnbits’ reverses the bottom n bits of a word. These instructions cannot simply be implemented with IDaSS operators, and I have not spent any time on these unimportant instructions.

At last the IMS T800 transputer has four instructions to move 2 dimensional blocks of data. These instructions perform copying, overlaying and clipping of graphics pictures based on byte sized pixels. Because I did not manage to implement the ordinary move instruction, I did not even try to implement these instructions.
16. PERFORMANCE

Because this design is not ready for silicon compilation, its performance measured in time units cannot be calculated. Nevertheless the performance measured in numbers of clock cycles is known. In this chapter the number of clock cycles needed to execute each instruction on this IDaSS design are compared with the IMS T800.

The reference book [INMOS88a] gives the number of clock cycles that each instruction takes to execute on the IMS T800. I have tried to make the same assumptions for my design. The reference book assumes all memory access is to on chip RAM. Instructions that can cause descheduling, such as communication, assume that communication proceeds immediately without the process being descheduled. I took some additional assumptions:

- The fetcher is assumed to have the instruction already loaded, so waiting for fetching the instruction is not counted.
- A jump is assumed to take two clock cycles, that is one clock cycle for loading the new instruction pointer in the instruction pointer register, while resetting the decoder, and one clock cycle for loading the next instruction in the function and operand register of the executer. So fetching of the instruction is not counted.
- The load/store unit is assumed to be not busy. In other words no waiting for previous store actions is counted.
- The workspace cache is assumed to contain the data before loading. In this case loading takes only one clock cycle.
- Scheduling and extracting processes is assumed to take two clock cycles: one for loading the request in the scheduler and one for servicing the request. Thus waiting for the servicing of other requests in the scheduler is not counted.
- Reading or writing via the Z-bus is assumed to take one clock cycle. Waiting for the scheduler to release the Z-bus is not counted.

The following symbols are used:

- \( b \) the bit number of the highest bit set in register ‘AR’
- \( n \) the number of bits of a shift
- \( w \) the number of words, plus non word aligned partial words, in a message.

The instructions are listed in a similar order to the order in which they were introduced in the previous chapter.
Table XXI Comparing addressing and memory access instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>IMS T800</th>
<th>IDaSS design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idl</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>stl</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ldnl</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>stnl</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ldip</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ldnlp</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>lb</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>sb</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>wsub</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>bsub</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>wcnt</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>bcnt</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ajw</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>gajw</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ldpi</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

For all instructions the IDaSS design needs less or equal clock cycles as the IMS T800. Note that the IDaSS design needs also two clock cycle for the load instructions ‘Idl’ and ‘ldnl’, if the data is not present in the workspace cache. Because of extra functions in the ‘ALU’ the IDaSS design performs the instructions ‘lb’, ‘sb’ and ‘wcnt’ much quicker than the IMS T800.

The same goes for all arithmetic instructions (see table XXII). By adding a lot of functionality to the ‘ALU’ all arithmetic and logical instructions could be performed in one clock cycle. Only due to data path restrictions, some instructions need more clock cycles. Remember that shifting a 32 bits word in one clock cycles, requires an enormous and slow barrel shifter. The IMS T800 does not use a barrel shifter, but shifts only one bit per clock cycle. Likewise 32 bits multiplication is done with the binary operator ‘*’ in IDaSS, but this becomes a very large and slow macrocell in silicon.
Table XXII Comparing arithmetic and logical instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>IMS T800</th>
<th>IDaSS design</th>
</tr>
</thead>
<tbody>
<tr>
<td>rev</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ldc</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mint</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>adc</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sub</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mul</td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>sum</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>diff</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>prod</td>
<td>b+4</td>
<td>1</td>
</tr>
<tr>
<td>fmul</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>eqc</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>gt</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>csub0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ccnt1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>and</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>or</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>xor</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>not</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>shl</td>
<td>n+2</td>
<td>1</td>
</tr>
<tr>
<td>shr</td>
<td>n+2</td>
<td>1</td>
</tr>
<tr>
<td>ladd</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>lsum</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>lsub</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ldiff</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>lmul</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>lshl</td>
<td>n+3</td>
<td>3</td>
</tr>
<tr>
<td>lshr</td>
<td>n+3</td>
<td>3</td>
</tr>
<tr>
<td>xdbl</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>csngl</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>xword</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>cword</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>norm</td>
<td>37</td>
<td>4</td>
</tr>
</tbody>
</table>
Table XXIII Comparing branching and program control instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>IMS T800</th>
<th>IDaSS design</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>cj</td>
<td>4</td>
<td>2</td>
<td>jump taken</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>jump not taken</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>2</td>
<td>loop back</td>
</tr>
<tr>
<td>lend</td>
<td>10</td>
<td>8</td>
<td>exit loop</td>
</tr>
<tr>
<td>call</td>
<td>7</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>gcall</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Table XXIV Comparing process scheduling and control instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>IMS T800</th>
<th>IDaSS design</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>startp</td>
<td>12</td>
<td>4</td>
<td>no more child processes</td>
</tr>
<tr>
<td>endp</td>
<td>13</td>
<td>6</td>
<td>else</td>
</tr>
<tr>
<td>stopp</td>
<td>11</td>
<td>5</td>
<td>error flag is set</td>
</tr>
<tr>
<td>stoperr</td>
<td>11</td>
<td>5</td>
<td>else</td>
</tr>
<tr>
<td>runp</td>
<td>10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>sthf</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sthb</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>stlf</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>stlb</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>saveh</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>savel</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ldpri</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Here can be seen that the scheduler in the IDaSS design performs some tasks for the executer, where the executer of the IMS T800 has to do it by himself. In the IDaSS design the scheduler works not only for the timers, links and event channel, but also for the executer. That simplifies execution of the process scheduling instructions.
Table XXV Comparing inter-process communication instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>IMS T800</th>
<th>IDaSS design</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>20</td>
<td>8</td>
<td>first ready on internal channel</td>
</tr>
<tr>
<td></td>
<td>2w + 20</td>
<td>1 + move 7</td>
<td>second ready on internal channel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>link</td>
</tr>
<tr>
<td>in</td>
<td>20</td>
<td>8</td>
<td>first ready on internal channel</td>
</tr>
<tr>
<td></td>
<td>2w + 18</td>
<td>1 + move 7</td>
<td>second ready on internal channel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>link</td>
</tr>
<tr>
<td>resetch</td>
<td>3</td>
<td>2</td>
<td>internal channel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>link</td>
</tr>
</tbody>
</table>

The IMS T800 does not make any difference in handling an internal channel or a link interface. Maybe the number of clock cycles need is coincidently the same or the architecture allows to make no difference. Maybe internal channel communication is performed by shared link hardware.

Table XXVI Comparing timer control instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>IMS T800</th>
<th>IDaSS design</th>
</tr>
</thead>
<tbody>
<tr>
<td>sttimer</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ldtimer</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

This is the only point where the IDaSS design loses from the IMS T800. The IMS T800 is able to store a new time in both timers in one clock cycle, while the IDaSS design needs two clock cycles. The current Z-bus protocol does not allow to write to several blocks in a single clock. But this could be changed, by making an exception for the timers. For example timer 1 can also load the value at the Z-bus if timer 0 is addressed to load the workspace register.

Table XXVII Comparing flag control instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>IMS T800</th>
<th>IDaSS design</th>
</tr>
</thead>
<tbody>
<tr>
<td>seterr</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>testerr</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>clrhalterr</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sethalterr</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>testhalterr</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
### Table XXVIII Comparing additional T800 instructions

<table>
<thead>
<tr>
<th>abbr.</th>
<th>IMS T800</th>
<th>IDaSS design</th>
</tr>
</thead>
<tbody>
<tr>
<td>dup</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bitcnt</td>
<td>b + 2</td>
<td>1</td>
</tr>
<tr>
<td>bitrevword</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>wsubdb</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
To get a picture of the real world, I have added table XXIX with the performance of the whole current transputer family. The performance of the T9000 cannot be found in data books yet, but I have collected this information from the articles [May93], [Shephe92] and Internet news [News94].

Table XXIX Performance of the transputer family

<table>
<thead>
<tr>
<th></th>
<th>T212</th>
<th>T414</th>
<th>T800</th>
<th>T9000</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>architecture</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>bit</td>
</tr>
<tr>
<td>internal clock speed</td>
<td>20</td>
<td>20</td>
<td>30</td>
<td>50</td>
<td>MHz</td>
</tr>
<tr>
<td>peak instruction rate</td>
<td>20</td>
<td>20</td>
<td>30</td>
<td>200</td>
<td>MIPS</td>
</tr>
<tr>
<td>peak floating point instruction rate</td>
<td>-</td>
<td>-</td>
<td>4.3</td>
<td>25</td>
<td>Mflops</td>
</tr>
<tr>
<td>on-chip static RAM / cache</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>Kbytes</td>
</tr>
<tr>
<td>sustained data rate to internal memory</td>
<td>40</td>
<td>80</td>
<td>120</td>
<td>800</td>
<td>Mbytes/sec</td>
</tr>
<tr>
<td>sustained data rate to external memory</td>
<td>20</td>
<td>26.6</td>
<td>40</td>
<td>200</td>
<td>Mbytes/sec</td>
</tr>
<tr>
<td>bidirectional data rate per link</td>
<td>1.6</td>
<td>1.6</td>
<td>2.35</td>
<td>17.58</td>
<td>Mbytes/sec</td>
</tr>
</tbody>
</table>
17. STRIPPING IDASS DOCUMENT FILES

In order to reduce the size of the documentation generated by IDaSS, I designed a tool to strip these document files automatically from not useful text. This chapter describes the usage and working of this tool.

IDaSS has an option to create a documentation file of a design or a part of it. The purpose of this option is to give a full description of the design in plain ASCII text format. In this way it would be possible to rebuild the design with only this text.

But for archive purposes it is more convenient to print the schematic as a picture and add the missing information in text format. The generated documentation file contains too much information for this purpose. Generally the information given by the picture could be deleted from the documentation file. In the past students have done this by hand, which means loads of boring work.

I thought that this could be done automatically and I was right. Coincidentally my housemate, Mark Venbrux, told me about the AWK tool under UNIX. This tool is intended to process text files, exactly what I needed. IDaSS generates text files under MS-DOS, so these had to be translated to UNIX text files, stripped with AWK under UNIX, and again translated back to DOS text files. Later I got myself a MS-DOS version of AWK, so the translating can be omitted. After a lot of tests and recommendations from A.C. Verschueren I came to a very useful tool, which has been and will be used by many other IDaSS users. By the way, the new version of IDaSS can switch off some kinds of information in the documentation generation. So this tool might become useless in the future. But that is the better way of course.

17.1 Usage of the stripper

The AWK tool can be executed under MS-DOS by typing the following at the prompt:

```
strpidas filename (filename without extension!)
```

This starts a batch file with the following contents:

```
awk -f strpidas.awk filename.txt > filename.doc
```

The filename.txt is the original IDaSS document file and the filename.doc is the stripped document file. The AWK program strpidas.awk will be explained in the next paragraph.
17.2 Explanation of the AWK program

Now I will give a brief explanation of the AWK program strpidas.awk.

First this program checks if the input file is a legal IDaSS document file. If the string 'IDaSS V0.08m document generated' cannot be found in the first line, then the whole input file will be copied unchanged to the output file.

A legal IDaSS document file, contains a number of block descriptions separated by a line of '=' characters. The first line of a block description always contains the name and type of the block. Dependent on the block type a number of lines of this block description will be printed. For a CAM this number is not a constant. In this case all lines through 'The default function is' are printed.

When a block is excluded from documentation, the third line of the block description will contain the text 'IS NOT DOCUMENTED'. The AWK program will look ahead to search this string. If it can be found this block will be skipped.

All kinds of blocks can contain 'Designer Comments', 'Control specification', 'Text for function' and 'Text for state'. These are always printed, starting with one of these strings and ending with the pattern '-------'.

Off-schematic connectors are not printed unless they have any designer comments. To achieve this the first line of the block description is saved in a variable, but not yet printed. When designer comments are found the block separation line and the variable will be printed before the designer comments itself. The same goes for bus descriptions, except that they are not separated by a line of '=' characters, but a line of '-' characters.

Connector descriptions are always printed. But I have compressed the information of 2 or 3 lines separated by an empty line, to one line without separation of an empty line and aligned to each other. The width of the connector has been taken from the second line and the default state of a three-state output connector is in the third line. The new connector descriptions are:

- Control connector (5 bits) without name
- Input connector (1 bit) with name 'h'
- Continuous output connector (30 bits) with name 'a'
- Three-State output connector (32 bits) with name 'd', default disabled.

The connectors of a CAM, FIFO and LIFO can have functions. To print these functions, the lines after 'The functions of the connectors are as follows' until 'These connectors are explained in more detail below' are printed to the output file.
The goal has not been reached completely, but a large part and the most typical transputer features has been implemented. These are:

- The link interfaces
- The process scheduler
- The event channel
- The timers
- Internal memory
- External memory interface for static RAM.
- The fetcher
- The decoder
- The load/store unit
- The executer
- 80 instructions

In some points my design is better than the INMOS T800 transputer. These points are:

- The decoder executes asynchronously all 'pfix' and 'nfix' instructions within one memory word. Only to pass a word boundary the executer is needed to change the instruction pointer, so that the fetcher will fetch the next word of instructions. This feature will speed up the execution of programs a lot, because the 'pfix' and 'nfix' instructions are needed everywhere data of more than 4 bits are used.
- The load/store unit pipes write cycles. In this way a write cycle takes only one clock cycle in the executer. Only if two store actions must be taken consecutively, then the executer must wait for the load/store unit to become ready with the previous store action.
- The workspace cache can speed up the loading of local variables. If data has been loaded or stored in the first 32 words of the workspace, then the next time this data is loaded, the workspace cache will give this data asynchronously. Because the executer has little working registers, these actions will take place very often, especially in loops.
- The fetcher can prefetch 8 instructions. If the first word of instructions pointed by the instruction pointer is loaded, then the fetcher continues with loading the next word of instructions. This will keep the executer going most of the time, without having to wait for fetching instructions. Only when jumping the fetcher must restart loading and the executer will have to wait.

But at some points the current IDaSS design is not better than the IMS T800:

- The 'move' instruction is very well implemented in the IMS T800, but it cannot be implemented in this IDaSS design. This could be solved by sharing the link hardware (see recommendations).
- This design has a very long asynchronous block chain. Many times the simulation got stuck on an asynchronous block chain of 21, while the maximum is 20. This could
indicate a low maximum clock frequency. Probably this problem lies in the wide spread control of the handshake signal. A solution could be to put the distributed functionality of the handshake signal in one operator (see recommendations).

Despite of the hard work and long time of my graduation period, there still is a long list of things remaining to be done. The features that have not been implemented yet are:

- 'move' instruction and 2d move instructions. See recommendations for implementation of these instructions.
- Divide instructions 'div', 'rem' and 'ldiv'.
- 'tin' instruction: search through a sorted linked list of timer processes and insert this process at the right place. Notice that this all happens in one instruction and not a program!
- 'Alt' instructions and 'Alt' handling in the 'out', 'outbyte', 'outword' and 'in' instructions.
- External memory interface for dynamic RAM.
- Floating-point unit and accompanying instructions.
- Time slice handling: Timer 0 generates a time slice signal, but the executer has no functionality for this signal yet. If a low priority process is in execution, then it should be added to the process queues and the next ready low priority process should be taken in execution.
- Interrupt handling: The scheduler generates an interrupt signal if a high priority process is scheduled, while the process currently in execution has low priority. But the executer has no functionality to handle this signal, yet.
- Updating the timer queues: if the scheduler schedules a timer process, it should also setup the next timer process into the timer hardware and update the timer queues.
- Analyse functionality and the 'testpranal'- instruction.
- Boot from the links.
19. RECOMMENDATIONS

About all the things that still have to be done or could have been done better, I have collected the ideas of how to tackle these problems in this chapter. Because of time storage I had not the opportunity to do this myself. Sometimes it was hard to stop designing and finish my part of the work with this report.

- Use partial frequency control to run timer 0 at exactly 1 MHz with an internal clock frequency of 17.5 and 22.5 MHz.

- Share hardware of all links, to leave a move instruction to this hardware. Considerations: only one link can read or write to memory at the same moment. Save a lot of big incrementers and decrementers. Save hardware to execute the move instruction in the executer.

- Place all registers of the links in a register file to save hardware. This register file can perfectly be combined with the shared hardware of all links.

- Use the floating-point unit designed by F.J.A. Korsten, G.H. Hotho, M. van Balen and A.C. Verschueren.

- Use the memory interface designed by E.B.C. Daniëls.

- Finish the process scheduler, so that it can update the timer queues when a timer requests for scheduling.

- Put the distributed functionality of the handshake signal in one operator, to decrease the asynchronous block chain and to speed up the design.
REFERENCES

[Bloks89] Bloks, R.H.J.
De architectuur van de transputer.
Collegediktaat Microcomputerarchitectuur, vakcode 5P020.

DOOM: A Decentralized Object-Oriented Machine.

POOL and DOOM a survey of ESPRIT 415 subproject A, Philips Research Laboratories.
P. 356-373.

[DeLeeu92] Leeuw, J. de en A. de Mes
Transputers: design and use as a building block.
Amsterdam: FWI University of Amsterdam, 1992.

[Flynn66] Flynn, M.J.
Very high-speed computing systems.

[Hey90] Hey, A.J.G.
Supercomputing with transputers-past, present and future.
P. 479-489.

[Hinton93] Hinton, J. and A. Pinder
Transputer hardware and system design.

[Hoare78] Hoare, C.A.R.
Communicating Sequential Processing.

The IMS T800 Transputer.

[INMOS88a] INMOS Limited
Transputer Instruction Set: A compiler writer’s guide.
INMOS doc. nr.: 72 TRN 119 05.

[INMOS88b] INMOS Limited
INMOS doc. nr.: 72 TRN 006 04.

[INMOS89a] INMOS Limited
The Transputer Applications Notebook: Architecture and Software.
INMOS doc. nr.: 72 TRN 206 00.

[INMOS89b] INMOS Limited
The Transputer Databook, 3rd ed.
INMOS doc. nr.: 72 TRN 203 02.

[INMOS89c] INMOS Limited
Transputer Technical Notes.
INMOS doc. nr.: 72 TRN 200.

[INMOS91] INMOS Limited
The T9000 transputer products overview manual.

[Kung90] Kung, H.T.
iWarp Multicomputer with an embedded switching network.

The T9000 Transputer.

Inside the transputer.

[Muijse90] Muijselaar, J.
Architectuur van de transputer: een beschrijving in pseudo Occam.
Stageverslagnr.: EB 264.

[News94] Internet news comp.sys.transputer
Several articles
REFERENCES

[Nicoud89] Nicoud J.D. and A.M. Tyrell
The transputer T414 Instruction Set.

[Peters91] Peterson, C. et al.
iWarp: A 100-MOPS, LIW microprocessor for multicomputers

[Shephe92] Shepherd, R.
The T9000.
In: Transputer applications - progress and prospects.
Amsterdam: IOS, 1992.
P. 77-81.

[Simar91] Simar, R.
The TMS320C40: A DSP for parallel processing.
In: ICASSP 91, Toronto, 14-17 May 1991.
Vol. 2, P. 1089-1092.

[Texas92] Texas Instruments
TMS320C4x User’s Guide.

[Tyrrel89] Tyrrell, A.M. and J.D. Nicoud
Scheduling and Parallel Operations on the Transputer.

[Versch90] Verschueren, A.C.
IDaSS for ULSI (IDaSS manual).
INDEX

Acknowledge ...................... 18
  overlapped .................... 18
Alt
  construct ...................... 7, 8
  instructions .................. 94
Applications .................... 4
AWK ................................ 105
Bus multiplexer .................. 35
C004 ................................ 3
C104 ................................ 3
C40 ................................ 5
Channel ............................ 7
  communication .................. 15
  event ........................... 9, 63
  external ....................... 15
  internal ....................... 15
  virtual ........................ 3
CISC ................................ 2
Concurrency ...................... 7
CPU ................................ 65
Decoder ............................ 67
Deschedule ........................ 14
DMA ................................ 11, 45
DOOM ................................. 6
EMI ................................ 11, 45
Error flag .......................... 10, 27, 41, 80
Event ............................... 9, 19
  acknowledge .................... 63
  request ........................ 63
Event channel .................... 63
External Memory Interface ....... 11, 45
Fetcher ............................. 65
Floating point unit .............. 3, 9, 17
Flynn ............................... 2
Function ........................... 67
Grain size ........................ 1
Halt-on-error flag ............... 80
IDA\$S ............................... 23
  documentation .................. 105
  implementation ................ 25
Implementation ................... 25
IMS T800 ............................ 9, 25, 97
INMOS .............................. 1, 26, 42, 59, 83, 87
Instruction ........................ 13
  nfix ............................ 13, 67
  opr ............................. 13
  pfix ............................ 13, 67
Instruction pointer .............. 12, 66, 67, 80
Instructions
  execution ....................... 87
  performance .................... 97
Interrupt ......................... 19, 28, 65, 72, 77
IWarp ............................... 6
Language ........................... 2, 17, 51
  communication .................. 3, 15
  control ........................ 53
  input ........................... 59
  interface ....................... 9, 16, 25, 53
  output .......................... 56
  speed ........................... 18, 26, 29, 42, 53
Load/store unit ................... 68
Memory ............................. 11
  external ........................ 11, 12, 33
  internal ........................ 11, 33
Microprocessor ................... 9
MIMD ............................... 2
Multiplex unit .................... 38
Multiprocessor ................... 2
Multiprocessor systems ........... 2
OCCAM ................................ 1, 7
OOSD project ...................... 23
Operand ............................ 67
Operand register ................ 12
PAR ................................. 7
Parallel computers ............... 1
Partial ............................ 41
Partial frequency control ....... 41
Performance ....................... 97
Priority ........................... 12

115
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>7</td>
</tr>
<tr>
<td>active</td>
<td>14</td>
</tr>
<tr>
<td>concurrent</td>
<td>13</td>
</tr>
<tr>
<td>inactive</td>
<td>14</td>
</tr>
<tr>
<td>queues</td>
<td>14, 71</td>
</tr>
<tr>
<td>scheduler</td>
<td>71</td>
</tr>
<tr>
<td>Processor</td>
<td>3, 12</td>
</tr>
<tr>
<td>pipelined, superscalar</td>
<td>3</td>
</tr>
<tr>
<td>RAM</td>
<td>11, 33</td>
</tr>
<tr>
<td>Register</td>
<td>12, 80</td>
</tr>
<tr>
<td>A</td>
<td>12, 80</td>
</tr>
<tr>
<td>B</td>
<td>12, 80</td>
</tr>
<tr>
<td>BPtr0</td>
<td>14, 72</td>
</tr>
<tr>
<td>BPtr1</td>
<td>14, 72</td>
</tr>
<tr>
<td>C</td>
<td>12, 80</td>
</tr>
<tr>
<td>FA</td>
<td>17</td>
</tr>
<tr>
<td>FB</td>
<td>17</td>
</tr>
<tr>
<td>FC</td>
<td>17</td>
</tr>
<tr>
<td>FPtr0</td>
<td>14, 72</td>
</tr>
<tr>
<td>FPtr1</td>
<td>14, 72</td>
</tr>
<tr>
<td>I</td>
<td>12, 80</td>
</tr>
<tr>
<td>O</td>
<td>12, 80</td>
</tr>
<tr>
<td>W</td>
<td>12, 80</td>
</tr>
<tr>
<td>RISC</td>
<td>2</td>
</tr>
<tr>
<td>ROM</td>
<td>11, 34</td>
</tr>
<tr>
<td>SASD project</td>
<td>23</td>
</tr>
<tr>
<td>Schedule</td>
<td>14</td>
</tr>
<tr>
<td>Scheduler</td>
<td>9, 13, 71</td>
</tr>
<tr>
<td>SEQ</td>
<td>7</td>
</tr>
<tr>
<td>SIMD</td>
<td>2</td>
</tr>
<tr>
<td>SISD</td>
<td>2</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>23</td>
</tr>
<tr>
<td>SPrMD</td>
<td>2</td>
</tr>
<tr>
<td>Stack</td>
<td>12, 17, 80, 82</td>
</tr>
<tr>
<td>Stripper</td>
<td>105</td>
</tr>
<tr>
<td>Supercomputers</td>
<td>1</td>
</tr>
<tr>
<td>System services</td>
<td>41</td>
</tr>
<tr>
<td>T212</td>
<td>1, 3</td>
</tr>
<tr>
<td>T414</td>
<td>3</td>
</tr>
<tr>
<td>T414a</td>
<td>1</td>
</tr>
<tr>
<td>T414b</td>
<td>1</td>
</tr>
<tr>
<td>T800</td>
<td>1, 3</td>
</tr>
<tr>
<td>internal architecture</td>
<td>9</td>
</tr>
<tr>
<td>T9000</td>
<td>1, 3, 9, 17, 103</td>
</tr>
<tr>
<td>Test system</td>
<td>31</td>
</tr>
<tr>
<td>Timer</td>
<td>9, 47</td>
</tr>
<tr>
<td>queues</td>
<td>14, 20, 71</td>
</tr>
<tr>
<td>timer 0</td>
<td>47</td>
</tr>
<tr>
<td>timer 1</td>
<td>48</td>
</tr>
<tr>
<td>TMS320C40</td>
<td>5</td>
</tr>
<tr>
<td>Transputer</td>
<td>1</td>
</tr>
<tr>
<td>internal architecture</td>
<td>9</td>
</tr>
<tr>
<td>U-bus</td>
<td>28</td>
</tr>
<tr>
<td>UNIX</td>
<td>105</td>
</tr>
<tr>
<td>VCP</td>
<td>3</td>
</tr>
<tr>
<td>Virtual channel processor</td>
<td>3</td>
</tr>
<tr>
<td>VLSI</td>
<td>1</td>
</tr>
<tr>
<td>Von Neumann</td>
<td>1</td>
</tr>
<tr>
<td>Workspace</td>
<td>12</td>
</tr>
<tr>
<td>descriptor</td>
<td>12, 80</td>
</tr>
<tr>
<td>pointer</td>
<td>12</td>
</tr>
<tr>
<td>Workspace cache</td>
<td>70</td>
</tr>
<tr>
<td>Z-bus</td>
<td>80, 84</td>
</tr>
<tr>
<td>arbiter</td>
<td>71</td>
</tr>
<tr>
<td>protocol</td>
<td>28</td>
</tr>
</tbody>
</table>