Eindhoven University of Technology

MASTER

# A small digital signal processor for Philips from specification via IDaSS and VHDL to silicon

Oerlemans, R.V.M.

*Award date:*
1994

Link to publication

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

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

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

EB 501

Eindhoven University of Technology
Department of Electrical Engineering
Digital Information Systems Group

Master thesis report
September 1993 - March 1994

# A small Digital Signal Processor for PHILIPS

## From specification via IDaSS and VHDL to silicon

By **ing. R.V.M. Oerlemans**

Supervisor **prof. ir. M.P.J. Stevens**
Coach **dr. ir. A.C. Verschueren**

The department of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of student projects and graduation reports.

# Abstract

The Digital Information Systems group of the Eindhoven University of Technology has developed an Interactive Design an Simulation System (IDaSS) for Ultra Large Scale Integration (ULSI) of digital circuits. With this tool it is possible to design and simulate a microcontroller on a very high level. Philips wanted to verify the use of IDaSS as a design tool. This has been done through designing and simulating the microcontroller MicroDsp with IDaSS after the functional specification of this chip had been approved. Then an implementation of the microcontroller in VHDL had to be made. This VHDL implementation will be used to create the MicroDsp chip layout using a silicon compiler. The size and performance of this chip give an indication of the benefits and disadvantages of IDaSS as a design tool. This report describes the design of the microcontroller MicroDsp. It is a chip with an 8-bit reduced instruction set microprocessor combined with an arithmetic accelerator unit, a timer unit, a serial interface and internal program and data memory. It provides a controller function and includes low-end digital signal processing capability for special applications. The chip is being developed in cooperation with Philips Semiconductors in Eindhoven. The Eindhoven University of Technology (TUE) is a partner in the development of this chip.

Philips has made a specification [MDSP93] of the MicroDsp. This specification is used for the design of the MicroDsp in IDaSS. During the design phase I have made modifications on the specification, erased errors and added new features. A new specification [MDSP94] was made that will be used to design the chip.

The MicroDsp microcontroller with the following elements:

- Peripheral Controller Cell (PCC)
- Arithmetic Accelerator Unit (AAU)
- PCC program memory of 4096+64 words of 16 bits
- PCC data RAM of 256 bytes
- 16 bit Capture Timer Unit (CTU)
- 10 I/O ports
- $I^2C$ bus interface and monitor control unit (IICC)

In this report the design in IDaSS and VHDL of the Capture Timer Unit and the $I^2C$ bus interface and monitor control unit are described.

An $I^2C$ bus interface and monitor control unit is used to control the MicroDsp with a "host" computer (PC). Communication is done via a serial two wire link. This means that a monitor program that is running on the PC can influence the progress of the program running on the MicroDsp. For example, the monitor program can start and stop the MicroDsp or read IO bus information or read the contents of a register of the PCC.

A capture-timer unit (CTU) is a combination of a capture unit and a timer unit. A timer unit is a time-base counter that counts a predefined number of clocks and then gives an interrupt. A capture unit is an element that can read and store the time in the time-base counter on the moment that a capture command is given.

All parts of the MicroDsp have been designed in IDaSS by the Eindhoven University of

Technology. The PCC processor and the program RAM have been made by Simons and Vostermans, the AAU coprocessor and the data RAM have been created by Verschueren while the Capture Timer Unit, $I^2C$ and Control, IO ports and MicroDsp toplevel have been build by me.

It was also the intention to describe all parts of the MicroDsp in VHDL. Therefore Simons and Vostermans have converted their IDaSS PCC to VHDL. The VHDL AAU has been made by Brand from Microtel. The Capture Timer unit and $I^2C$ and Control unit have been translated into VHDL by me. A VHDL description of the IO ports has not yet been made. For the program RAM and data RAM simple VHDL modules are used.

Designing with IDaSS has been done fast compared to the time necessary to develop such a processor immediately in VHDL or in a lower level description language. With a tool like IDaSS the efficiency and quality of the definition of new IC's can be improved considerably. If IDaSS is going to be used by Philips as a design tool it has to be accepted by a tool support group such as for example ED&T. A lot of disadvantages then have to be improved.

# Table of contents

# 1 Introduction

Philips wants to verify the use of IDaSS as design tool for their intergrated cirquits. A microprocessor is one kind of chip that Philips makes. Designing a new type of microprocessor is something that is only done by few because the world simply doesn't need many different microprocessors. Most designers try to speed up existing microprocessors. Only for a special kind of application a new type of microprocessor will have to be designed.

In one of the laboratories of Philips Semiconductors, such a special application is being developed. It is the controlling of a electric motor that is used in video recorders and disk drives. The controlling of the speed of such a motor has to be very accurate. Therefore a very fast but small and cheap microprocessor is needed to compute the speed and acceleration of the motor and to control the current that drives the motor. Very fast but small and cheap seem contradictive, but if the microprocessor only needs to perform a limited set of instructions this looks possible.

### Developing a chip

In this document, you will find a report of the design of a microcontroller called the MicroDsp. The MicroDsp is a microcontroller with an 8-bit reduced instruction set microprocessor combined with an arithmetic accelerator unit, a timer unit, a serial interface and internal program and data memory. It provides a controller function and includes low-end digital signal processing capability for special applications. The chip is being developed in cooperation with Philips Semiconductors in Eindhoven. This is done at the Product Concept and Application Laboratory in Eindhoven (PCALE). The laboratory has several groups such as a group for telecommunication systems, a television group and an industrial group. The industrial group has a subgroup called Motor Control. For this group, J. den Ouden has made a functional specification of the MicroDsp.

The Eindhoven University of Technology (TUE) is a partner in the development of this chip. The Digital Information Systems group has developed an Interactive Design an Simulation System (IDaSS) for Ultra Large Scale Integration (ULSI) of digital circuits. The designer of this tool IDaSS is Ad Verschueren. With IDaSS it is possible to design and simulate a microcontroller on a very high level. Philips wants to use IDaSS for the verification of the functional specification of the MicroDsp. This will be done through simulation of an IDaSS implementation of the MicroDsp. If the functional specification is approved, an implementation of the microcontroller in VHDL must be made. This VHDL implementation can be used to create the MicroDsp chip layout using a silicon compiler. The size and performance of this chip give an indication of the benefits and disadvantages of IDaSS as a design tool.

### Project team

At the TUE, the Digital Information Systems group has formed a MicroDsp project-team under leadership of professor Stevens. This team exists of Ad Verschueren, Wido Kruijtzer, Bart Vostermans, Peter Simons and myself.

Philips has two divisions working on the MicroDsp: PCALE and MicroTel. At PCALE Jos den Ouden, Roland Broekman, Ewout Rotte and Edwin Warrens work on the MicroDsp. The MicroTel team consists of Ronald Kemp and Hjalmar Brand.

In Tabel I a task overview is given of the people that are involved in the development of the MicroDsp:

**Table I** Relation between project items and designers

| Item Person | AV | WK | BV | PS | BO | JdO | RB | ER | EW | RK | HB |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Toplevel | | | | | s/i | s | | | | v | |
| PCC | | | i/v | i/v | | s/h | | | | | |
| AAU | i | | | | | s/h | | | | | v |
| CTU | | | | | s/i/v | s | | | | | |
| I2C | | | | | s/i/v | s | | | | | |
| I/O ports | | | | | s/i | s | | | | | |
| Program RAM | | | i | i | s | s | | | | | |
| Data RAM | i | | | | s | s | | | | | |
| IDaSS Tool | d | | | | | | | | | | |
| IDaSS->VHDL Compiler | | d | | | | | | | | | |
| PCC C Compiler | | | | | | | | | d | | |
| PCC Assembler | | | | | | | | | d | | |
| PCC Simulator | | | | | | | d | | | | |
| Applications | | | | | | | | m | | | |

Abbreviations:

| | | | | | |
|---|---|---|---|---|---|
| AV | = | Ad Verschueren | d | = | design |
| WK | = | Wido Kruijtzer | h | = | hand layout |
| BV | = | Bart Vostermans | i | = | idass |
| PS | = | Peter Simons | m | = | motor control |
| BO | = | Bob Oerlemans | s | = | specification |
| JdO | = | Jos den Ouden | v | = | vhdl |
| RB | = | Roland Broekman | | | |
| ER | = | Ewout Rotte | | | |
| EW | = | Edwin Warrens | | | |
| RK | = | Ronald Kemp | | | |
| HB | = | Hjalmar Brand | | | |

As can be seen in Tabel I, my part of the development of the MicroDsp is the rewriting of the specification of the MicroDsp toplevel, $I^2C$, CTU, RAMs and IO. Furthermore the IDaSS design and simulation of these blocks and the VHDL generation of the $I^2C$ and CTU blocks.

**About this report**

The next chapters will explain how the MicroDsp has been developed. In chapter 2 is explained how IDaSS can be used for the design of a microcontroller. In chapter 3 the development of the parts of the MicroDsp that have been designed by me will be described in detail. Other parts of the MicroDsp are described globally. In this chapter the paragraph numbering refers to the hierarchy of the MicroDsp design. For example, if a schematic is described at paragraph 3.1, a subschematic will be described at paragraph 3.1.1, etc. A paragraph is partitioned in three sections that represent three developing steps: specification, design and simulation. An example of a paragraph that describes the capture timer unit is:

3.1 Capture Timer Unit (CTU)
CTU specification
CTU design
3.1.1 Prescaler Unit (PRESCALER)
PRESCALER specification
PRESCALER design
...
PRESCALER simulation
3.1.2 Timer Register Unit (TIMER_REG)
...
CTU simulation
3.2 ...

Chapter 4 describes the VHDL code generation of the CTU and the $I^2C$ units. Finally, the last chapter will describe the results and give conclusions of my part of the developing of the MicroDsp.

# 2 IDaSS

The MicroDsp has been designed in IDaSS. IDaSS is an interactive design and simulation environment for digital circuits. It is targeted towards VLSI and ULSI designs of complex data processing hardware (microprocessors, coprocessors and signalprocessors of all kinds). It can also be used for simpler designs, as long as the complete design is a synchronous machine (a single clock source for all clocked elements in the design). Simulating asynchronous logic with internal feedbacks is impossible with IDaSS, because the built-in simulator is not designed to do so (the results will not mirror actual hardware behaviour).

IDaSS describes a design as a tree-like hierarchy of schematics. The schematics contain elements like registers, ALU's, memories, state machine controllers and the like, and are entered graphically. Rectangles (called 'blocks') represent all schematic elements, which are connected by lines representing the (bidirectional) buses. Small squares at the boundaries of the rectangles represent the input and (three-state) output ports of the elements, these are called 'connectors'. The connectors come in several shapes to make a distinction between input, (disabled) output, bidirectional and control connectors.

A controller can test and control the elements of the schematic it is placed in, and can change it's state based upon test results. Controllers can be placed in a schematic just like all other blocks. There operational characteristics are entered in textual form, describing a state machine. The language used can describe microprogrammed controllers (including a subroutine stack) and Moore state machines. Tests done by the controller can only be based upon directly clocked elements in the schematic.

Elements in a schematic can also be controlled by adding a 'control connector'. This connector can be connected to any bus in the system, the value of which will determine the functions of that block. A textual PLA-like specification 'couples' the values on the bus with the functions to be executed.

IDaSS is targeted towards ULSI by allowing multiple schematics and controllers to be present in a single design. This is done by allowing 'lower level' schematics to be placed in a schematic as a single element, thus forming a hierarchical 'tree' of schematics.

Controllers can test and control blocks placed in schematics at lower levels in the hierarchy. Controllers can also (to some extent) control other controllers in their own or lower level schematics. Synchronisation of controllers is simplified by the use of special 'semaphore' bits in registers and a user-defined set of 'signals' which can be used for system-wide communication between controllers.

### IDaSS connector symbols

IDaSS uses different symbols for connectors, as shown in Figure 1:

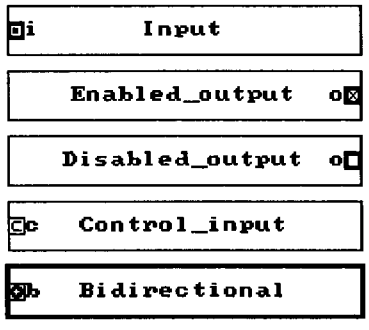


**Figure 1** Connectors

For a continuous output, the symbol for an enabled output is used. Three-state outputs show their state by toggling between the symbols for enabled and disabled outputs.

### IDaSS elements

IDaSS only knows a few basic elements. With these elements a complete schematic can be build. The elements are:

- Register
- Buffer
- Constant
- Operator
- Memory
- Schematic
- State controller

A register is a block that contains a number of clocked memory elements. The width of the register determines the number of elements used. An IDaSS register can hold, load, reset, set, increment or decrement. The register's function is controlled by a state controller or by a special connector at the register. A buffer is block that has an input, an output and an optional control connector. This control connecter determines if the buffers output is equal to the input (enabled) or if the output is floating (tree-state). A constant is a block which contains a constant value, but looses that value if a clock tick passes. An operator is a block that may contain one ore more functions. A function is a relation between an output and one ore more input connectors of the operator. An operator can be seen as a combinatorial network with AND's OR's etc. A memory is a block that may contain data. It can be RAM, ROM, FIFO, LIFO, etc. A schematic is a block that may contain several subblocks. In this way a hierarchical structure can be brought into the design. A state controller is a special block that is a combination of a state-register and a combinatorial network which determines the next

state. In every state, the controller controls other blocks to execute a function that is specified in that state. The next state is actually controlled by the contents of the registers in this schematic or lower level schematics. There are no wires connected on the state controller: test and control lines are hidden.

### IDaSS busses

Busses are used to connect blocks. A bus is a line that is connected to one or more connectors of blocks in the schematic. A bus can be several bits wide. The width of a bus is always the same as the width of the connectors the bus is connected to.

### IDaSS viewers

In order to have a continuous display of a register's contents or bus value, we can place a 'value viewer' on the schematic. A value viewer is a box that is connected to a register or a bus and displays its value. A 'function viewer' indicates the function a block is performing, for example: the function viewer of a register could indicate 'load' or 'reset'.

### IDaSS borders

Every block in an IDaSS schematic has a border. The border is thin if this block is the lowest level in the schematic. A thick border indicates that the block consists of more elements. This is a so called lower level schematic. A grey border is used to indicate a state controller.

### IDaSS automatic document generation

IDaSS can automatically generate a text containing documentation for almost any block in the system. It is also possible to generate documentation for the complete set of signals, any schematic (with all blocks below it in the hierarchy) or the complete system at once. The text only contains actual design information, no graphical (schematics) or current state information. Comments appended to the design elements by the designer are included. Because the documentation system extracts actual information from the design, the comments need only clarify the intended function of the design elements -what they are used for, not what they are. The text is in a 'flat' ASCII format, and is intended to be reworked with a suitable word processor.

### IDaSS filing and library management system

All files used to store blocks and/or signals are text files in a flexible and compact format. They contain not only the design, but also the state of the simulation at the time the file was created and any documentation attached to the system.

# 3 MicroDsp

The MicroDsp is a microcontroller with a 8-bit RISC microprocessor named Peripheral Controller Cell (PCC) combined with an Arithmetic Accelerator Unit (AAU), a Timer Unit (CTU), an $I^2C$ serial bus interface and internal program and data RAM. It should provide a controller function and should include low-end DSP capability for a wide range of applications. The MicroDsp is intended to be a solution suited to many problems as it should provide just the amount of performance required using minimum silicon and supply current. This is, for example, important for portable applications.

A MicroDsp test chip of the Peripheral Controller Cell (PCC) together with the Arithmetic Accelerator Unit (AAU) is to be designed to verify correct implementation of the PCC and AAU blocks, to develop and debug PCC systems and programs and to set up working prototypes of applications implemented with the PCC/AAU combination.

Philips employee J. den Ouden has written the specification [MDSP93] of the MicroDsp. The title of this specification is 'MicroDSP : PCC plus AAU test chip specification 1.1', and it is dated 15 Sept. 1993. This specification is used for the first design of the MicroDsp in IDaSS. During the design phase, modifications have been made on this specification, errors have been erased and new features have been added. This is done by me in cooperation with J. den Ouden. A new specification [MDSP94] was made and it has been used for my final design of the MicroDsp. This specification has version 1.2 and is dated 18 Jan. 1994.

**MicroDsp specification**

The MicroDsp is a microcontroller with the following elements:

- Peripheral Controller Cell (PCC)
- Arithmetic Accelerator Unit (AAU)
- PCC program memory of 4096+64 words of 16 bits
- PCC data RAM of 256 bytes
- 16 bit Capture Timer Unit (CTU)
- 10 I/O ports
- $I^2C$ bus interface and monitor control hardware (IICC)

In Figure 2 the functional block diagram of the MicroDsp emulator/test chip is given.

The 8-bit RISC microprocessor called PCC is the centre block of the MicroDsp. This PCC has an address bus called ADR and a instruction bus called IR. Via these busses the program memory is connected to the PCC. This memory provides the PCC instructions needed to run. Via the IObus the peripherals Arithmetic Accelerator Unit (AAU), Capture Timer Unit (CTU), Inter IC bus interface and Control unit (IICC), and internal data memory (DATARAM) and 9 I/O ports (LIO) are connected. The input and output pins of the chip are given according to the specification in the pinning list below:

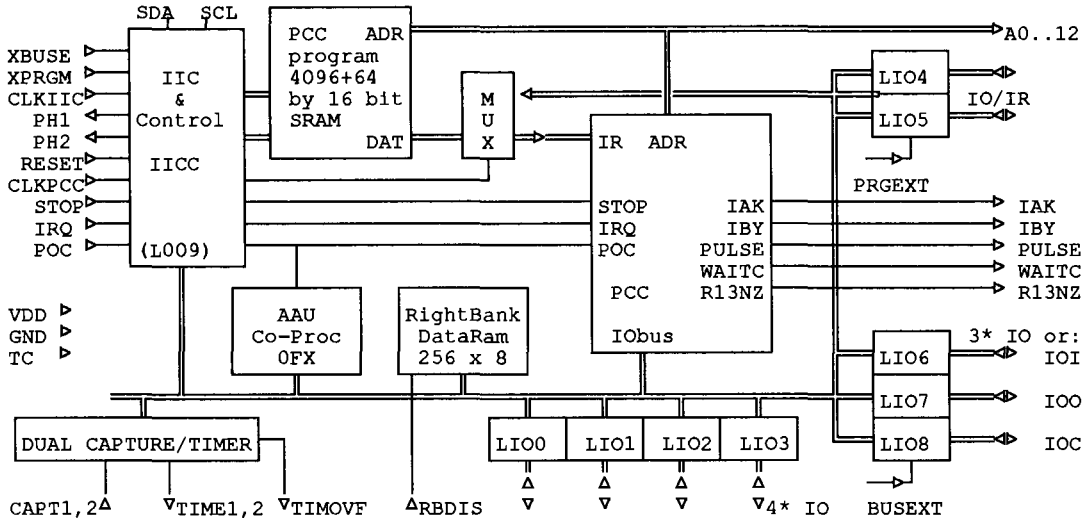


**Figure 2** Functional block diagram of the MicroDsp

## MicroDsp test chip pinning:

| # | pin name | type | description |
|---|---|---|---|
| 4 | VDD,GND | pwr | Minimum number of power pins |
| 1 | TC | in | Test chain Control input |
| 1 | CLKIIC | in | $I^2C$ controller clock |
| 1 | CLKPCC | in | PCC/AAU system clock |
| 2 | PH1, PH2 | out | internal generated PCC system phase clocks |
| 1 | RESET | in | initialize monitor $I^2C$ and control hardware |
| 1 | XPRGM | in | external program enable |
| 1 | XBUSE | in | external IO bus enable |
| 1 | POC | in | PCC Power On Clear |
| 1 | IRQ/TCI1 | in | PCC Interrupt Request / Test Chain In |
| 1 | IAK/TCO1 | out | PCC Interrupt Acknowledge / Test Chain Out |
| 1 | IBY | out | PCC Interrupt Busy |
| 1 | SDA | in/out | $I^2C$ data |
| 1 | SCL | in/out | $I^2C$ clock |
| 1 | STOP | in | PCC STOP control |
| 1 | WAITC | out | PCC WAIT Cycle indicator |
| 1 | PULSE | out | PCC PULSE enable signal |
| 1 | R13NZ | out | PCC R13 Not Zero indicator |
| 8 | LIO0[0..7] | in/out | Left Bank Port 0 |
| 8 | LIO1[0..7] | in/out | Left Bank Port 1 |
| 8 | LIO2[0..7] | in/out | Left Bank Port 2 |
| 8 | LIO3[0..7] | in/out | Left Bank Port 3 |
| 8 | LIO4[0..7] IR[0..7] | in/out | Left Bank Port 4 / instruction input |
| 8 | LIO5[0..7] IR[8..15] | in/out | Left Bank Port 5 / instruction input |
| 8 | LIO6[0..7] IOI[0..7] | in/out | Left Bank Port 6 / IO data in bus |
| 8 | LIO7[0..7] IOO[0..7] | in/out | Left Bank Port 7 / IO data out bus |
| 8 | LIO8[0..7] IOC: | in/out | Left Bank Port 8 / IO bus control |
| | 0 SRE | out | Select Right bank address Enable |
| | 1 SLE | out | Select Left bank address Enable |
| | 2 RRE | out | Read Right bank data Enable |
| | 3 RLE | out | Read Left bank data Enable |
| | 4 WRE | out | Write Right bank data Enable |
| | 5 WLE | out | Write Left bank data Enable |
| | 6 LBDIS | in | Left Bank Disable for internal IO |
| | 7 RBDIS | in | Right Bank Disable for internal IO |
| 13 | A0..A12 | out | PCC Address |
| 1 | TIMEOVF | out | TIME base counter OVerFlow |
| 2 | CAPT1,2/TCI2 | in | Capture inputs, separate test chain inputs |
| 2 | TIME1,2/TCO2 | out | Timer outputs, separate test chain outputs |

The MicroDsp is provided with Test Chain (TC) inputs and outputs. With this, almost all internal flip-flops can be tested. The flip-flops are chained, and the contents is serially transferred to the TCO pins. The MicroDsp has two Test Chain Inputs (TCI1,2) and two Test Chain Outputs (TCO1,2).

The internal and external IO transfers are done via two IO busses, IOI and IOO. The IOC bus is used to control the IO transfers. IO has been partitioned into two banks : Left Bank IO and Right Bank IO. If an IO device should be accessed, first the IO device should be selected with SLE or SRE combined with an IO address. Later data can be transferred using RLE, RRE, WLE or WRE. LBDIS and RBDIS are used to disable the internal IO. In this way external IO can be put on the same address as internal IO devices.

Clock timing for the MicroDsp is specified in the PCC specification [PCC93]. There are two non-overlapping clocks used: PH1 for the input phase and PH2 for the output phase. In the IDaSS design it was not possible to use seperate input and output phases because IDaSS only has one synchronisation point for both transactions.

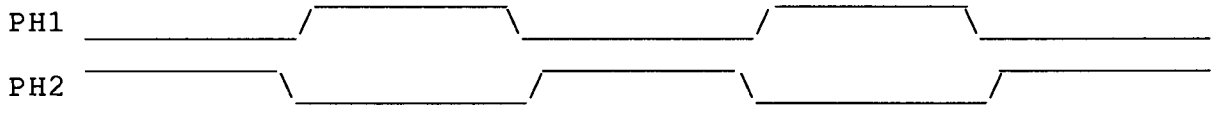


**Figure 3** Two non-overlapping clocks PH1 and PH2

The MicroDsp is provided with two reset pins : RESET and Power On Clear (POC). The RESET pin is used for the reset of the $I^2C$ and control part of the MicroDsp. The POC is used to reset the PCC, AAU and Timer and to Initialize the RAMs and IO.

For the communication with a remote PC, two serial IO pins are provided: SDA and SCL. These pins are specified in the $I^2C$ specification [$I^2C$92]. SDA is the $I^2C$ serial data line, SCL is the $I^2C$ clock line.

### MicroDsp design

Designing the MicroDsp toplevel in IDaSS is done after the design of the elements of the MicroDsp was completed. Because a lot of connectors and signals were already specified, it was simple to connect the blocks. In Figure 4 the IDaSS design of the MicroDsp toplevel is given. In this figure, you will find the following elements:

- IICC
- PCC
- AAU
- CTU
- AMUX
- DMUX
- PRAM
- DRAM
- LIO

The most important internal signals used to connect these blocks are:

| name | bits | description |
|---|---|---|
| • sle | 1 | select left bank enable |
| • wle | 1 | write left bank enable |
| • rle | 1 | read left bank enable |
| • sre | 1 | select right bank enable |
| • wre | 1 | write right bank enable |
| • rre | 1 | read right bank enable |
| • ioo | 8 | io output bus |
| • ioi | 8 | io input bus |
| • lbdis | 1 | left bank disable |
| • rbdis | 1 | right bank disable |
| • poc | 1 | power on clear |
| • irq | 1 | interrupt request |
| • stp | 1 | stop |

There are more signals necessary to connect all blocks. In the next paragraphs these signals and blocks will be described in detail.

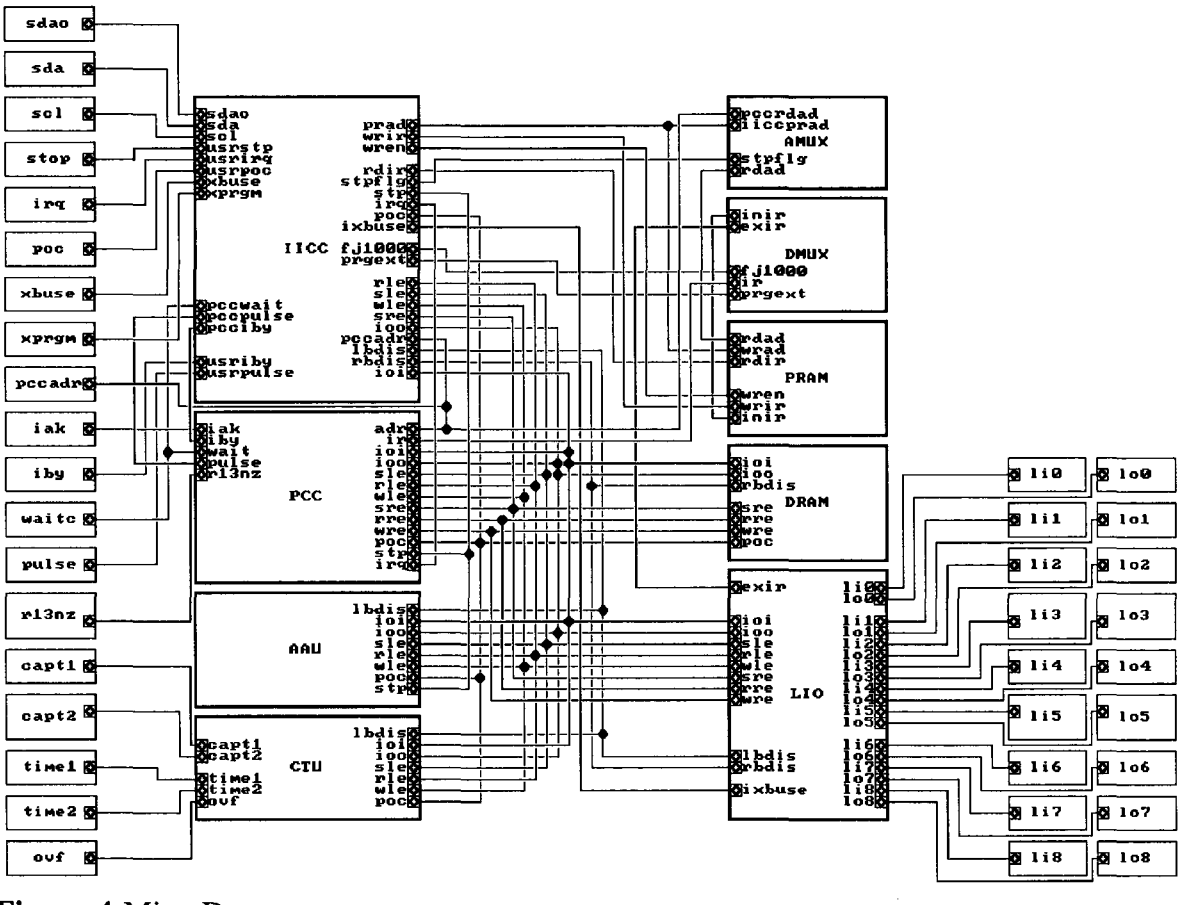


**Figure 4** MicroDsp

## 3.1 Peripheral Controller Cell (PCC)

The Peripheral Controller Cell (PCC) provides a very fast processor function optimised for bit field handling and quick event response. The PCC is based on the 8X305 RISC architecture, which is a proven concept for fast bit oriented control. Low transistor count makes a very small core design possible. Harvard structure, execution of most instructions in a single cycle of the two phase non-overlapping clock and keeping program memory, data and I/O accesses on-chip, mean that performance can be very high, mainly limited by program memory speed. A full custom design in C200DM technology has been realised (size : 1.06 mm2, worst case performance of 16 MIPS at 5 Volt, full static design, suitable for low voltage operation). An IDaSS design has to be made to be able to simulate the PCC with peripherals on a high level. A VHDL implementation has to be made to be able to synthesize the PCC with the use of a silicon compiler.

The basic architecture has been extended with an interrupt facility that has a one clock cycle interrupt latency and performs some context switching. Conditional jump execution is delayed by one clock wait cycle to allow use of relatively slow program memory. An accessory ROM generator is also available. PCC assembler and simulator are running on IBM-PC and Apollo.

PCC Features according to the Philips specification:

- Fetch, decode, execute 16 bit instructions in single 2 phase clock cycle
- Bit oriented instruction set (addressable single/multiple bit sub-fields)
- Separate busses for instruction, instruction address and 3-state IO
- Thirteen 8 bit general purpose working registers
- Source/destination architecture
- Single clock latency for interrupt, including context switching
- Wait cycle for conditional jumps (relaxed program memory spec)
- 8 kword (16 kbyte) program memory space
- Full static CMOS implementation
- Very small processor block size
- Very low supply current per MIPS
- Designed for testability (minimal)

### PCC specification

The PCC is a high speed full static CMOS 8 bit micro-control RISC with a bit-oriented instruction set. The PCC has a Harvard structure and can fetch, decode and execute a 16 bit instruction in a single clock cycle (PH1,PH2).

Within one instruction cycle the 8 bit data processing path can be programmed to rotate, mask, shift and/or merge single or multiple bit sub-fields and, in addition, perform an ALU operation; in the same instruction an external data field can be input, processed and output to a specified destination. Likewise, single or multiple bit sub-fields can be moved from a given source to a given destination. To summarize : fixed or variable length data fields can be fetched, processed, operated on by the ALU and moved to a different location, all in a single two phase clock cycle.

To enable fast response on asynchronous hardware events without loss of performance an interrupt facility is provided which has a single clock response latency and which includes a context switch. Program memory receives a 13 bit address to access 16 bit instructions allowing a program of up to 16 kbyte. Data I/O is done via a separate 8 bit bi-directional three-state bus that multiplexes I/O data and addresses under control of 6 I/O control enables. The I/O bus also can be split in two separate busses, one for input and one for output. In-line PULSE instructions and the PULSE output provide an extra facility for fast control of user hardware without affecting program context.

When calculating a conditional program jump address (instructions XEC and NZT) the PCC inserts a wait cycle so that the program address can always be provided early in the memory access cycle. This allows use of relatively slow program memory. (This mode can be omitted for slower clocks).

Internal data registers

The PCC has 18 internal data registers, nine 8-bit registers can freely be used and nine others have (also) a special purpose. All register names are pre-declared assembler variables.

| register | purpose |
|---|---|
| R0\|R0I=AUX | implicit second argument for ALU operations (ADD, AND, XOR) |
| R1..R6 | 6 general purpose 8-bit data registers |
| R7=IOL | 8-bit register, copies Left Bank IO address in main program context. R7 is automatically re-transmitted on interrupt return. |
| R10=OVF | carry of the last add operation. OVF cannot be destination. OVF uses a separate flag in interrupt context. |
| R11 | used by the PCC assembler for subroutine return table index |
| R12,R13 | general purpose 8-bit data registers.<br>XMIT to R12, R13 sends a data byte to the Left and Right Bank ports, not to the register. (see XML, XMR instructions) |
| R14..R16 | 3 general purpose 8-bit data registers |
| R17=IOR | 8-bit register, copies Right Bank IO select address in main program context |

AUX has a separate interrupt context register and for both normal and interrupt context there is a shadow register in the ALU. The shadow register data is used as implicit data for ADD, AND and XOR operations. Both AUX registers and their ALU shadow registers are cleared at POC. The interrupt context AUX contents, but not its ALU shadow register, can be modified by moving data to R0I "MOVE Rn,R0I" (Note that this data only is used in interrupt context if AUX is specified as source data).

In main program context R7 and R17 are automatically written if IO addresses are selected, in interrupt context a SELect of IO does not affect R7 or R17. On interrupt return R7 is automatically re-transmitted to restore the main context Left Bank select address. If IOR is changed by the interrupt routine R17 has to be restored using MOVE R17,IOL before executing interrupt return.

R12 and R13 are used for the XML and XMR special XMIT instructions that send an 8 bit data byte to Left and Right bank IO ports. Internal registers R12 and R13 therefore cannot be loaded directly using XMIT.

If R13 contains a non-zero value the R13NZ output goes HIGH. This feature can be used to speed up and reduce code in "compare-and-set bit" operations with data and IO flags.

IO data bus

IO bus ports and/or data RAM are connected to the 8 bit 3-state user bus. A RAM block is commonly addressed via the Right Bank, while single ports reside on the Left Bank. To enable access of a RAM location or a port it can be selected by sending the port address on the bus using XMIT, MOVE, etc to IOR=R17 or IOL=R7 or the assembler directive SEL with an RIO or LIO address. These instructions put the 8 bit address value on the bus and assert the Right or Left Bank select enable. A RAM block latches the complete Right Bank select address, while a port that detects its address sets its select flag. A bank address or port select flag is valid until a new address is transmitted on the Bank it is assigned to. Port 000X on the Left Bank is automatically selected on reset (POC='1') and on interrupt acknowledge (IAK='1'), so it can be used for very quick interrupt service and/or as interrupt vector address.

Reading port data takes place in the PCC input phase (PH1), while address selects and data (re-)writes are done on the output phase (PH2). When accessing data on the IO bus, a bit field can be defined with a length of 1 to 8 bits and a LSBit position from 0 to 7. (If Length plus LSB position overflows the byte size, the bit field is truncated).

Note that the whole byte of an IO port is read in the input phase (PH1) and copied, even if only a part of the port bits is affected by the instruction, so that the modified bit field can be merged in the PCC and the complete byte can be written back in the output phase (PH2).

Port registers can be simple latches as the PCC performs the master flip-flop function (on PH1). Port latches are written on PH2, so port output latches can change during PH2.

PCC Instruction set

The PCC is a highly Reduced Instruction Set Controller (RISC) having only 8 basic 16 bit instructions most of which can be executed with a single clock. Four instructions operate on data (MOVE, ADD, AND and XOR), two conditional jumps (XEC, NZT) are available for program control and table handling, one instruction provides data constants (XMIT) and one makes jumps over the whole address range (JMP).

Data operation instructions

ALU data operation instructions MOVE, ADD, AND and XOR specify source and destination, rotation, or bit field position and length. Within one data operation instruction cycle, the 8 bit data processing path can be programmed to rotate, mask, shift and/or merge single or multiple bit sub-fields, and perform an ALU operation. AUX is implied argument for ADD, AND, OR and XOR operations. Increment (INC), complement (NOT) and inclusive or (OR) are single argument instructions : specified data source (register or IO bit field) is also destination. INC, NOT, OR, the four instructions for interrupt control and the PULSE instruction, all indicated with '+', are modified basic instructions that use OVF (not writable) as dummy destination for their code. MOVE Rn,R0I from register Rn other then R7 writes to the AUX register of the interrupt context, using OVF as destination code.

| | | |
|---|---|---|
| 0 MOVE | Source to Destination<br>used for data transport between register(s) and IO port(s) | |
| + DSI | clears interrupt enable flag | (MOVE R7(0),OVF) |
| + ENI | set interrupt enable flag (cleared at POC, IAK) | (MOVE R7(1),OVF) |
| + RTI | return from interrupt | (MOVE R7(2),OVF) |
| + RTE | return from interrupt set interrupt enable flag | (MOVE R7(2),OVF) |
| 1 ADD | Source + AUX --> Destination, update OVF<br>provides arithmetic. | |
| + INC | Source/Destination + 1 --> Source/Destination<br>increments data without using AUX or OVF | (ADD Rx(R),OVF)<br>(ADD IOx(L),OVF) |
| 2 AND | Source .and. AUX --> Destination<br>can be used to isolate data bits and evaluate logic functions. | |
| + OR | Source/Dest. .or. AUX --> Source/Destination<br>inclusive or of S/D with AUX | (AND Rx(R),OVF)<br>(AND IOx(L),OVF) |
| 3 XOR | Source .exor. AUX --> Destination<br>contributes to logic evaluations and arithmetic | |
| + NOT | Source/Dest. .exor. FF --> Source/Destination<br>Complements data without using AUX. | (XOR Rx(R),OVF)<br>(XOR IOx(L),OVF) |

Conditional jump instructions XEC and NZT

"Conditional" jump instructions XEC and NZT calculate a jump address on basis of a data source field. As source data access starts at the beginning of the instruction cycle a new address would be valid much later than for other instructions (sequential instructions start address increment in the previous cycle, JMP provides the full new address in the instruction). To prevent that XEC and NZT instructions reduce maximum operating speed a wait cycle is inserted, so that new addresses are valid early in a program memory access cycle for all instructions allowing relatively slow ROMs or long address setup times.

Conditional jump instructions XEC and NZT calculate new address bits to replace the lower 8 bits if source is register, or 5 bits if source is IO, of the program counter. For correct operation the target address must therefore be in the current 5 or 8 bit page. The PCC assembler ORG directive supports conditional page alignment.

4 XEC execute in page at (Displacement + SRC)

XEC provides indexed execution (on register or IO port), it is very powerful for constant table access and for indexed jump tables useful for subroutine return, vectored interrupt handling as well as state machine service. XEC replaces 8 bits or 5 bits in the program counter (before increment) and executes the instruction at the calculated address. If this instruction is not an accepted jump only one instruction is done and the program continues with the instruction following the XEC. If the XEC executes a JMP or a NZT (source non zero) the program continues at the jump address. XEC instructions may execute chained XEC instructions.
If an instruction executed by a XEC is aborted by an interrupt the XEC itself will also be re-executed after interrupt return. XEC instructions always insert a wait cycle.

5 NZT jump in page if Source not zero

Non-Zero-Transfer is used for program flow control and signal polling. If the source data is non-zero the NZT instruction replaces 8 bits (if source is register) or 5 bits (if source is IO bit field) in the program counter after incrementing. If the NZT source data is zero the program counter is simply incremented. NZT instructions always insert a wait cycle.

Load literal instruction XMIT

6 XMIT Constant --> Destination loads data into specified destination
if register : 8 bit literal, if IO bit field : 5 bit literal

+ XML send 8 bit literal to IO Left Bank (uses R12 as dummy destination)
+ XMR send 8 bit literal to IO Right Bank (uses R13 as dummy destination)
+ PULSE send 8 bit literal to IO bus, assert PULSE line

Unconditional branch instruction JMP

The unconditional JMP instruction enables jumps over the full address range.

7 JMP jump to Address

## PCC design

The PCC has been designed in IDaSS by B. Vostermans and P. Simons. They are employees of the Eindhoven University of Technology. Their design is given in Figure 5.

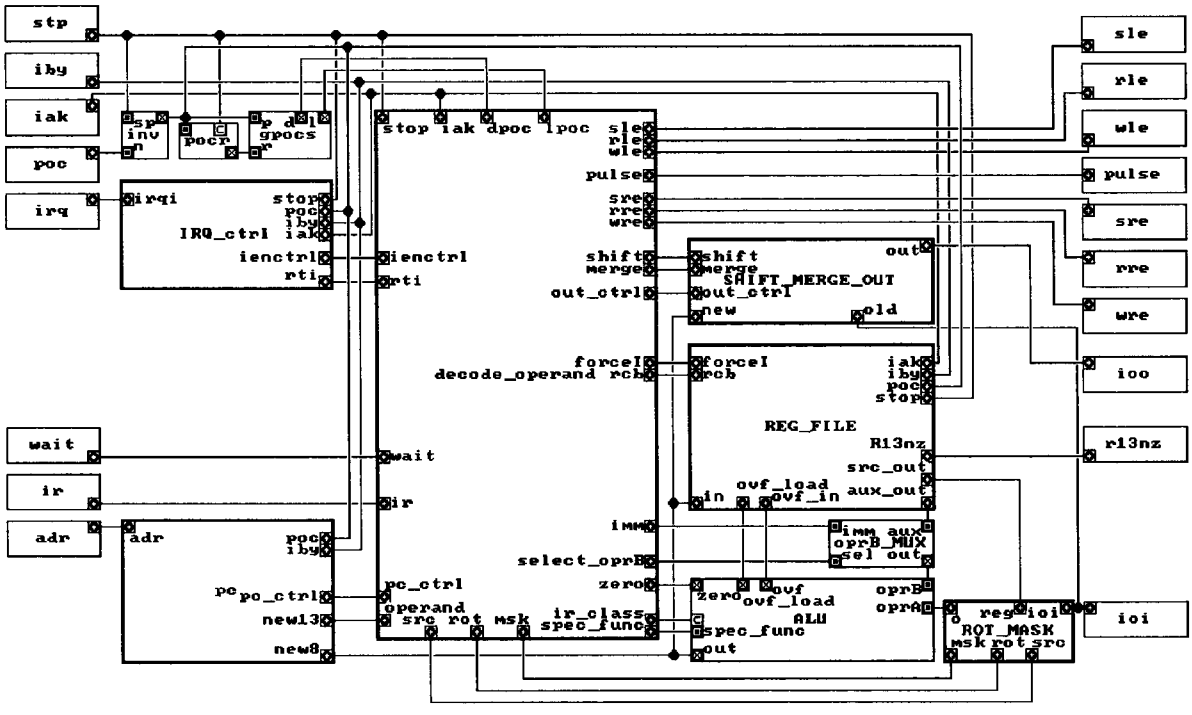


**Figure 5** PCC

## PCC simulation

The IDaSS PCC has been simulated by B. Vostermans and P. Simons. According to them the IDaSS implementation is corresponding the specification [PCC93] of the PCC. However, this has not been verified with the special PCCTEST program. The IDaSS PCC (version 2) is used in the MicroDsp toplevel schematic.

## 3.2 Arithmetic Accelerator Unit (AAU)

The Arithmetic Accelerator Unit (AAU), a coprocessor for the PCC, is designed to speed up otherwise tedious byte-wise multiply/divide and shift operations for calculations with signed or unsigned integers in floating or fixed point data formats.

The AAU features signed (four-quadrant) and un-signed multiplication, division, normalise and adjust, in short format (both operandi 16 bits) and long format (operandi 16 and 32 bits). Short integer (16 bit) signed and unsigned multiplication also can be done while accumulating with previous result(s). In this mode overflow and underflow are detected and resulting values are clipped automatically to their appropriate maximums to prevent control calamities without any software overhead. The multiply-accumulate feature is very powerful for digital signal processing in control loops. Normalise and shift facilities of the AAU enable to implement relative fast floating point calculation routines.

**AAU specification**

AAU data registers X, Y, Z, CMD and STA are located in PCC IO 0xF0..0xFF. Starting access on addresses 0xF0 or >0xFA provides Burst Mode Access incrementing IO addresses automatically at consecutive accesses. The AAU allows "chained operations" : intermediate results of formula evaluation may be left in AAU accumulator registers for re-use in next operations. Both features increase performance while reducing PCC code required.

Operations basically require one clock for initialise, 16 or 32 AAU clocks for execution and one to four AAU clocks for finishing. The PCC program may continue with other business while the AAU is BUSY.

| operation | clocks for word | for long |
|---|---|---|
| Multiply | 18 | 34 |
| Divide | 19 | 36 |
| shift | n+1,<=17 | n+1,<=33 |

Together with the PCC the AAU provides a capability to do low end DSP calculations like about 640.000 digital filter stage calculations (multiply/accumulate/clip) per second (for C3DM : at 16 MHz clock).

Features :
- . multiply 16*16 and 16*32 bits
- . divide 16:16 and 32:16 bits (with overflow detection)
- . multiply 16*16 accumulate (32 bits) with overflow clipping
- . Normalise (16/32 bit) for floating point operations
- . Shift left/right (16/32 bits) over 0..15/31 bits
- . All operations un-signed and signed, four quadrants
- . Results within 19 (short) or 36 (long) PCC clock cycles (worst case)
- . Chained operations may be used to reduce data moves
- . Burst Mode Access for optimal performance and reduced code
- . PCC may proceed during AAU operation

## AAU design

The AAU has been designed in IDaSS by A. Verschueren. This AAU is different from the handcraft AAU. In the IDaSS design most calculations are computed 2 cycles faster. Another difference is that the IDaSS AAU has a POC input while in the original design the POC line has to be connected to the STOP input. Furthermore, the handcraft AAU also provides a BUSY output while the IDaSS AAU does not have that output. The IDaSS design is given in Figure 6.

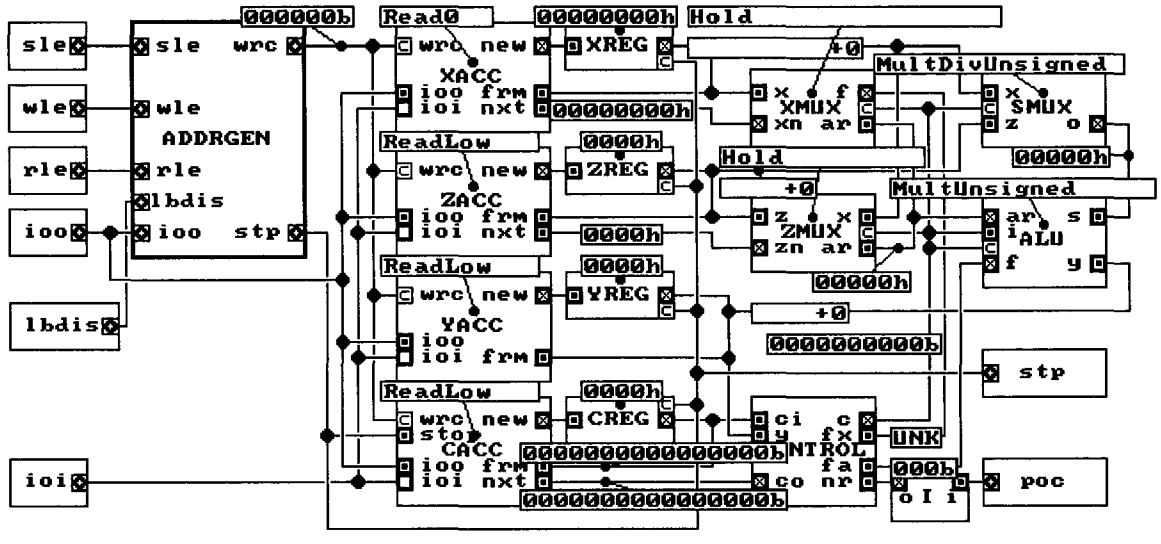


**Figure 6** AAU

## AAU simulation

The IDaSS AAU has been simulated by A. Verschueren. The first simulation gave an error for the division -18/3. This gave quotient -5 with remainder -3, witch is of course the same as quotient -6 with remainder 0, what should have been the answer. This error was a result of a wrong algorithm that was used for the divide action. This algorithm was also used for the handcraft AAU that was already implemented. In both designs this error then had to be erased. According to Verschueren the IDaSS implementation is not exactly corresponding the specification [AAU93] of the AAU because of differences in timing and pinning. On the other hand, the functional behaviour is according to the specification. This AAU is used in the MicroDsp toplevel schematic.

## 3.3 I²C interface and Control (IICC)

The MicroDsp can be controlled by a "host" computer like a Personal Computer (PC) via an on chip I²C interface (I²C = Inter-IC). Communication is done via a serial two wire link. This means that a monitor program that is running on the PC can influence the progress of the program running on the MicroDsp. For example, the monitor program can start and stop the MicroDsp or read IO bus information or read the contents of a register of the PCC. The I²C device in the PC is called the master, and the I²C device on the MicroDsp is called the slave.

The MicroDsp has to have the possibility to download PCC instructions into the program memory. For small PCC monitor programs an additional memory segment is available. Via I²C control the PC can start and stop PCC program execution and access control and status ports. In this way the PCC emulator running on the PC can access the PCC. All system information can be displayed on the screen and all writable ports can be modified from the keyboard.

Control hardware on the MicroDsp chip is kept to a minimum by locating most of the "intelligence" in the controlling monitor program of the PC, this in order to reduce design complexity and design risk while preserving maximum flexibility.

### IICC specification

For the MicroDsp I²C interface a number of I²C addresses is reserved without looking at eventual conflicts with existing I²C devices. These are used to address MicroDsp control ports. The following ports are provided:

| addr | name | R/W | functional description |
|---|---|---|---|
| 10 | FCTL | R&W | External signal override controls |
| 11 | PADH | R | PCC ADR[8..12] state on last instruction |
| 12 | PADL | R | PCC ADR[0..7] state on last instruction |
| 13 | LBAD | R | PCC IO last Left Bank address |
| 14 | RBAD | R | PCC IO last Right Bank address |
| 15 | IODT | R | PCC IO last IOO data |
| 16 | MADH | R&W | PCC program memory address 5 upper bits |
| 17 | MADL | R&W | PCC program memory address 8 lower bits |
| 18 | MDTH | R&W | PCC program memory data 8 upper bits |
| 19 | MDTL | R&W | PCC program memory data 8 lower bits |
| 1A | XCTL | R&W | MicroDsp start control |
| 1B | STAT | R | MicroDsp status |
| 1C | MAIL | R&W | MicroDsp debug port |

The following PCC system data access and control functions must be available from the hardware for the PC monitoring/debugging program:

| data access and control functions | unit |
|---|---|
| PCC program memory words Read and Write | MADH/L, MDTH/L |
| PCC start, stop, breakpoints and single step | FCTL, XCTL |
| Memory and IO internal/external control | FCTL, XCTL |
| PCC internal registers R0..R7, OVF, R11..R17 | IODT |
| Last PCC address (interrupt return address) | PADH/L |
| Last run time Left and Right bank address | LBAD,RBAD |
| Last Left and Right bank data | IODT |
| PCC state and user control lines access | STAT |
| Interrupt context data: AUXI, OVFI access | FCTL, XCTL, IODT |
| Interrupt enable flag ENI access | FCTL, XCTL, IODT |
| Program data | MAIL |

A functional block diagram of the IICC is given in Figure 7.

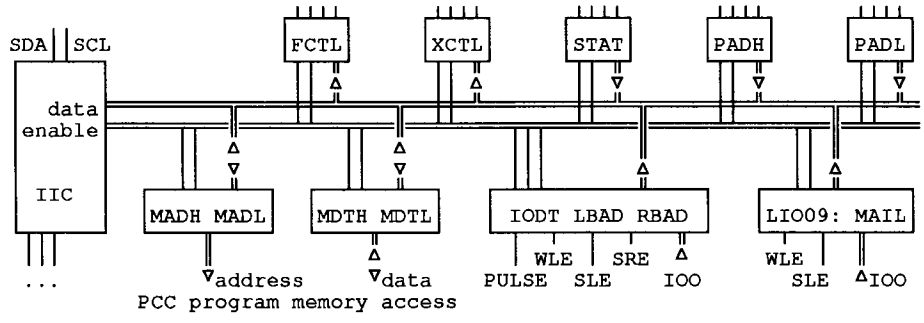


**Figure 7** Functional block diagram of IICC

## IICC design

In the IDaSS design of the IICC unit, we find the following elements:

- inter IC control unit (IIC)
- program memory access unit (MADT)
- force control unit (FCTL)
- address state unit (PAD)
- IO view unit (IOBUS)
- start control unit (XCTL)
- status unit (STAT)
- mail unit (MAIL)
- program type determination unit (PRGTYPE)
- monitor control unit (MONCTRL)
- stop control unit (STOPCTRL)

The IIC unit is the centre of the total IICC design. Globally, this block is a serial to parallel converter. It converts the serial information that is received from a PC host into bytes that can be read by the surrounding blocks such as MADT, FCTL etc. On the other hand if the PC

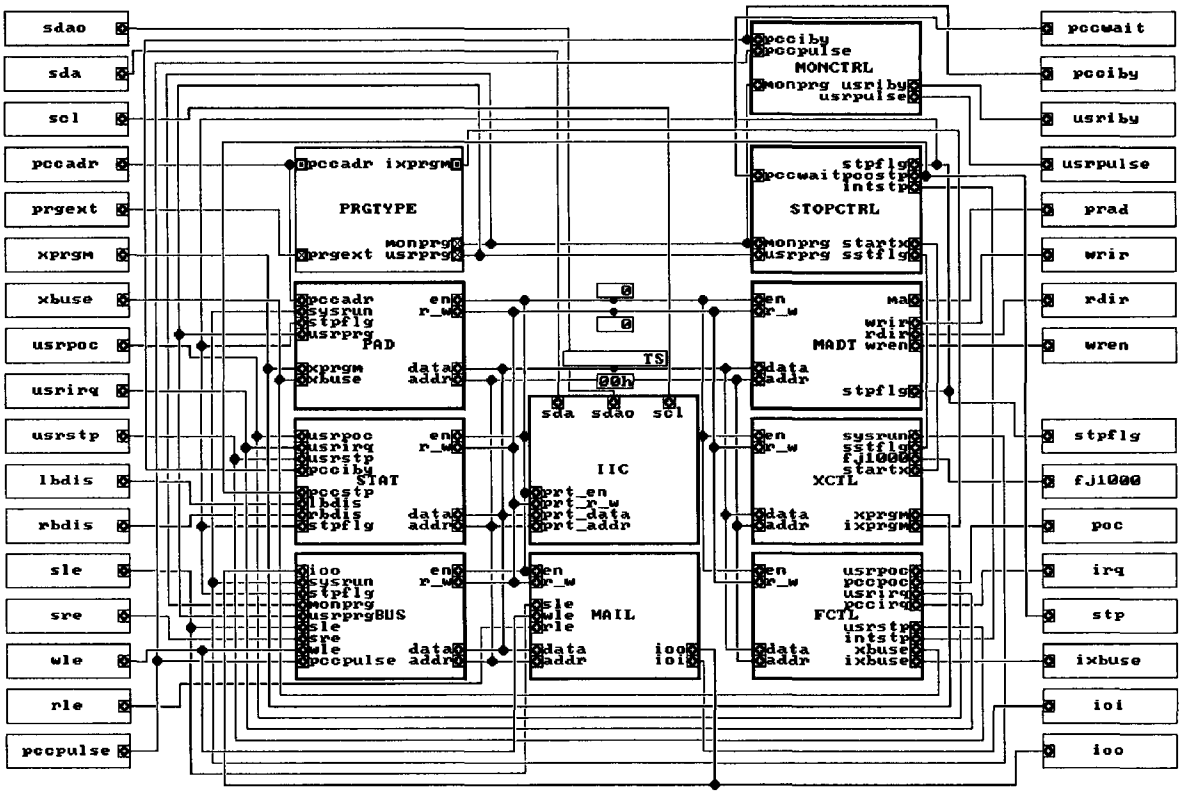


**Figure 8** IICC

host has to read something from one of the IICC subblocks, these bytes will be coded into a serial bit stream, that will be transferred to the PC host. In the list below all signals are described.

External signals:

| name | bits | description |
|---|---|---|
| • sda | 1 | serial data in |
| • sdao | 1 | serial data out |
| • scl | 1 | serial clock |
| • ioi | 8 | io input bus |
| • ioo | 8 | io output bus |
| • sle | 1 | select left bank enable |
| • wle | 1 | write left bank enable |
| • rle | 1 | read left bank enable |
| • sre | 1 | select right bank enable |
| • lbdis | 1 | left bank disable |
| • rbdis | 1 | right bank disable |
| • rden | 1 | read program ram enable |
| • wren | 1 | write program ram enable |
| • prad | 13 | program ram address |
| • rdir | 16 | read instruction bus |
| • wrir | 1 | write instruction bus |
| • pccwait | 1 | pcc wait |
| • pccadr | 13 | pcc address bus |
| • fj1000 | 1 | force jump to 1000h |
| • stpflg | 1 | stop flag |
| • stp | 1 | stop |
| • usrstp | 1 | user stop |
| • poc | 1 | power on clear |
| • usrpoc | 1 | user driven poc |
| • irq | 1 | interrupt request |
| • usrirq | 1 | user interrupt request |

```
        • usrpulse          1           user pulse
        • pccpulse          1           pcc pulse

        • usriby            1           user interrupt busy
        • pcciby            1           pcc interrupt busy

        • prgext            1           external program
        • xprgm             1           user external program
        • ixbuse            1           external bus enable
        • xbuse             1           user external bus enable

Internal signals:

        • addr              8           internal address bus
        • en                1           internal enable signal
        • r_w               1           internal read/!write signal
        • data              8           internal bidirectional data bus
```

The sda and sdao lines have to be connected to each other, but because some additional logic is necessary to prevent metastability, these signals are separated input and output ports. The metastability circuit will be described in VHDL.

In the next paragraphs all IICC elements will be described in detail.

### 3.3.1 Inter-IC (IIC)

Philips PCALE employee H. Schutte has developed a simple bidirectional 2-wire bus for efficient inter-IC control. This bus is called the Inter IC or $I^2C$-bus. An $I^2C$-bus compatible device incorporates an on-chip interface which allows it to communicate directly with every other device on the $I^2C$-bus. Some features of the $I^2C$-bus are:

- Two bus lines are required; a serial data line (SDA) and a serial clock line (SCL)
- Each device connected to the bus is software addressable by a unique address
- Serial, 8-bit oriented, bidirectional data transfers can be made at up to 100 kbit/s in the standard mode or up to 400 kbit/s in the fast mode
- On chip filtering rejects spikes on the bus data line to preserve data integrity

The $I^2C$-bus supports any IC fabrication process (NMOS, CMOS, bipolar). Two wires, serial data (SDA) and serial clock (SCL), carry information between the devices connected to the bus. Each device is recognised by a unique address and can operate either as transmitter or receiver, depending on the function of the device.

#### IIC Data validity

The data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW.

#### IIC START and STOP conditions

Within the procedure of the $I^2C$-bus, unique situations arise which are defined as START and STOP conditions. A HIGH to LOW transition on the SDA line while SCL is HIGH is one such unique case. This situation indicates a START condition. A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition. Detection of START and STOP conditions by devices connected to the bus is easy if they incorporate the necessary interfacing hardware. However, microcontrollers with no such interface have to sample the SDA line at least twice per clock period in order to sense the transition.

#### IIC Transferring data

Every byte put on the SDA line must be 8-bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an acknowledge bit. Data is transferred with the most significant bit (MSB) first. If a receiver can't receive another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the transmitter into a wait state. Data transfer then continues when the receiver is ready for another clock line SCL.

Data transfer with acknowledge is obligatory. The acknowledge related clock pulse is generated by the master. The transmitter releases the SDA line (HIGH) during the acknowledge clock pulse. The receiver must pull down the SDA line during the acknowledge clock pulse so that it remains stable LOW during the HIGH period of this clock pulse. Set-up and hold times must also be taken into account. Usually, a receiver which has been addressed is obliged to generate an acknowledge after each byte has been received. When a slave-receiver doesn't acknowledge the slave address (for example, it's unable to receive because it's performing some real-time function), the data line must be left HIGH by the slave. The master can then generate a STOP condition to abort the transfer.

If a slave-receiver does acknowledge the slave address but, some time later in the transfer cannot receive any more data bytes, the master must again abort the transfer. This is indicated by the slave generating the not acknowledge on the first byte to follow. The slave leaves the data line HIGH and the master generates a STOP condition.
If a master receiver is involved in a transfer, it must signal the end of data to the slave-transmitter by not generating an acknowledge on the last byte that was clocked out of the slave. The slave-transmitter must release the data line to allow the master to generate the STOP condition.

### IIC Formats with 7-bit addresses

Data transfers follow the format shown in Figure 9. After the start condition (S), a slave address is sent. This address is 7 bits long followed by an eighth bit which is a data direction bit (R/!W) -a 'zero' indicates a transmission (WRITE), a 'one' indicates a request for data (READ). A data transfer is always terminated by a stop condition (P) generated by the master. However, if a master still wishes to communicate on the bus, it can generate a repeated START condition (Sr) and address another slave without first generating a STOP condition. Various combinations of read/write formats are then possible within such a transfer. Possible data transfer formats are:

- **Master-transmitter transmits to slave-receiver. The transfer direction is not changed (Figure 10).**
- **Master reads slave immediately after first byte (Figure 11).** At the moment of the first acknowledge, the master-transmitter becomes a master receiver and the slave-receiver becomes a slave transmitter. This acknowledge is still generated by the slave. The STOP condition is generated by the master
- **Combined format (Figure 12).** During a change of direction within a transfer, the START condition and the slave address are both repeated, but with the R/W bit reversed.

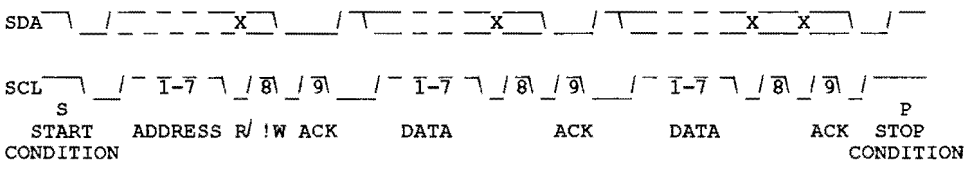


**Figure 9** A complete data transfer

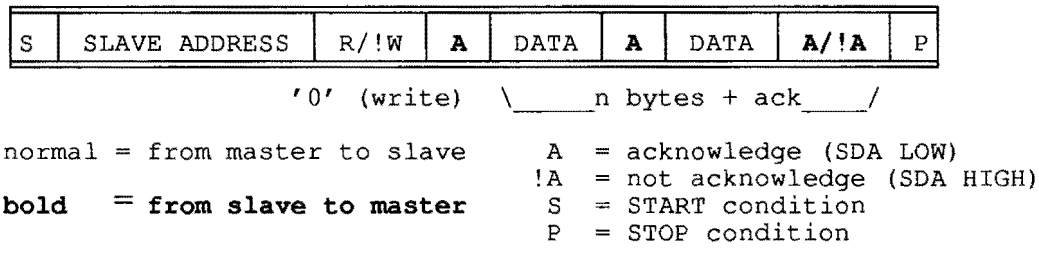


**Figure 10** A master-transmitter addresses a slave receiver with a 7-bit address. The transfer direction is not changed

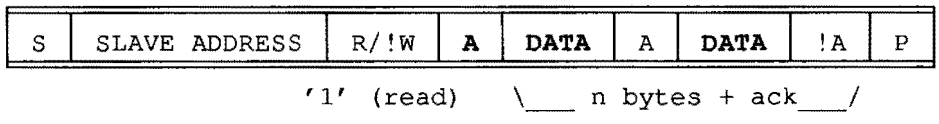


**Figure 11** A master reads a slave immediately after the first byte

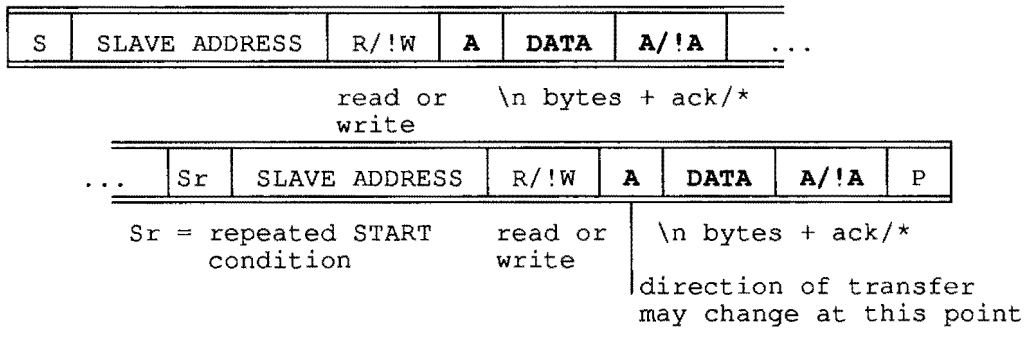


**Figure 12** Combined format

## IIC specification

It is advised that the PC to MicroDsp interface is a point to point interface that exists of a single master located in the PC and a single slave located on the MicroDsp. For a convenient program trace mode or commands like "step n instructions" $I^2C$ data transfers must be done as quickly as possible, so the 400 kbit/s $I^2C$ specification is preferred.
The PC $I^2C$ master issues all START and STOP conditions, generates the $I^2C$ clock SCL and

sends I²C address bytes including the Read/Write control bit. At the end of byte output the master can check reception of the acknowledge to ensure that the connected MicroDsp slave is responding. On reading I²C data the master will also provide an acknowledge according to the I²C specification. This acknowledge will be ignored by the slave. As most MicroDsp registers are writable and readable a more extensive I²C interface test can be done at the start of the PC monitor program.

Host computer program command sequences start by sending an I²C start (S) condition and a new I²C address that points to the MicroDsp port which is the first source or destination for I²C data byte. The MicroDsp Control hardware automatically advances to the next I²C port addresses. Data transfers can be ended at any time with an I²C stop condition (P) from the PC master I²C driver. While the PCC is running I²C control is only practical to read the MicroDsp status (from STAT port), debug data (from MAIL port) and to control write (to port FCTL) to stop the PCC. While in stop mode all I²C actions described below can be used. I²C sequences as transmitted by the PC controlling program will be described here using the following conventions, appearance and order:

| | | |
|---|---|---|
| | 1. S | force I²C start condition |
| | 2. portname | I²C port address (7 bits) |
| | 3. +R or +W | read / write control bit in the address byte |
| | 4. (*A*) or (*!A*) | address acknowledge level resp. ACK or NAK from MicroDsp |
| for N | 5. [data] or [*data*] | data transfer (8 bits) resp. to or from MicroDsp |
| bytes | 6. (A) or (!A) or (*A*) or (*!A*) | data acknowledge level resp. ACK or NAK resp. to or from MicroDsp |
| | 7. P | force I²C stop condition |

**IIC design**

The IIC unit is the connection between the host computer and the MicroDsp. All inside information that is captured in the IICC register units can be transferred to the host computer via the serial SDA and SCL lines. Therefore the IIC unit has to be able to read and write these registers. A special internal data bus DATA and address bus ADDR have been made to perform these transfers. A read/write line R_W indicates if such a register has to be read or written. An enable line EN determines if the busses are valid.

The IIC unit has four registers that indicate the condition of the data transfer. The registers S and P indicate the START and STOP conditions and the registers UP and DN indicate the state of the SCL line, thus the validity of the SDA data. This information is used by a state-controller CTRL which controls the serial-parallel converter SP_CONV and the registers BIT_CNT, ADDRESS, DATA and R_W.

The SDA_DEL and SCL_DEL registers are used to delay the SCL and SDA signal one bit in order to detect if the SDA and SCL lines change from '1' to '0' or from '0' to '1'. (In the final VHDL design 3 or more clocks delay are necessary to avoid metastability but for the functional simulation this delay is not needed).

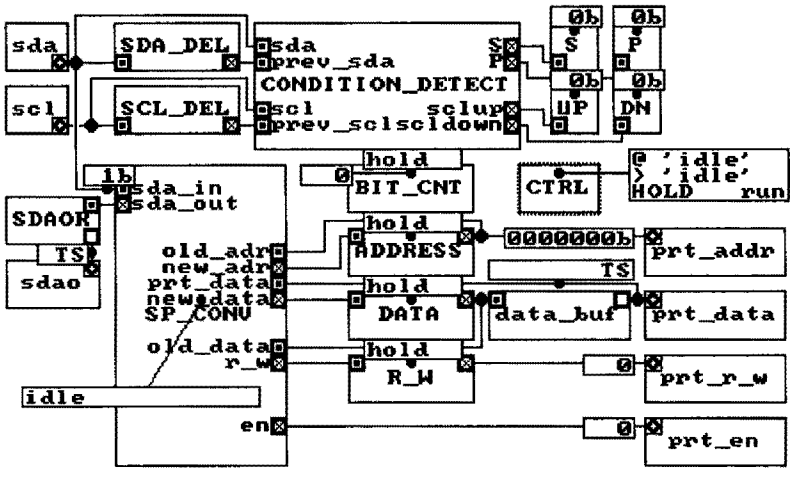


**Figure 13** IIC

```
'SDA_DEL' is a register.

This register is 1 bit wide.
The default function is 'load'.
This register is loaded with value 0 following system reset.

The value loaded for the 'reset' command is 0.
```

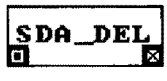


**Figure 14** SDA_DEL register

```
'SCL_DEL' is a register.

This register is 1 bit wide.
The default function is 'load'.
This register is loaded with value 0 following system reset.

The value loaded for the 'reset' command is 0.
```

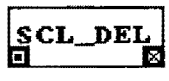


**Figure 15** SCL_DEL register

The CONDITION_DETECT operator is used to determine the next state of the state-controller. Every next state depends on the behaviour of the SDA and SCL lines. The condition detect operator can detect the specified START and STOP conditions and the state of the SCL line (HIGH or LOW).

```
'CONDITION_DETECT' is an operator.

This operator has 1 function.
The default function is 'S_P_det'.

Text for function 'S_P_det' of 'CONDITION_DETECT':
----------------------v----------------------
S := prev_sda /\ prev_scl /\ (sda not) /\ scl.
P := (prev_sda not) /\ prev_scl /\ sda /\ scl.
sclup := (prev_scl not) /\ scl.
scldown := prev_scl /\ (scl not).
----------------------^----------------------
```

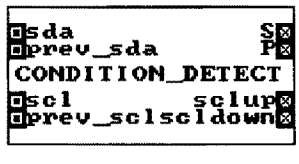


**Figure 16** CONDITION DETECT operator

A state controller CTRL is used to control the serial parallel converter SP_CONV and to control the registers ADDRESS, DATA and R_W. In this way serial bits received at the SDA

input line will be shifted in the DATA and ADDRESS registers. If these registers are valid and the transfer direction has been set in the R_W register, the enable signal EN will be made HIGH.


**Figure 17**
CTRL

```
'CTRL' is a state machine controller.

This state machine controller has 12 states.
No stack is available for 'subroutine' calls.
This controller is enabled following system reset.

This state machine controller has no connectors.

Text for state number 1 (reset state) of 'CTRL':
----------------------v----------------------
idle:
  sp_conv idle;
  [ s = 1
  | 1 bit_cnt reset;
      address reset;
      data reset;
      r_w reset;
      -> receive_addr_bit
  | 0 << ]
----------------------^----------------------

Text for state number 2 of 'CTRL':
----------------------v----------------------
receive_addr_bit:
  [ bit_cnt < 7
  | 1 [ up = 1
      | 1 sp_conv addr_sp;
          address load;
          bit_cnt inc;
          <<
      | 0 << ]
  | 0 -> receive_direction ]
----------------------^----------------------

Text for state number 3 of 'CTRL':
----------------------v----------------------
receive_direction:
  [ up = 1
  | 1 sp_conv set_dir;
      r_w load;
      -> prepare_addr_ack
  | 0 << ]
----------------------^----------------------

Text for state number 4 of 'CTRL':
----------------------v----------------------
prepare_addr_ack:
  [ dn = 1
  | 1 sp_conv send_ack;
      sdaor load;
      -> send_addr_ack
  | 0 << ]
----------------------^----------------------

Text for state number 5 of 'CTRL':
----------------------v----------------------
send_addr_ack:
  [ dn = 1
  | 1 bit_cnt reset;
      [r_w = 1
      | 1 sp_conv read_prt_data;
          data load;
          -> get_data_bit
      | 0 -> receive_data_bit ]
  | 0 sdaor enable; "keep sending ack"
      << ]
----------------------^----------------------

Text for state number 6 of 'CTRL':
----------------------v----------------------
get_data_bit:
  [ bit_cnt < 8
  | 1 sp_conv data_ps;
      data load;
      sdaor load;
      sdaor enable;
      bit_cnt inc;
```

```
      -> send_data_bit
  | 0 bit_cnt reset;
      [address < 1Bh
      | 1 address inc ];
      ->receive_data_ack ]
----------------------^----------------------

Text for state number 7 of 'CTRL':
----------------------v----------------------
send_data_bit:
  [ dn = 1
  | 1 sdaor enable;
      -> get_data_bit
  | 0 sdaor enable; "keep sending ack"
      << ]
----------------------^----------------------

Text for state number 8 of 'CTRL':
----------------------v----------------------
receive_data_ack:
  [dn = 1
  | 1 [sda_del = 1 "NACK"
      | 1 -> wait_S_P
      | 0 sp_conv read_prt_data;
          data load;
          -> get_data_bit]
  | 0 << ]
----------------------^----------------------

Text for state number 9 of 'CTRL':
----------------------v----------------------
wait_S_P:
[ (s = 1) \/ (p = 1)
  | 1 [ s = 1
      | 1 -> receive_addr_bit
      | 0 -> idle ]
  | 0 << ]
----------------------^----------------------

Text for state number 10 of 'CTRL':
----------------------v----------------------
receive_data_bit:
  [ (s = 1) \/ (p = 1)
  | 1 [ s = 1
      | 1 -> receive_addr_bit
      | 0 -> idle ]
  | 0 [ bit_cnt < 8
      | 1 [ up = 1
          | 1 sp_conv data_sp;
              data load;
              bit_cnt inc;
              <<
          | 0 << ]
      | 0 sp_conv write_prt_data;
          data_buf enable;
          -> prepare_data_ack ]]
----------------------^----------------------

Text for state number 11 of 'CTRL':
----------------------v----------------------
prepare_data_ack:
  [ dn = 1
  | 1 data_buf enable;
      sp_conv send_ack;
      sdaor load;
      -> send_data_ack
  | 0 data_buf enable;
      << ]
----------------------^----------------------

Text for state number 12 of 'CTRL':
----------------------v----------------------
send_data_ack:
  [ dn = 1
  | 1 bit_cnt reset;
      data reset;
      [ address < 1Bh
      | 1 address inc ];
      -> receive_data_bit
  | 0 sdaor enable;
      << ]
----------------------^----------------------

End of state descriptions.
```

The SP_CONV operator is used to shift the information bits received from the SDA line into the DATA and ADDRESS registers or to shift the information byte in the DATA register out of this register to drive the SDA line.

```
'SP_CONV' is an operator.

This operator has 8 functions.
The default function is 'idle'.

Text for function 'addr_sp' of 'SP_CONV':
----------------------v----------------------
new_adr := (old_adr shl: 1) \/ (sda_in width: 7).
en := 0.
----------------------^----------------------

Text for function 'data_ps' of 'SP_CONV':
----------------------v----------------------
sda_out:=old_data at: 7.
new_data:=(old_data shl: 1).
en:=0.
----------------------^----------------------

Text for function 'data_sp' of 'SP_CONV':
----------------------v----------------------
new_data:=(old_data shl: 1) \/ (sda_in width: 8).
en:=0.
----------------------^----------------------

Text for function 'idle' of 'SP_CONV':
----------------------v----------------------
en := 0.
----------------------^----------------------

Text for function 'read_prt_data' of 'SP_CONV':
----------------------v----------------------
en:=1.
new_data:=prt_data.
----------------------^----------------------

Text for function 'send_ack' of 'SP_CONV':
----------------------v----------------------
sda_out := 0.
en := 0.
----------------------^----------------------

Text for function 'set_dir' of 'SP_CONV':
----------------------v----------------------
r_w := sda_in. "0 indicates a write from the master"
en := 0.
----------------------^----------------------

Text for function 'write_prt_data' of 'SP_CONV':
----------------------v----------------------
en:=1.
----------------------^----------------------

End of function descriptions.
```

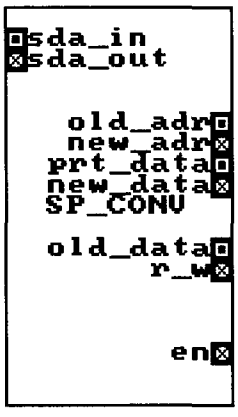


**Figure 18**
SP_CONV

The BIT_CNT register is used to count the number of bits that are shifted in or out the DATA or ADDRESS register. This information is used by the state controller.

```
'BIT_CNT' is a register.

This register is 4 bits wide.
The default function is 'hold'.
This register is loaded with value 0 following system reset.

The value loaded for the 'reset' command is 0.
```

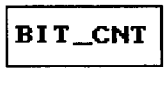


**Figure 19**
BIT_CNT
register

```
'ADDRESS' is a register.

This register is 7 bits wide.
The default function is 'hold'.
This register is loaded with value 0 following system reset.

The value loaded for the 'reset' command is 0.
```

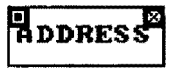


**Figure 20** ADDRESS register

```
'DATA' is a register.

This register is 8 bits wide.
The default function is 'hold'.
This register is loaded with value 0 following system reset.

The value loaded for the 'reset' command is 0.
```


**Figure 21** DATA register

```
'R_W' is a register.

This register is 1 bit wide.
The default function is 'hold'.
This register is loaded with value 0 following system reset.

The value loaded for the 'reset' command is 0.
```

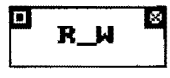


**Figure 22** R_W register

## IIC simulation

When we are going to simulate the IIC unit we need input for the SDA and SCL lines. This has to be a serial input which means that every clock pulse the SDA and SCL lines are sampled. Therefore we connect to ROM units to these pins. The ROM units contain serial bit information that is read by the IIC unit.

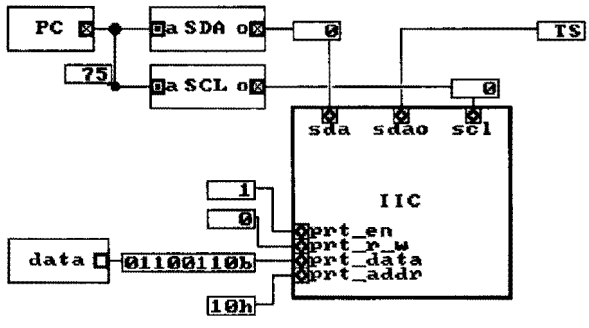


**Figure 23** IIC simulation

Suppose we want to load the FCTL register with value 01100110b. Then we have to send a serial code to the IIC unit that addresses the FCTL register and writes the value into it. This code looks like:

S;FCTL;+W;*(A)*;[data];*(A)*;P

with FCTL = 10h
and [data] = 01100110b

The SCL signal in the ROM is described as 001100110011... . So every $I^2C$ bit transfer takes four IIC clock pulses. In this case it will take 75 clock pulses until the data appears on the data bus and the enable signal becomes valid.

If we want to read the contents of the STAT register we have to send:

S;STAT;+R;*(A);[data]*;(!A);P

with STAT = 1Bh

According to the specification we use !A to indicate that this data byte is the last data byte that has to be transmitted. This also takes 75 clock pulses.

### 3.3.2 Program memory access ports (MADT)

When the MicroDsp is in operation, a program is running on the PCC. This program is normally read from the internal program RAM. At power on, this program RAM is empty and therefore a program must first be loaded into this RAM before the MicroDsp can be running. The way to fill the RAM with a program is to download the program code via the $I^2C$ bus.

The MADT block is the connection between the $I^2C$ unit and the program RAM. This MADT unit can be used to:

- download PCC programs
- download small PCC monitor routines
- verify program memory operation

PCC programs will be stored in the lower part of the program RAM, PCC monitor routines will be stored in the higher part of the same RAM. It is of course also possible to download test vectors into the RAM to verify the program memory operation. Therefore it has to be possible to read back the information that was stored in the RAM.

**MADT specification**

The MADT unit is a combination of four 8-bit register blocks: MADH/MADL, MDTH/MDTL. While the PCC is stopped (STPFLG HIGH), a target program memory address can be loaded into MADH/MADL. The program memory data at the location pointed to by address ports is automatically read immediately after writing MADH and MADL. For program memory data read the $I^2C$ must then receive a stop condition (P) and restart an $I^2C$ read sequence. For writing data into the program memory, the MDTH and MDTL ports are to be filled via the $I^2C$ and after MDTL is received data write follows automatically.

## MADT design

The MADT unit basic elements are the four registers MADH, MADL, MDTH, MDTL. These address registers MADH and MADL can load an address received from the I²C unit. The data registers MDTH and MDTL can load data received from the I²C unit or the program memory. These operations of these four registers are controlled by the ACTION_SEL operator. This operator converts the I²C internal access signals into local register control signals. The specification of these signals can be found in the IDaSS description of the registers below. The operator also controls the function of the MD_SPLIT operator. This operator has three functions that select the data to be stored in the data registers. These will be described in the SPLIT section. First the description of the ACTION_SEL operator follows.

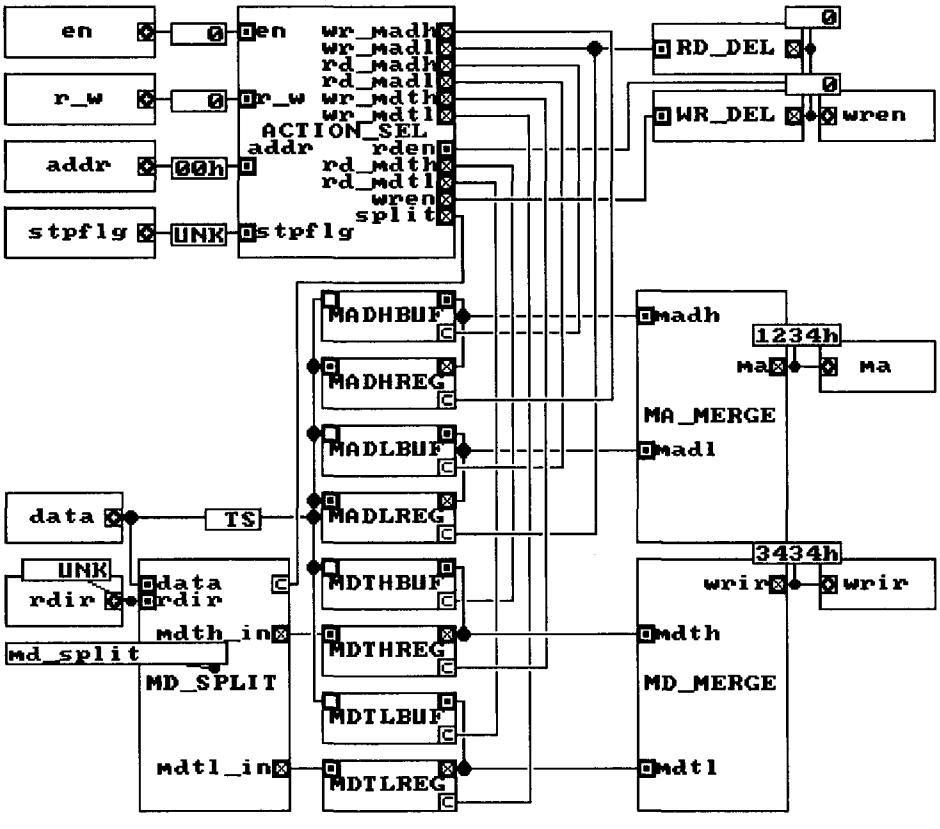


**Figure 24** MADT

```
'ACTION_SEL' is an operator.

This operator has 1 function.
The default function is 'sel'.

Text for function 'sel' of 'ACTION_SEL':
-----------------------v----------------------
_wr:=en /\ (r_w not) /\ stpflg.
_rd:=en /\ r_w /\ stpflg.
_madh:= 16h = addr.
```

```
_madl:= 17h = addr.
_mdth:= 18h = addr.
_mdtl:= 19h = addr.
wr_madh:=_wr /\ _madh.
wr_madl:=_wr /\ _madl.
rd_madh:=_rd /\ _madh.
rd_madl:=_rd /\ _madl.
wr_mdth:=(_wr /\ _mdth) \/ rden.
wr_mdtl:=(_wr /\ _mdtl) \/ rden.
rd_mdth:=_rd /\ _mdth.
rd_mdtl:=_rd /\ _mdtl.
split:=((_wr /\ _mdtl) \/ rden),((_wr /\ _mdth) \/ rden).
wren:=_wr /\ _mdtl.
----------------------^----------------------

End of function descriptions.
```

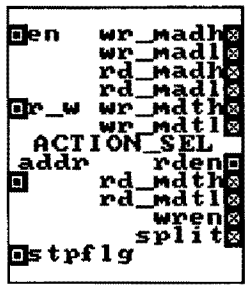


**Figure 25** ACTION_SEL operator

The MD_SPLIT operator actually is a multiplexer that has 3 functions. The first 'md_split' splits a program memory data word into two bytes that are then loaded into the data registers. The second and third function are 'data_to_mdth' and 'data_to_mdtl'. These functions route a data byte received from the $I^2C$ to the corresponding data register.

```
'MD_SPLIT' is an operator.

This operator has 3 functions and is controlled by an unnamed control input.
The default function is 'md_split'.

Control specification:
----------------------v----------------------
%00 md_split.
%01 data_to_mdth.
%10 data_to_mdtl.
----------------------^----------------------

Text for function 'data_to_mdth' of 'MD_SPLIT':
----------------------v----------------------
mdth_in:=data.
----------------------^----------------------

Text for function 'data_to_mdtl' of 'MD_SPLIT':
----------------------v----------------------
mdtl_in:=data
----------------------^----------------------

Text for function 'md_split' of 'MD_SPLIT':
----------------------v----------------------
mdtl_in:=rdir from: 0 to: 7.
mdth_in:=rdir from: 8 to: 15
----------------------^----------------------
```

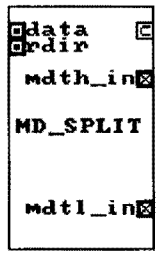


**Figure 26** MD_SPLIT operator

```
Here follows the description of one of the four basic registers. These registers all have the
same description.

'MADHREG' is a register.

This register is 8 bits wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with unknown values after a system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v----------------------
%1 load.
----------------------^----------------------
```


**Figure 27** MADHREG register

Buffers are necessary to transport the data from the registers to the I²C unit. If a buffer is enabled the data can be transferred via the bidirectional DATA bus.

```
'MADHBUF' is a TS buffer.

This TS buffer is 8 bits wide and is controlled by an unnamed control input.

Control specification:
----------------------v----------------------
%1 enable.
----------------------^----------------------
```


**Figure 28** MADHBUF buffer

## MADT simulation

When we simulate the MADT unit we have to check wether we can read several words from the program memory and send these via the I²C bus to the host computer and wether we can receive several words from the host and write these into the program memory.

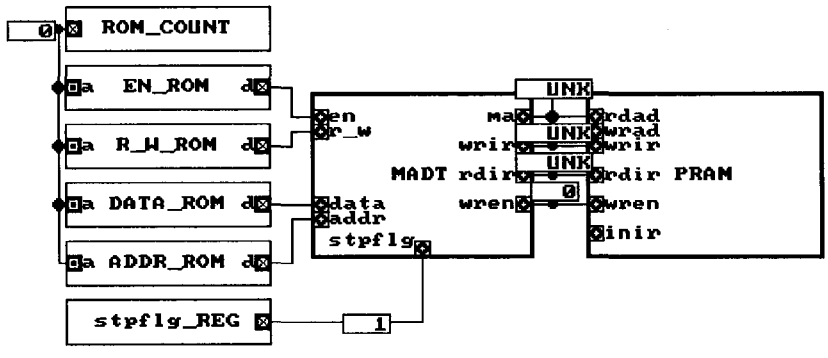


**Figure 29** MADT Simulation

First we will try to read one byte from the program memory. Reading more bytes is just a repetition of the same procedure. Suppose we have just received a request from the I²C unit to read the program memory at address 0123h. Then we have to store the this address into the MADH and MADL register. This can be done by addressing the MADH register at address 16h and writing the high byte of the program memory address (01h) into it. So the R_W signal has to be LOW for writing and the enable signal HIGH. Then we can write the low byte (23h) at address 17h. This next address location will be automatically selected by the I²C unit which has an automatic address increment mode, but for the simulation of this block separately we have to do the incrementing ourselves. After the low byte has been written the program memory will be read and the data will be stored into the MDTH and MDTL registers. We will now read these registers by addressing MDTH at address 18h making the R_W signal HIGH for reading and making the enable signal EN HIGH. Now the high byte of the data word will appear at the DATA bus. This byte can now be transmitted to the host computer via the I²C unit. After this transfer is done the low byte may be read from the MDTL register at address 19h. Then this byte can be transmitted via the I²C unit also.

If we have to write a word into the program memory we have to address the MADH and MADL registers in the same manner as above. After the low byte has been written a program memory read follows automatically and the value will be stored in the MDTH and MDTL register. However, we do not want to use this data, we want to fill the MDTH and MDTL registers with the data that we have just received from the $I^2C$ unit. Therefore we overwrite the data in MDTH and MDTL registers by addressing the MDTH register first, making the R_W signal LOW, putting the high byte of the data word on the DATA bus and making the enable signal HIGH. Then we do the same for the low byte. After this low bytes has been stored in the MDTL register a program memory write follows automatically and the data in MDTH and MDTL is stored in the program memory at the address in MDTH and MDTL.

This completes the simulation of the MADT unit.

Note: For burst transfers the RAM address is transmitted for every word while this address only is incremented for every next data word. In this case an $I^2C$ transfer of one memory word costs 5 $I^2C$ bytes + START and STOP conditions. If an auto-increment mode was provided this could save 2 of the 5 bytes because MADH and MADL do not need to be transmitted for every word but only once at the beginning of the transfer. If MDTH would be selected automatically after MDTL was accessed this would save another byte of the $I^2C$. This improvement has not been implemented because they are not in the specification and thus not required.

### 3.3.3 External signal override controls (FCTL)

The PC monitor program can manipulate PCC system control lines POC, STOP, IRQ and XBUSE by overriding the external control inputs via the $I^2C$ port FCTL.

**FCTL specification**

FCTL is an 8 bit read/write $I^2C$ port.

| bit | name | function |
|---|---|---|
| 0. | POC_F0 | force PCCPOC to '0' |
| 1. | POC_F1 | force PCCPOC to '1' |
| 2. | IRQ_F0 | force PCCIRQ to '0' |
| 3. | IRQ_F1 | force PCCIRQ to '1' |
| 4. | STP_F0 | force PCCSTP to '0' |
| 5. | STP_F1 | force PCCSTP to '1' |
| 6. | XBUSF0 | force IO bus to disable external access |
| 7. | XBUSF1 | force IO bus to enable external access |

At RESET all bits are cleared so no overriding is done.

The PCC "Power On Clear" can be forced to '0' to override user Power On Clear (USRPOC) for emulator monitor actions and forced to '1' to perform emulator monitor controlled PCC system reset. POC requires at least two cycles PCC execution to complete internal clear actions. PCC Stop (PCCSTP) also must be driven LOW during that time.

PCCPOC := USRPOC * !POC_F0 + POC_F1

The PCC Interrupt Request can be forced to '0' or '1' by IRQ_F0 and IRQ_F1.

PCCIRQ := USRIRQ * !IRQ_F0 + IRQ_F1

The PCC Stop signal is the primary control for the monitor for PCC program execution. Beside STP_F0 and STP_F1, PCC program execution control involves another flag: The Stop flag (STPFLG). While executing internal monitor program (MONPRG) the USRSTP and internal stop forcing (STP_F1) are deactivated automatically and PCC execution control is done via the internal STPFLG.

PCCSTP := (STP_F1 + USRSTP) * !(STP_F0 + MONPRG) + STPFLG

**FCTL design**

The FCTL unit exists of the FCTL register REG, a control operator CTRL, a three-state buffer and a FORCE operator. The first three elements will be found in most of the designs of the IICC units. The REG register will always be used to store data received from the $I^2C$ unit or to store MicroDsp internal and external status or data. The buffer BUF gives the

possibility to use the internal bidirectional DATA bus to transport REG data. The control operator is there to control the function of the register and the buffer. It determines when the register has to load or hold and when the three-state buffer has to be enabled.

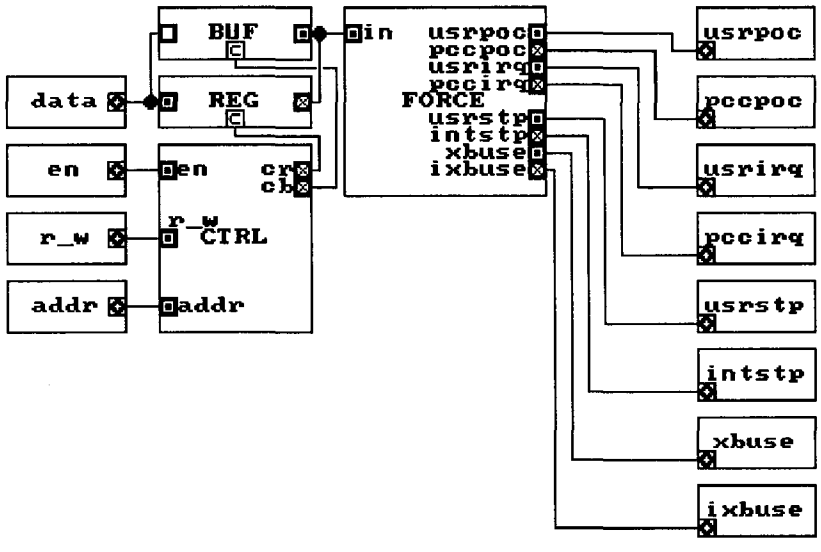


**Figure 30** FCTL

```
'CTRL' is an operator.

This operator has 1 function.
The default function is 'select'.

Text for function 'select' of 'CTRL':
----------------------v-----------------------
cr:=(10h = addr) /\ (r_w not) /\ en.
cb:=(10h = addr) /\ (r_w) /\ en
----------------------^-----------------------
```

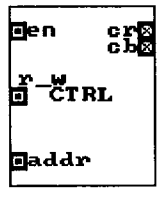


**Figure 31** CTRL operator

In this case the REG register is used to store the force control data. This register has to be zero after RESET because no forcing is allowed at that moment.

```
'REG' is a register.

This register is 8 bits wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with value 0 following system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v-----------------------
%0 hold.
%1 load
----------------------^-----------------------
```

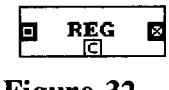


**Figure 32** REG register

```
'BUF' is a TS buffer.

This TS buffer is 8 bits wide and is controlled by an unnamed control input.

Control specification:
----------------------v----------------------
%1 enable
----------------------^----------------------
```


**Figure 33**
BUF
buffer

The FORCE operator is unique for the FCTL unit. It provides the possibility to overrule the POC, IRQ, STP and XBUSE external signals. If the contents of the register REG is zero, no forcing is done.

```
'FORCE' is an operator.

This operator has 1 function.
The default function is 'force'.

Text for function 'force' of 'FORCE':
----------------------v----------------------
pccpoc:=usrpoc /\ ((in at:0) not) \/ (in at:1).
pccirq:=usrirq /\ ((in at:2) not) \/ (in at:3).
intstp:=usrstp /\ ((in at:4) not) \/ (in at:5).
ixbuse:= xbuse /\ ((in at:6) not) \/ (in at:7).
----------------------^----------------------
```


**Figure 34**
FORCE
operator

## FCTL simulation

Simulating the FCTL unit is simple. We address the unit with address 10h and we store a data byte in the register REG. Suppose this data byte is AAh witch is 10101010b, then the signals PCCPOC, PCCIRQ, INTSTP and IXBUSE are all forced to '1'.

## 3.3.4 Program address register (PAD)

The IICC address ports PADH and PADL hold the last PCC program address accessed while executing user program. With this PCC address the monitor can restart the user program at the correct position after a forced PCC stop. The PADH byte also provides information about external program memory and external bus use.

### PAD specification

If a user program is running on the PCC, the program address has to be captured. Therefore the address must be stored on SYSRUN * !STPFLG * USRPRG. The PCC address bus is 13 bits wide. The lower 8 bits of the PCC address are captured in PADL and the upper 5 bits are captured in PADH.
The upper 3 bits of the PADH register which are not used for the address are used to store other information:

| bit | name | function |
|---|---|---|
| 5. | XPRGM | pin level of external program memory select |
| 6. | XBUSE | pin level of external PCC data bus enable |
| 7. | not used | |

### PAD design

The PAD design is globally the same as the FCTL design. The registers PADH and PADL are used to capture the PCC address. The control operator CTRL controls the function of these registers (load, hold or enable).

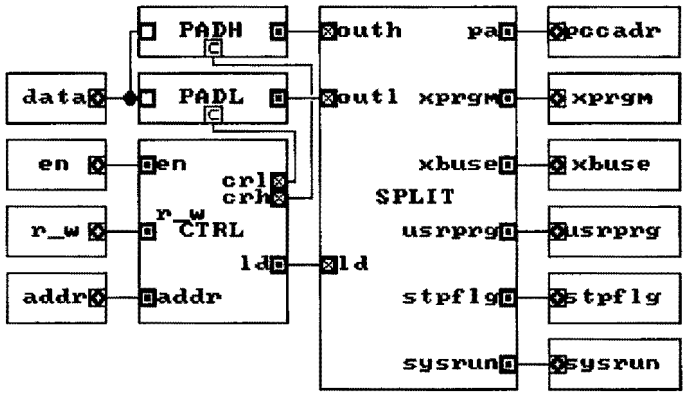


Figure 35 PAD

The SPLIT operator determines when the PCC address must be read and splits this address into two bytes with the addition of the status signals XPRGM and XBUSE.

```
'SPLIT' is an operator.

This operator has 1 function.
The default function is 'split'.

Text for function 'split' of 'SPLIT':
----------------------v----------------------
outl:= pa from: 0 to: 7.
outh:= (pa from: 8 to: 12),
       (xprgm width: 1),
       (xbuse width: 2).
ld:= sysrun /\ (stpflg not) /\ usrprg.
----------------------^----------------------
```

## PAD simulation

To test the function of the PAD unit we first have to be sure that SYSRUN is HIGH, STPFLG is LOW and USRPRG is HIGH. Only in this case the program address is loaded into the PADH and PADL registers. If now the R_W signal and the EN signal are made HIGH and the ADDRESS signal is 11h resp. 12h then the high byte resp. low byte appear on the DATA bus.

### 3.3.5 IO-bus select address and data capture (IOBUS)

The IOBUS block consists of the blocks LBAD, RBAD and IODT. While the PCC is in interrupt mode, the Left and Right Bank select addresses are not copied into the PCC internal registers R7 and R17. As monitor access of IO ports destructs the run-mode IO-select status this status is captured in the LBAD and RBAD ports while executing user program. The IODT read only $I^2C$ port is connected to the PCC IOO bus and captures data from this bus.

#### IOBUS specification

The blocks LBAD and RBAD are used to capture the Left and Right Bank address, if the PCC is about to stop and the capturing is not done by the PCC registers R7 and R17. The condition for this situation is SYSRUN * ! STPFLG * USRPRG together with SLE for a capture in LBAD and RLE for a capture in RBAD. The address kept in LBAD and RBAD are used by the monitor on a restart of normal PCC program execution for reselecting the IO port.
The IODT read-only $I^2C$ port is connected to the PCC IOO bus and captures data if written to the Left Bank IO or if the PCC executes a monitor break PULSE instruction.
To access any data in the PCC system a sequence of PCC instructions is down-loaded into the monitor segment of the program memory. This monitor routine accesses the required data and moves it to the Left Bank IO, so it is captured by IODT. This routine is executed with SYSRUN LOW to enable IODT write and to silence the PCC system WLE.

A 'soft' breakpoint in the PCC program performs a JMP to the monitor program segment. There a PULSE instruction is executed, setting the STPFLG to suspend PCC operation. With SYSRUN HIGH the PULSE bus value [0..255] is also written into IODT to indicate which breakpoint was found. Stopped state can be detected from the $I^2C$ status register STAT. Write IOO bus data into IODT on: (PULSE * MONPROG * SYSRUN) + (WLE * !SYSRUN)

#### IOBUS design

The IOBUS unit contains the LBAD, RBAD and IODT registers. They are controlled by the CTRL operator in the same way as described by the FCTL unit. The LD operator determines when the data from the IO bus has to be captured.

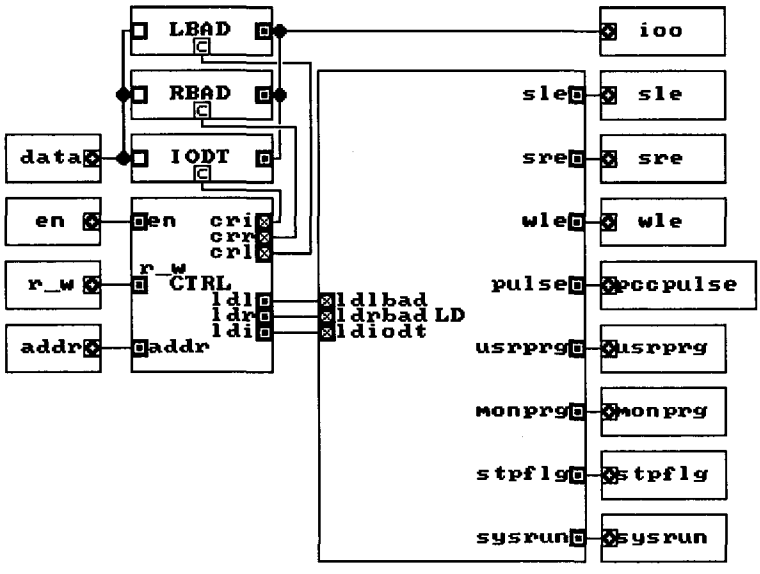


**Figure 36** IOBUS

The LD operator has three outputs. Each output controls the loading of IO bus data for the corresponding register.

```
'LD' is an operator.

This operator has 1 function.
The default function is 'load_ctrl'.

Text for function 'load_ctrl' of 'LD':
----------------------v-----------------------
ldlbad := sle /\ sysrun /\ (stpflg not) /\ usrprg.
ldrbad := sre /\ sysrun /\ (stpflg not) /\ usrprg.
ldiodt := (pulse /\ sysrun /\ monprg) \/ (wle /\ (sysrun not)).
----------------------^-----------------------
```

## IOBUS simulation

To simulate the IOBUS unit we only have to check wether we can read the three registers LBAD, RBAD and IODT and wether these registers capture the data on the IO bus at the right moment.

## 3.3.6 Execution start control (XCTL)

To be able to start and stop the PCC with the PC monitor program, a special register is provided. The bits in this 8-bit register have a start and stop control function.

### XCTL specification

Start of PCC execution is to be done by first loading a small PCC monitor program beginning at address 0x1000, then forcing the program instruction multiplexer to 0xF000 (JMP to address 0x1000) for one cycle and releasing the PCC monitor STOP signal. On writing port XCTL with STARTX HIGH the STPFLG is cleared and with the FJ1000 bit set a forced jump to 0x1000 instruction (code 'F000') is executed. The routine starting at 0x1000 may contain data moves to access or write PCC system data or only a jump instruction to the starting point in the user program.

| bit | name | function |
|---|---|---|
| 0. | SYSRUN | enable PCC system bus writes and PULSE #n into IODT |
| 1. | SSTFLG | execute single step |
| 2. | XPRGF0 | force to internal memory program |
| 3. | XPRGF1 | force to external memory program after JMP execution |
| 4. | STARTX | startup PCC by clearing the STPFLG |
| 5. | FJ1000 | forced JMP 0x1000 at startup |
| 6. | not used | |
| 7. | not used | |

PCC signals from the pins, that have been blocked while in monitor mode, are permitted to the PCC immediately on executing the first JMP instruction JMP=(IR15*IR14*IR13), which must be to user start location. Signals sampled on PH2 of the JMP will affect PCC operation in the first cycle of the user program. Switch to external memory program must be done at the same moment to enable instruction access of the first user instruction.

PCC program execution can be suspended by a MicroDsp RESET, by setting the STPFLG on execution of a PULSE #n instruction while in the monitor program segment, or writing a forced stop (STP_F1) which also sets the STPFLG.

The SYSRUN flag controls PCC signals SRE, SLE, WRE, WLE, PULSE and IAK. If SYSRUN is LOW all signals are inhibited for the PCC user bus system and only the monitor ports are written. With SYSRUN LOW port IODT accepts LIO data, while SYSRUN is HIGH enables writing the data of a PULSE #n instruction into the IODT port. The PCC IBY level is sampled into a flipflop while SYSRUN is HIGH and this sampled level is output to user IBY while in monitor mode ADR12 is HIGH.

The SSTFLG allows the PCC to execute only one instruction when in non-monitor mode. For XEC and NZT instructions this means that the PCC.WAIT signal must be used to allow two-clock execution. On executing the user program instruction the SSTFLG is cleared, the STPFLG and the PCC.STOP are set to stop the PCC. Now the monitor program can take

over. User program is executed if external memory is accessed (XPRGM HIGH) or if lower 4096 internal memory is accessed (ADR12 LOW). The STPFLG is set on USRPRG * SSTFLG * !WAIT.

Program access can be selected to internal 4160 words SRAM or to external memory using the whole of the PCC program address range of 8192 words. If initially internal memory is selected (XPRGM LOW), the monitor program can force to access external memory access by overriding XPRGM with the XCTL startup port XPRGF1 control bit.

RESET initializes the STPFLG=!XPRGM, so a XPRGM ='0' -select internal program- sets the STPFLG to block PCC operations, waiting for monitor program control. Connecting XPRGM to '1' enables external program and the PCC starts instantly without intervention of the monitor. (for proper operation POC timing must be respected).

Switching internal/external is possible with XPRGM='0' by setting the XPRGF1 bit in the XCTL port. The program internal/external multiplexer control IXPRGM (see figure x.) ORs the external memory controls:

IXPRGM := XPRGM * !XPRGF0 + XPRGF1

Memory switching provides the option to partially monitor external memory programs by stopping PCC execution (STP_F1='1'), selecting memory (XPRGF0='1') and executing the internal memory monitor routines.

XBUSE at '1' allows all IO to access an external IO bus. XBUSE can be overridden by the FCTL port signals XBUSF0 and XBUSF1:

IXBUSE := XBUSE * !XBUSF0 + XBUSF1

If the split data bus and its controls are to be routed to external hardware port LIO6 is used for data to PCC (IOI), port LIO7 for data from PCC (IOO) and port LIO8 for bus control lines. In case an external IO is addressed the external selected port can indicate this by setting LBDIS='1' for Left Bank Disable, or RBDIS='1' for Right Bank Disable, to disable internal Left or Right Bank access by disabling propagation of chip internal RLE, WLE, RRE and WRE signals. LBDIS and RBDIS also cause the internal IOI lines to be driven from the port LIO6 pins.

## XCTL design

The XCTL design is globally the same as the FCTL design. The register REG is used to store the XCTL data. The control operator CTRL controls the function of the register (load, hold or enable).

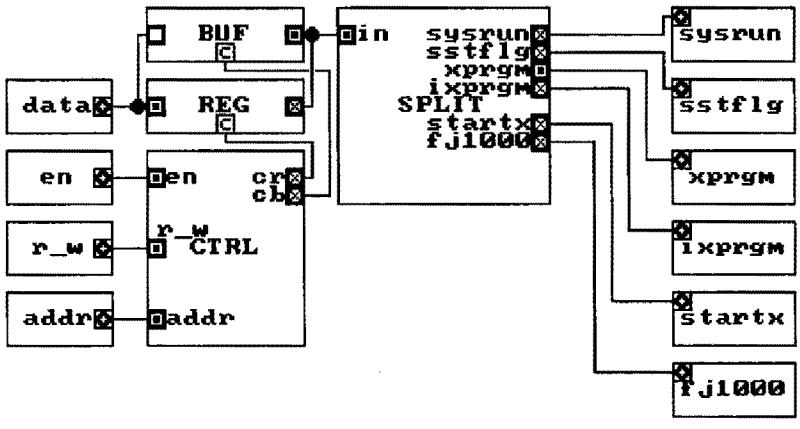


**Figure 37** XCTL

The SPLIT operator splits the data byte of XCTL into single control signals.

```
'SPLIT' is an operator.

This operator has 1 function.
The default function is 'split'.

Text for function 'split' of 'SPLIT':
----------------------v----------------------
sysrun:=in at:0.
sstflg:=in at:1.
ixprgm:=xprgm /\ ((in at:2) not) \/ (in at:3).
startx:=in at:4.
fj1000:=in at:5.
----------------------^----------------------
```

## XCTL simulation

We can simulate the XCTL unit by addressing the unit with address 19h and writing a data byte into it. If for example bit 3 of that byte is HIGH then the XPRGM value will be overruled and IXPRGM will become HIGH.

## 3.3.7 System status register (STAT)

The IICC status register STAT enables access of the state of external pins for the monitor.

### STAT specification

The port STAT provides the following signal status:

| bit | name | function |
|---|---|---|
| 0. | USRPOC | level of user Power On Clear |
| 1. | USRIRQ | level of user Interrupt Request |
| 2. | USRSTP | level of user Stop |
| 3. | PCCIBY | level of PCC Interrupt Busy |
| 4. | PCCSTP | level of PCC Stop |
| 5. | LBDIS | level of internal Left Bank select disable |
| 6. | RBDIS | level of internal Right Bank select disable |
| 7. | STPFLG | level of Stop flag |

### STAT design

The STAT design contains the STAT register REG, a control operator CTRL and a merge operator MERGE. The function of the control unit is the same as described at the FCTL unit.

The MERGE operator merges 8 status bits into one status byte.

```
'MERGE' is an operator.

This operator has 1 function.
The default function is 'merge'.

Text for function 'merge' of 'MERGE':
----------------------v----------------------
out:=stpflg, rbdis, lbdis, pccstp,
     pcciby, usrstp, usrirq, usrpoc.
----------------------^----------------------
```

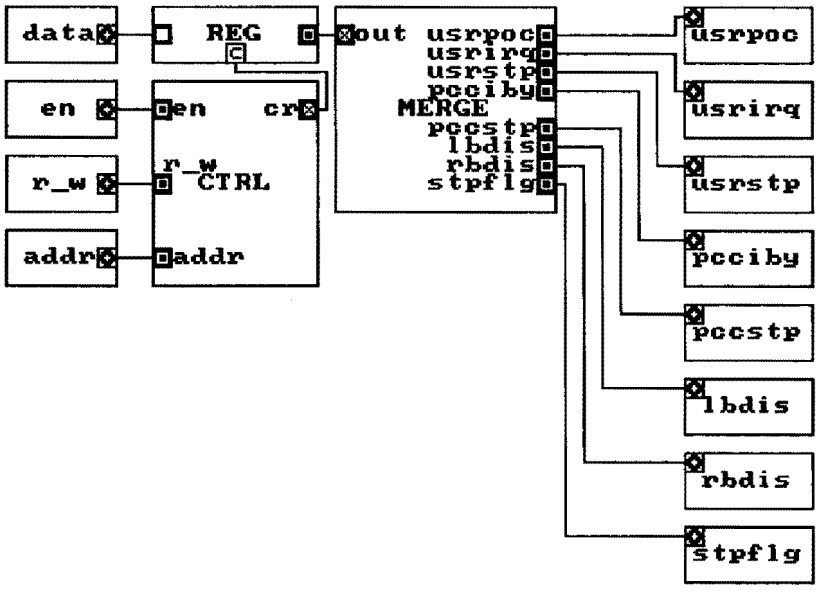


**Figure 38** STAT

### STAT simulation

Simulating the STAT unit is simple. Select the unit at address 1Bh and make R_W and EN HIGH. Then the STAT data should appear on the DATA bus.

## 3.3.8 Mail port (MAIL)

The MAIL port enables signalling of data from the MicroDsp system via LIO9 to the monitor while running a program. The I²C interface may continuously look at this port. The monitor program on the host computer then has the possibility to display debug information, for instance the state of a software state machine.

### MAIL specification

The MAIL port is connected to the IOO and IOI bus. It is possible to receive information from the PCC but the PCC may also use information stored in the MAIL port by the I²C unit.

### MAIL design

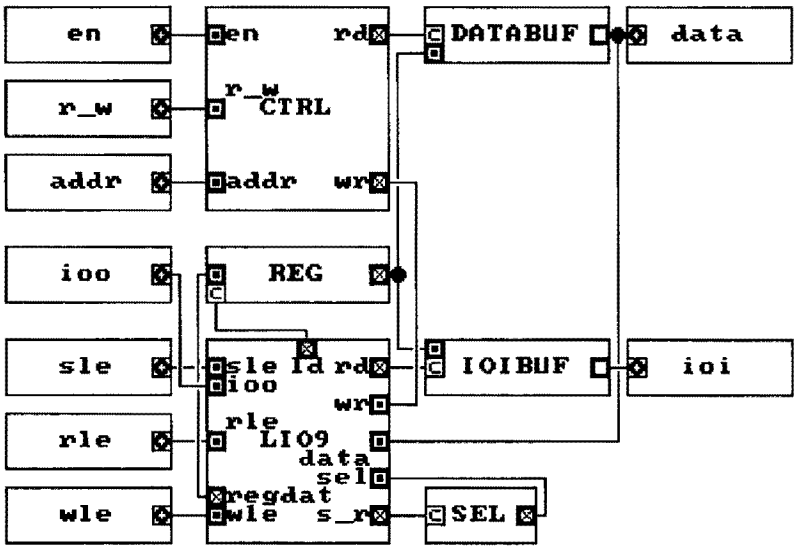


**Figure 39** MAIL

```
'LIO9' is an operator.

This operator has 1 function.
The default function is 'Read_Write'.

Text for function 'Read_Write' of 'LIO9':
----------------------v----------------------
"Select LIO9 or
 Load ioo data into REG or
 Load data data into REG or
 Enable IOIBUF"

_match:=(ioo=9h).

s_r:=(sle /\ (_match not)),(sle /\ _match).
ld:=(wle /\ sel) \/ wr.
rd:=rle /\ sel.
regdat:=(wle /\ sel)
  if0: data
  if1: ioo.
----------------------^----------------------
```

## MAIL simulation

For the MAIL simulation we do two checks. First we try to write the MAIL register with the $I^2C$ unit and read it via the LIO9 port. Then we try to write the MAIL register via the LIO9 IO port and to read the contents via the $I^2C$.

Writing the MAIL register is done by making addressing the IICC MAIL unit at address 1Ch, putting data on the DATA bus, making the R_W signal LOW and the EN signal HIGH. Now the data is stored into the MAIL register. We can read it via LIO9 by selecting LIO9 with the PCC instruction SLE and putting address 09h at the IOO bus. Then we enable reading with RLE and the data appears on the IOI bus.

Now we try to write the MAIL register via the LIO9 port. We select the port at address 09h with SLE. Then we write the value on the IOO bus into the MAIL register with WLE. We can read this register via the $I^2C$ port if we address the MAIL port at address 1Ch, make the R_W signal HIGH and the enable signal LOW. The data should now appear on the DATA bus.

This completes the simulation of the MAIL port.

## 3.3.9 Program type control (PRGTYPE)

The PRGTYPE operator determines the type of program execution. We have three types of program execution:

- user program (USRPRG)
- monitor program (MONPRG)
- external program (PRGEXT)

We have user program execution if the PCC address is in the range of 0000h .. 0FFFh. Monitor program execution is done from 1000h .. 103Fh. PRGEXT is valid if external program memory is enabled.

```
'PRGTYPE' is an operator.

This operator has 1 function.
The default function is 'prgtype'.

Text for function 'prgtype' of 'PRGTYPE':
----------------------v-----------------------
_adr6:=pccadr at:6.
_adr7:=pccadr at:7.
_adr8:=pccadr at:8.
_adr9:=pccadr at:9.
_adr10:=pccadr at:10.
_adr11:=pccadr at:11.
_adr12:=pccadr at:12.

_adr6_11:=(_adr11 \/ _adr10 \/ _adr9 \/ _adr8 \/ _adr7 \/ _adr6).

prgext:=ixprgm \/ (_adr12 /\ (_adr6_11 not)).
usrprg:=(ixprgm not) /\ (_adr12 not).
monprg:=(ixprgm not) /\ _adr12 /\ (_adr6_11 not).
----------------------^-----------------------
```

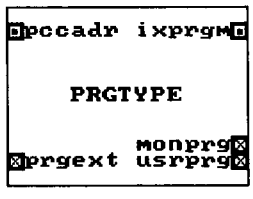


**Figure 40**
PRGTYPE operator

## 3.3.10 Monitor control (MONCTRL)

The PCC IBY level is sampled into a flipflop while SYSRUN is HIGH and this sampled level is output to user IBY while in monitor mode ADR12 is HIGH.

### MONCTRL design

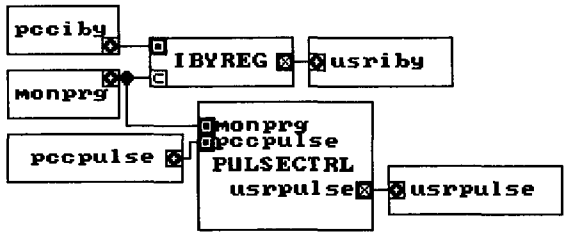


**Figure 41** MONCTRL

The PULSE signal is used for stopping the PCC program execution by setting the STPFLG on execution of a PULSE #n instruction while in the monitor program segment. This PCCPULSE signal may not be passed to the user. Therefore operator PULSE only enables the PCCPULSE signal while not in monitor program mode.

```
This operator has 1 function.
The default function is 'pulsectrl'.

Text for function 'pulsectrl' of 'PULSECTRL':
----------------------v-----------------------
usrpulse:=pccpulse /\ (monprg not).
----------------------^-----------------------
```

## 3.3.11 Stop control (STOPCTRL)

The STOPCTRL unit controls the STPFLG. There are several conditions necessary for the STPFLG to be set.

### STOPCTRL specification

RESET initializes STPFLG = !XPRGM, so a XPRGM = '0' - select internal program - sets the STPFLG to block PCC operations, waiting for monitor program control. In normal operation the stpflag is set on

USRPRG * SSTFLG * !WAIT

and is cleared when STARTX is HIGH.

### STOPCTRL design

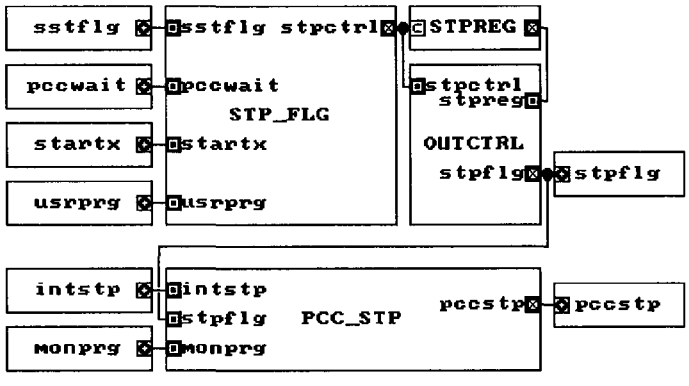


**Figure 42** STOPCTRL

```
'STP_FLG' is an operator.

This operator has 1 function.
The default function is 'stop'.

Text for function 'stop' of 'STP_FLG':
----------------------v----------------------
stpctrl:=startx,(usrprg /\ sstflg /\ (pccwait not)).
----------------------^----------------------
```

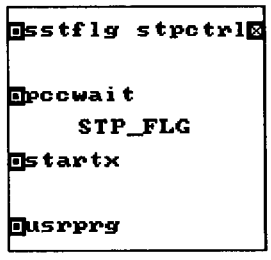


**Figure 43** STP_FLG operator

The OUTCTRL operator is there to make the STPFLG output valid immediately after the conditions change.

```
'OUTCTRL' is an operator.

This operator has 1 function.
The default function is 'outctrl'.

Text for function 'outctrl' of 'OUTCTRL':
----------------------v----------------------
stpflg:=( stpreg        /\ ((stpctrl at:1) not)) \/
        ((stpctrl at:0) /\ ((stpctrl at:1) not)).
----------------------^----------------------
```

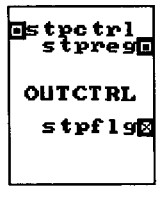


**Figure 44** OUTCTRL operator

```
'STPREG' is a register.

This register is 1 bit wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with value 0 following system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v----------------------
%01 setto:1.
%1x reset.
----------------------^----------------------
```


**Figure 45** STPREG register

The PCCSTP signal is only HIGH if the STPFLG is HIGH or if the internal stop signal INTSTP is HIGH while not in monitor mode.

```
'PCC_STP' is an operator.

This operator has 1 function.
The default function is 'pcc_stp'.

Text for function 'pcc_stp' of 'PCC_STP':
----------------------v----------------------
pccstp:=(intstp /\ (monprg not)) \/ stpflg.
----------------------^----------------------
```

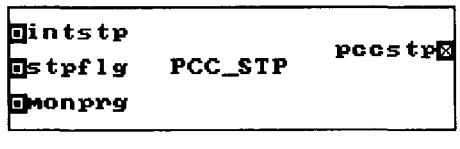


**Figure 46** PCC_STP operator

## IICC simulation

The IICC unit has been simulated by checking the specifications for the several units separately. These simulations are described in the corresponding sections. For this simulation the environment as in Figure 47 has been used.

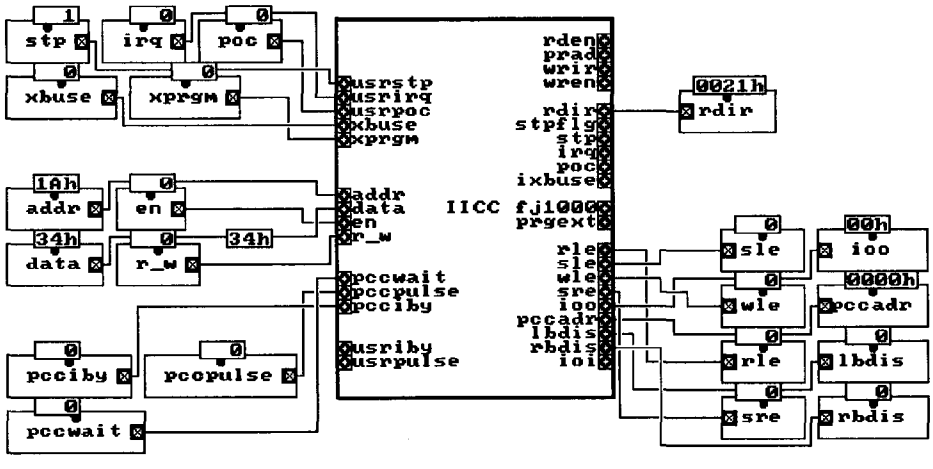


**Figure 47** IICC simulation

## 3.4 Capture Timer Unit (CTU)

A timer unit is a time-base counter that is counting up to a predefined value, and then gives an interrupt. A capture unit is a register that can read the time in the time-base counter on the moment that a capture command is given. A capture-timer unit (CTU) is a combination of a capture unit and a timer unit.

The capture timer unit will be used for several applications. Actions in a "real time system" in general are triggered by events, signals from the controlled system. For instance to control a "brushless motor" these trigger events are the detected zero voltage crossings of the motor coils. From the time difference (=difference in capture time) between zero crossings the motor speed is calculated and from previous speed calculations, the acceleration. Values of wanted speed, actual speed and acceleration are used to determine the required start time delay for powering up the next coil driver.
In the PCC program this means that after a capture interrupt a delay time for the next coil start is to be calculated and this delay value is added, by the PCC, to the captured time value to be re-written into the capture timer. At the timer interrupt the PCC then can write a start command to the coil driver.

### CTU Specification

The timer section runs on the PCC system clock. A prescaler brings the input frequency down to obtain the required time resolution of a time-base counter. Two 16-bit timer registers are used to capture the time in the time-base counter or to program a value into it on which the timer has to react if the timer-base counter equals this value.

The timer registers are located on the Left Bank of the MicroDsp memory environment:

| LIO address | name | description |
|---|---|---|
| 0x0A | CPTM1H | Capture Timer 1 High byte |
| 0x0B | CPTM1L | Capture Timer 1 Low byte |
| 0x0C | CPTM2H | Capture Timer 2 High byte |
| 0x0D | CPTM2L | Capture Timer 2 Low byte |
| 0x0E | PRESCL | Prescaler |

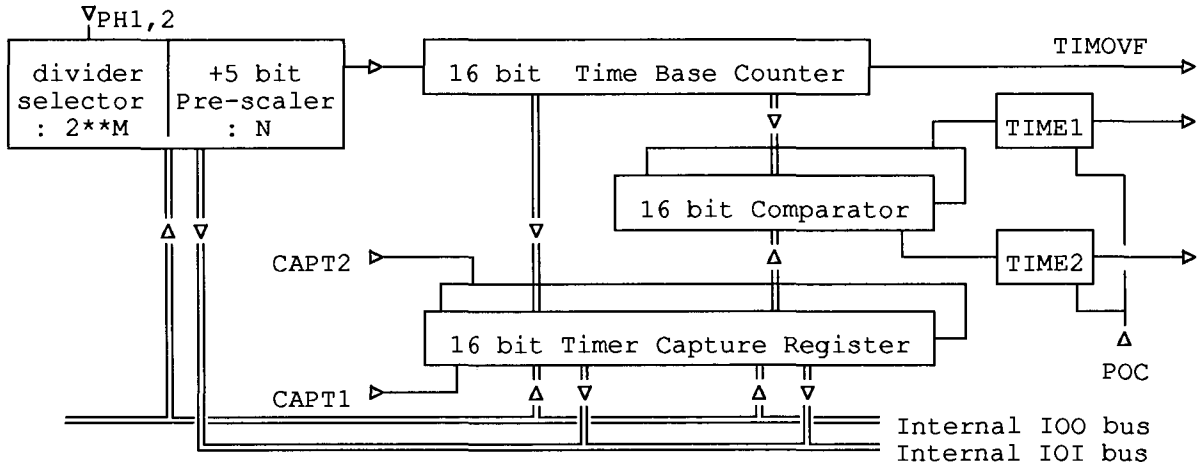


**Figure 48** Functional block diagram of the CTU

## CTU design

As can be seen in the functional block in Figure 48, the CTU contains the following elements:

- Prescaler
- Time-base counter
- 2 Timer-capture units
- 2 Comparators
- 2 TIME registers

These elements can also be found in the IDaSS design of the CTU. Furthermore, the block DIS is added to disable CTU access in case the Left Bank is disabled. In the schematic in Figure 49 the IDaSS design of the CTU is given.

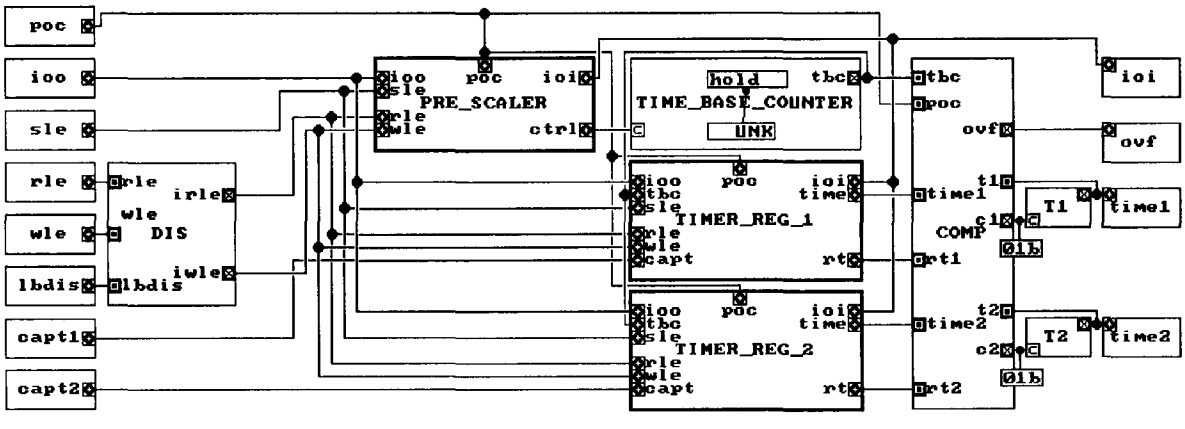


**Figure 49** CTU

The DIS block is an operator with RLE, WLE and LBDIS as input. The outputs IRLE and IWLE are the CTU internal RLE and WLE lines. This means that they are ZERO if LBDIS

is HIGH and that they are RLE and WLE if LBDIS is LOW.

The prescaler has an output CTRL, that controls the time-base counter operation. The time-base counter output TBC is connected to the comparator COMP. This TBC value is continuously compared with the output values TIME of the timer-register blocks. If a match occurs, the corresponding register T will be set and the CTU TIME output will become active. If a time-base counter value has to be captured the time will be read via the TBC input of one of the timer-register blocks and the value can be read via the IOI bus.

## 3.4.1 Time-base counter (TIME_BASE_COUNTER)

The time-base counter is the centre of the CTU design. It is a 16-bit register that normally increments when the prescaler gives a pulse to it. The rest of the time the time-base counter has to hold the register value. The time-base counter must have the possibility to reset in case of special conditions like power on clear (POC) and prescaler programming. So the time-base counter should have 3 modes: hold, inc, reset. The counter is reset on loading the prescaler and provides its inverted MSB as output TIMOVF to enable time-base extension.
Below, the IDaSS description of the time-base counter register is given:

```
'TIME_BASE_COUNTER' is a register.

This register is 16 bits wide and is controlled by an unnamed control
input.
The default function is 'hold'.
This register is loaded with unknown values after a system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v-----------------------
%01 inc.
%1x reset.
----------------------^-----------------------
```

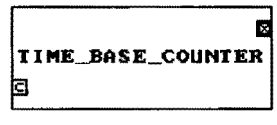


**Figure 50** Time-base counter

## 3.4.2 Prescaler (PRESCALER)

The prescaler will be designed as a subblock that controls the time-base counter. It has to give a pulse to the time-base counter if the prescaler has counted a programmed number of clocks. The prescaler must have the possibility to be programmed via the IOO-bus. The programmed value also has to be read via the IOI-bus. Therefore the control lines SLE, RLE and WLE have to be connected to this subblock. At power on clear (POC) the prescaler is initialized in a hold state. It will start giving pulses to the time-base counter after the prescaler has been programmed with a valid value.

### PRESCALER specification

The 3+5 bit (value=M+N) programmable prescaler can be written from the PCC IOO bus. The first prescaler section includes a 7 bit binary counter out of which a signal is selected under control of the upper 3 bit value M, so that every $2^M$ a clock enable is given to the second prescaler section. This second prescaler section is a programmable 5-bit divide by N counter. The total prescaler division therefore will be : divide by $2^M$ * N. Loading the prescaler with a new value resets the time-base counter.

## PRESCALER design

The pre_scaler is a subblock of the CTU design. It contains the following elements.

- SELECT operator
- PS_SEL register
- IO_CTRL operator
- RUNCTRL operator
- RUN register
- PRESCL register
- PCCTRL operator
- P_COUNT register
- LCCTRL operator
- L_COUNT register
- TBCCTRL operator
- BUFPSC buffer

The schematic of the prescaler design is given in Figure 51.

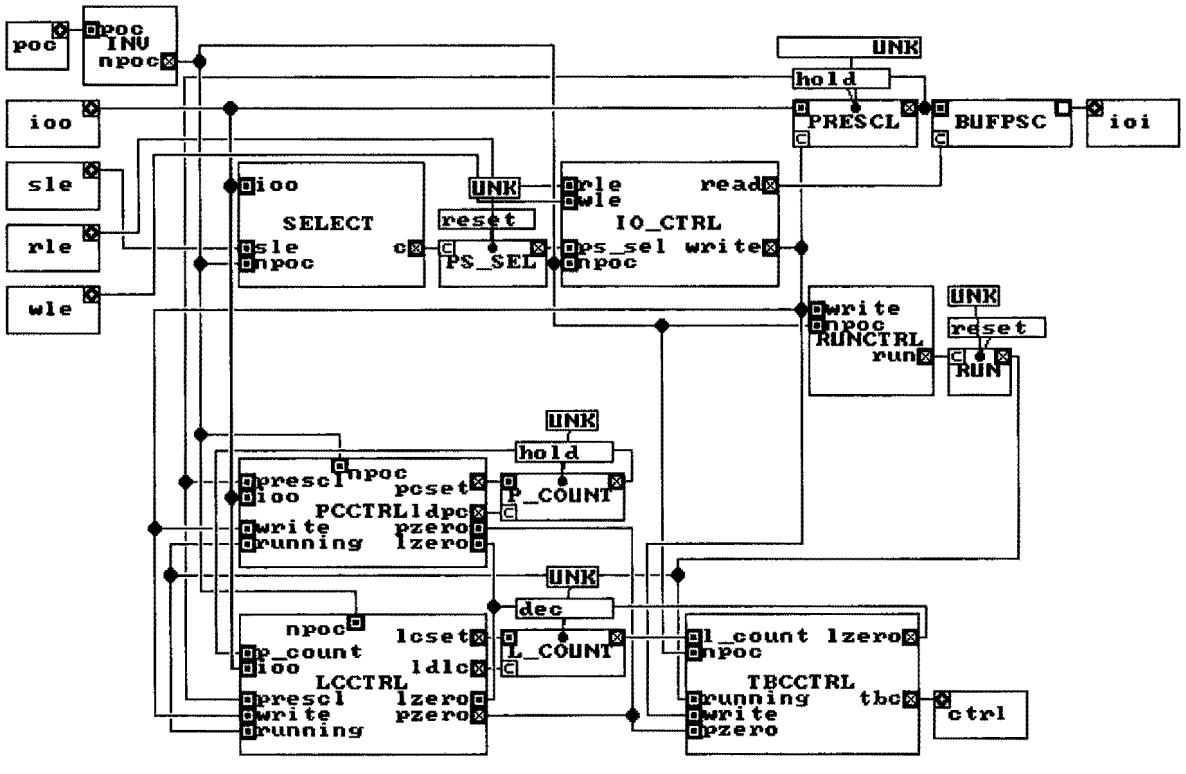


**Figure 51** Pre-scaler

The SELECT operator is used to set the PS_SEL register if the prescaler is selected through a PCC SLE instruction with the address of the prescaler. The IO_CTRL operator determines if a read of the pre-scale value or a write of a new pre-scale value has to be performed. The PRESCL register holds this pre-scale value. A RUNCTRL operator and a RUN register are

used to determine if the prescaler is giving pulses to the time-base counter or not.

As specified, the prescaler is divided in two parts. The first prescaler section includes a 7 bit binary counter P_COUNT out of which a signal is selected under control of the upper 3 bit value M, so that every $2^M$ a clock enable is given to the second prescaler section. This second prescaler section is a programmable 5-bit divide by N counter L_COUNT. The first section is controlled by the PCCTRL operator and the second section is controlled by the LCTRL operator. The total prescaler division therefore will be : divide by $2^M$ * N. In case the value N is set to zero the 5 bit prescaler divides by 32. The output of the prescaler to the time-base counter is controlled by the TBCCTRL operator.

A SELECT operator is used to determine if the prescaler block has been selected for reading or writing. If an SLE action with the right address has been performed, the PS_SEL register will be set. The PS_SEL register indicates the selection of the prescaler. If the PS_SEL register is HIGH, then all RLE and WLE actions will influence the running of the prescaler.

```
'PRE_SCALER\SELECT' is an operator.

This operator has 1 function.
The default function is 'select'.

Text for function 'select' of 'PRE_SCALER\SELECT':
----------------------v----------------------
_prescladr:=0Eh. "Pre-scaler address"
c:=((npoc not) \/ (npoc /\ sle /\ (ioo ~= _prescladr))), (npoc /\ sle /\
(ioo=_prescladr)).
----------------------^----------------------
```

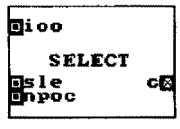


**Figure 52** SELECT operator

```
'PRE_SCALER\PS_SEL' is a register.

This register is 1 bit wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with unknown values after a system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v----------------------
%0l setto: 1.
%lx reset.
----------------------^----------------------
```

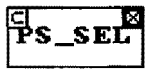


**Figure 53** PS_SEL register

The IO_CTRL operator is used to provide internal READ and WRITE signals. These signals are active if the prescaler is in the selected state. In the PRESCL register, the value of the prescaler is kept. This value can be read via the IOI bus via a BUFPSC buffer.

```
'PRE_SCALER\IO_CTRL' is an operator.

This operator has 1 function.
The default function is 'reg_ctrl'.

Text for function 'reg_ctrl' of 'PRE_SCALER\IO_CTRL':
----------------------v----------------------
write:=(npoc /\ wle /\ ps_sel).
read :=(npoc /\ rle /\ ps_sel).
----------------------^----------------------
```

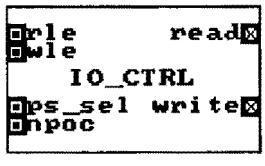


**Figure 54** IO_CTRL operator

```
'PRE_SCALER\PRESCL' is a register.

This register is 8 bits wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with unknown values after a system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v----------------------
%1 load.
----------------------^----------------------
```


**Figure 55** PRESCL register

```
'PRE_SCALER\BUFPSC' is a TS buffer.

This TS buffer is 8 bits wide and is controlled by an unnamed control input.

Control specification:
----------------------v----------------------
%1 enable.
----------------------^----------------------
The default state is disabled.
```


**Figure 56** BUFPSC buffer

The CTU can be in two states: running or stopped. In the case that the timer is running, the prescaler gives pulses to the time-base counter. This is only done after the prescaler has been programmed with a certain value. Programming always requires a write to the PRESCL register. Then the timer is in the running state and this is indicated with a HIGH value in the RUN register. If the prescaler receives a POC, then the timer will be stopped. In this case the RUN register will contain a LOW value. The RUNCTRL operator is there to control the RUN register. In IDaSS this looks like:

```
'PRE_SCALER\RUNCTRL' is an operator.

This operator has 1 function.
The default function is 'runctrl'.

Text for function 'runctrl' of 'PRE_SCALER\RUNCTRL':
----------------------v----------------------
run:=(npoc not),write.
----------------------^----------------------
```


**Figure 57** RUNCTRL operator

```
'PRE_SCALER\RUN' is a register.

This register is 1 bit wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with unknown values after a system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v----------------------
%01 setto: 1.
%1x reset.
----------------------^----------------------
```


**Figure 58** RUN register

The prescaler has been divided into two sections: a power counting part and a linear counting part. The power counter is the P_COUNT register and the linear counter is the L_COUNT register. The L_COUNT register decrements by default. If this register reaches the zero value, the P_COUNT register is decremented by one. Next, the L_COUNT register is reloaded, and starts decrementing again. If both registers become zero, a pulse is given to the time-base counter. The linear counter will be described later, the power counter next.

The PCCTRL operator is used to set the P_COUNT register to a predefined value. This value is $2^M$, where M are the upper three bits of the pre-scale value in the PRESCL register. As soon as the power value has been set into the P_COUNT register this register will decrement once and then the PCCTRL operator will wait on the linear counter to expire. If this has happened the P_COUNT register will decrement once again. This continues until the P_COUNT register becomes zero. Then, the P_COUNT register will be loaded again and a pulse for incrementing is send to the time-base counter.

```
'PRE_SCALER\PCCTRL' is an operator.

This operator has 1 function.
The default function is 'pcctrl'.

Text for function 'pcctrl' of 'PRE_SCALER\PCCTRL':
----------------------v----------------------
pcset:=write
        if0: ((1 width: 7) shl:(prescl from:5 to:7))
        if1: ((1 width: 7) shl:(ioo    from:5 to:7)).
ldpc:=npoc
 if0: npoc,npoc
 if1: (running
  if0: write, write
  if1: (lzero \/ write),((pzero /\ lzero) \/ write)).
----------------------^----------------------
```


**Figure 59**
PCCTRL
operator

```
'PRE_SCALER\P_COUNT' is a register.

This register is 7 bits wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with unknown values after a system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v----------------------
%01 loaddec.
%10 dec.
%11 loaddec.
----------------------^----------------------
```

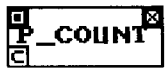


**Figure 60**
P_COUNT
register

The L_COUNT register is the linear counting section of the prescaler. This register continuously counts down from N to zero. N is a value that is formed of the lower 5 bits of the pre-scale value in the PRESCL register. If zero is reached, N is reprogrammed into the counting register by the LCCTRL operator.

```
'PRE_SCALER\LCCTRL' is an operator.

This operator has 1 function.
The default function is 'lcctrl'.

Text for function 'lcctrl' of 'PRE_SCALER\LCCTRL':
----------------------v----------------------
lcset:=write
        if0: (prescl from:0 to:4)
        if1: (ioo    from:0 to:4).
ldlc:=npoc
 if0: npoc
 if1: (running
  if0: write
  if1: (lzero \/ write)).
pzero:=p_count=0.
----------------------^----------------------
```

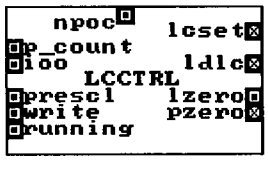


**Figure 61**
LCCTRL
operator

```
'PRE_SCALER\L_COUNT' is a register.

This register is 5 bits wide and is controlled by an unnamed control input.
The default function is 'decrement'.
The value loaded for the 'reset' command is 0.
Control specification:
----------------------v----------------------
%1 loaddec.
----------------------^----------------------
```


**Figure 62** L_COUNT register

The TBCCTRL operator controls the output pulses of the prescaler. Only in case that both counters P_COUNT and L_COUNT have reached zero and the prescaler is in running state, a pulse for incrementing is given to the time-base counter.

```
'PRE_SCALER\TBCCTRL' is an operator.

The default function is 'tbcctrl'.
Text for function 'tbcctrl' of 'PRE_SCALER\TBCCTRL':
----------------------v----------------------
_match:=(l_count=0).
lzero:=_match.
tbc:=npoc
 if0: npoc, npoc
 if1: (running
  if0: write,running
  if1: write,(_match /\ pzero)).
----------------------^----------------------
```

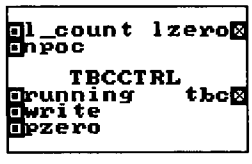


**Figure 63** TBCCTRL operator

## PRESCALER simulation

Simulating the prescaler requires the use of five registers for input: POC, IOO, SLE, RLE and WLE. The outputs are checked by looking at the viewers on the IOI bus and the CTRL output. The CTRL output MSB stands for resetting the time-base counter, the LSB for incrementing. Power On Clear operation is checked by setting the POC register to 1 and setting the SLE, RLE and WLE registers to 0 while the IOO bus carries an unknown value (UNK). This results in the situation that the IOI bus is Three-State (TS) and the CTRL output is zero. Giving clock ticks does not change the output because POC stays active. Now we deactivate POC by setting the POC register to 0. Next we select the prescaler by setting IOO to 0Eh and SLE to 1. After a clock tick the prescaler is selected. We now write a value 1 into the prescaler by setting SLE to 0, setting WLE to 1 and setting IOO to 1. This results in a CTRL output 11b, which means that the time-base counter is reset after a clock tick and that the time-base counter has to increment. We make the WLE register 0 again. We see 01b at the CTRL output which means that the time-base counter still has to increment. After a clock tick the CTRL output again gives 01b. This is correct because a value of 1 in the prescaler means that the PCC clock is divided by 1, which is the same as giving a increment pulse every clock. We now program a value 010 00001b into the prescaler. This means that every $2^2*1=4$ clock pulses an increment pulse has to be given. This simulation is correct. We can now try to read the contents of the prescaler by making RLE 1. This gives 010 00001b at the IOI bus. The last check is giving a POC. Now the CTRL output stays 00b (also when POC is 0 again) which means that the time-base counter will not be reset and will not be incremented until a new value has been programmed into the prescaler.

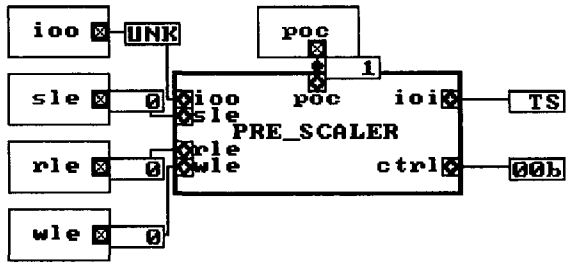


**Figure 64** Pre-scaler in POC state

### 3.4.3 Timer-register unit (TIMER_REG)

In the CTU design, two timer-register units are necessary to capture the time of the time-base counter if the CAPT input rises or to give a time output if the time-base value equals the value of the timer-capture unit.

#### TIMER_REG specification

The timer registers to PCC IO bus interface is cyclic: after access of the HIGH byte the LOW byte is addressed; after access of the LOW byte the HIGH byte is addressed again, etc. While the LOW byte is selected the TIME pulses are frozen until LOW byte write is done to avoid unwanted TIME (interrupt) pulses. Timer interrupt logic has to be build with external hardware.

#### TIMER_REG design

The timer-register units TIMER_REG_1 and TIMER_REG_2 are subblocks of the CTU schematic. They are connected to the IOO and IOI bus, to provide the possibility to program and read the timer_register unit. Therefore also the lines SLE, RLE and WLE are connected to this block. At power on clear (POC) the timer-register unit has to be initialized, therefore a POC input is provided. The timer-register unit must also indicate when the TIME register has to be reset. In the figure below the timer register unit is shown.

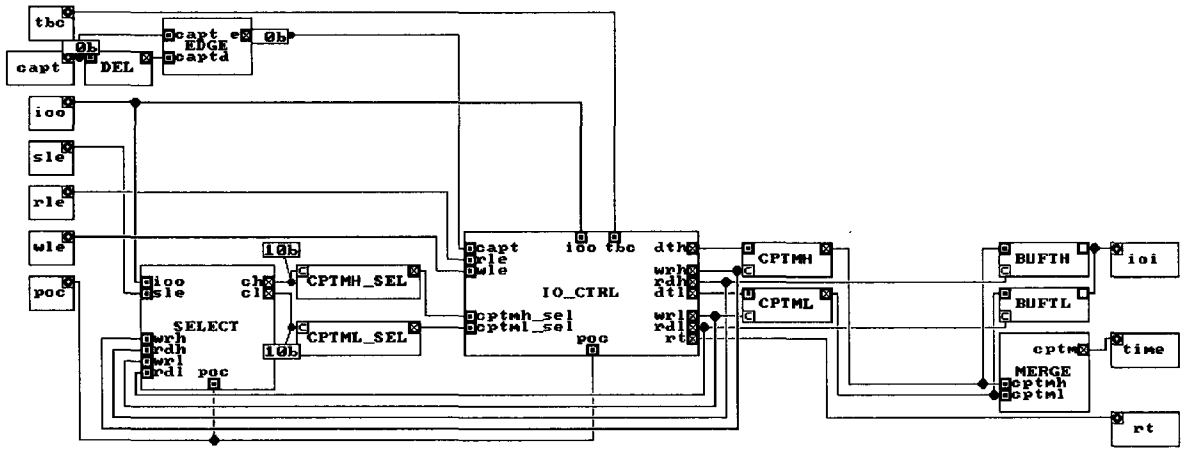


**Figure 65** Timer-register unit

In the figure can be seen that the timer-register unit contains several elements. These are:

- SELECT operator
- 2 CPTM_SEL registers
- IO_CTRL operator
- 2 CPTM registers
- 2 BUFT buffers
- MERGE operator
- EDGE operator

The select operator is used to determine if a timer-register unit is addressed. If one of the timer-register units is selected through a SLE with the address of the timer-register unit, then the CPTMH_SEL or CPTML_SEL register will be set. The CPTMH_SEL is set if the HIGH byte of the capture-timer register is accessed, CPTML_SEL is set if the corresponding LOW byte is accessed.

```
'TIMER_REG_1\SELECT' is an operator.

This operator has 1 function.
The default function is 'select'.

Text for function 'select' of 'TIMER_REG_1\SELECT':
----------------------v----------------------
_cptmhadr:=0Ah. "Timer_register 1 High byte"
_cptmladr:=0Bh. "Timer_register 1 Low  byte"
ch:=(poc \/ (sle /\ (ioo ~= _cptmhadr)) \/ wrh \/ rdh), (((poc not) /\ sle /\
(ioo=_cptmhadr)) \/ wrl \/ rdl).
cl:=(poc \/ (sle /\ (ioo ~= _cptmladr)) \/ wrl \/ rdl), (((poc not) /\ sle /\
(ioo=_cptmladr)) \/ wrh \/ rdh).
----------------------^----------------------
```

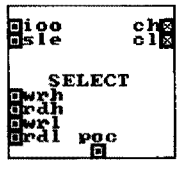


**Figure 66** Select operator

```
'TIMER_REG_1\CPTMH_SEL' is a register.

This register is 1 bit wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with unknown values after a system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v----------------------
%01 setto: 1.
%1x reset.
----------------------^----------------------
```

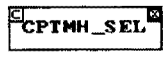


**Figure 67** Select register

The centre of the timer-register unit is a 16 bit capture-timer register CPTM. Reading and writing is done byte-wise so the 16-bit capture-timer register is partitioned into two 8-bit registers CPTMH and CPTML. These registers are controlled by the IO_CTRL operator. If the capture-timer registers are read, the IOI-bus buffers BUFTH and BUFTL are enabled. If the capture-timer registers are written, we have two possibilities. Either the registers are filled with new data from the IOO bus (a timer action) or the registers are filled with the actual time of the time-base counter (a capture action). In the first case writing is done byte after byte, in the second case the two bytes are written simultaneously. The IO_CTRL operator also has an RT output which indicates that the time TIME output registers of the CTU should be reset if the LOW byte of the capture-timer register has been written.

```
'TIMER_REG_1\IO_CTRL' is an operator.

This operator has 1 function.
The default function is 'io_ctrl'.

Text for function 'io_ctrl' of
 'TIMER_REG_1\IO_CTRL':
----------------------v----------------------
_npoc:=poc not.
wrh:=_npoc /\ ((wle /\ cptmh_sel) \/ capt).
wrl:=_npoc /\ ((wle /\ cptml_sel) \/ capt).
rdh:=_npoc /\ rle /\ cptmh_sel.
rdl:=_npoc /\ rle /\ cptml_sel.
dth:=capt
 if0: ioo
 if1: (tbc from: 8 to: 15).
dtl:=capt
 if0: ioo
 if1: (tbc from: 0 to: 7).
rt:=(cptml_sel /\ wle),(cptml_sel /\ (wle not)).
----------------------^----------------------
```

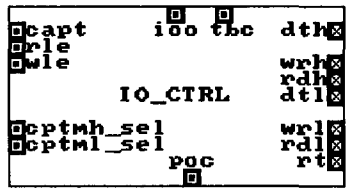


**Figure 68** IO_CTRL operator

```
'TIMER_REG_1\CPTMH' is a register.

This register is 8 bits wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with unknown values after a system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v----------------------
%1 load.
----------------------^----------------------
```


**Figure 69** Capture-timer register

```
'TIMER_REG_1\BUFTH' is a TS buffer.

This TS buffer is 8 bits wide and is controlled by an unnamed control input.

Control specification:
----------------------v----------------------
%1 enable.
----------------------^----------------------

The default state is disabled.
```


**Figure 70** IOI Buffer

The MERGE operator is a simple block that concatenates the two 8-bit inputs so that a 16-bit output is formed.

```
'TIMER_REG_1\MERGE' is an operator.

This operator has 1 function.
The default function is 'merge'.

Text for function 'merge' of 'TIMER_REG_1\MERGE':
----------------------v----------------------
cptm:=cptmh,cptml.
----------------------^----------------------
```

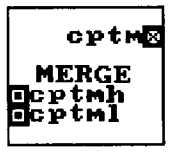


**Figure 71** M e r g e operator

The EDGE operator is a block that detects the up-going edge of the CAPT input. This is done by comparing the actual CAPT value with the previous CAPT value.

```
'TIMER_REG_1\EDGE' is an operator.
This operator has 1 function.
The default function is 'edge'.

Text for function 'edge' of 'TIMER_REG_1\EDGE':
----------------------v----------------------
e:=capt /\ (captd not).
----------------------^----------------------
```


**Figure 72** EDGE operator

## TIMER_REG simulation

When we simulate the timer-register unit we use the environment given in the figure. The timer-register unit has 7 inputs. The POC, IOO, SLE, RLE and WLE input do not need any explanation, the TBC input carries the time-base counter time and the CAPT input is used by the user to give a capture pulse.

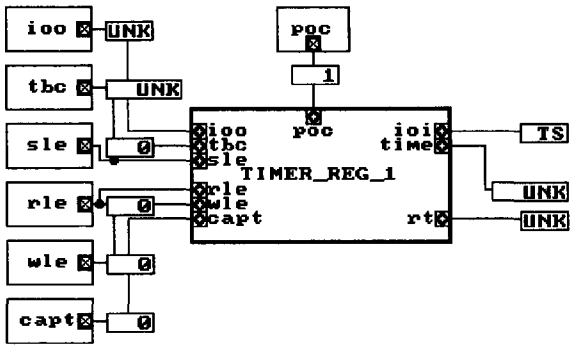


**Figure 73** Timer-register unit in POC state

We start with the POC state, that is POC is HIGH and SLE, RLE and WLE are LOW. Now the IOI bus driver of the timer register unit is in tree-state, and the TIME output and RT output are unknown. After one clock tick the RT output becomes 00b. This is part of initializing the timer-register unit. Now we can make POC LOW.

Suppose we want to capture the time of the time-base counter. For simulating this we put a random value in the TBC register for example 1234h and we make the CAPT register HIGH. After a clock tick the timer-register unit has copied the value 1234h into it's register. The value occurs at the TIME output of the timer-register unit. We now make the CAPT register LOW again. If we want to read this value with the PCC we have to select the HIGH byte of the timer register unit, in this case at address 0x0A. We do this by filling the IOO bus register with 0x0A and making SLE HIGH. After one clock tick this register is selected. Now we make SLE LOW again and RLE HIGH. After a clock tick a value 12h appears at the IOI bus. This is the HIGH byte of the captured value in the timer-register unit. After another clock tick the LOW byte will appear on the IOI bus. We now make RLE LOW again and the IOI bus becomes tree-state. The RT output has become 01b, but this result is not used.

If we want to program a value into the timer-register unit, we first have to select the unit if this was not done before. Therefore we make SLE HIGH and we put 0Ah on the IOO bus. After a clock tick, the unit is selected. We make SLE LOW again. Now we can write a value via the IOO bus into the unit. We load a value in the IOO register, for example 43h and we make WLE HIGH. After a clock tick, this value is programmed into the HIGH byte of the timer-register unit. The RT output becomes 10b, which means that the TIME registers of the CTU have to be reset after writing of the LOW byte. We now load the IOO register with 21h which means that after a clock tick (with WLE still HIGH) the value 21h is programmed into the LOW byte of the timer-register unit. The value 4321h appears on the TIME output of the timer-register unit. The RT output becomes 00h.

This completes the simulation of the timer-register unit. A simulation of the complete CTU can be found in the paragraph CTU simulation.

### 3.4.4 Dual comparator (COMP)

A dual comparator COMP is necessary to compare the time-base-counter value TBC with the TIME values of the timer-register units. If they match, the control output C sets the time register T. The reset-timer inputs RT are used to reset the time registers. The input T is used to look at the T registers output. The comparator provides also an OVF output which indicates the overflow of the time-base counter. OVF is the inverted 15th output-bit of the time-base counter. This block also has a POC input to initialize the T registers with a HIGH value at power up.

```
'COMP' is an operator.

This operator has 1 function.
The default function is 'compare'.

Text for function 'compare' of 'COMP':
----------------------v----------------------
c1:=poc
 if0: (t1
  if0: t1,((tbc=time1) /\ ((rt1 at:0) not))
  if1: (rt1 at:1),(t1 not))
 if1: (poc not),poc.
c2:=poc
 if0: (t2
  if0: t2,((tbc=time2) /\ ((rt2 at:0) not))
  if1: (rt2 at:1),(t2 not))
 if1: (poc not),poc.
ovf:=(tbc at:15) not.
----------------------^----------------------
```

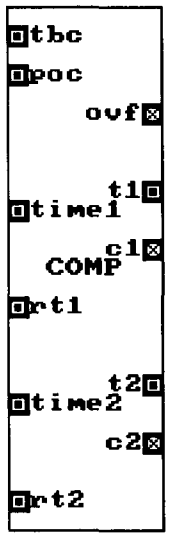


**Figure 74** Dual comparator

## 3.4.5 Time output registers (T)

The TIME output of the CTU is made HIGH if the comparator finds a match of the time-base counter value and the timer-capture register. The TIME output has to stay HIGH until a timer-capture LOW-byte write is done. So the TIME output has to be held HIGH for a period. This can be done with a register; the TIME register T. TIME1,2 signals are set high on POC and on TIME1,2==TimeBaseCounter. TIME outputs go low on a write into the lower timer byte starting the time delay. Thus to schedule a rising edge on TIME1,2 a certain time interval dT after a capture this time interval dT is to be added to the CAPT1,2 value and restored in the Capture/Timer register.

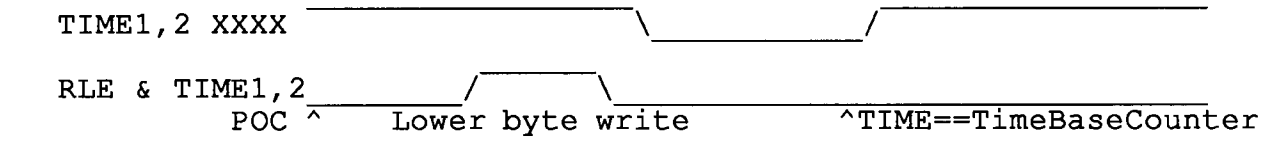


**Figure 75** CTU output timing

```
'T1' is a register.

This register is 1 bit wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with unknown values after a system reset.

The value loaded for the 'reset' command is 0.

This register has the following connectors:

Control connector without a name:
Has a width of 2 bits and is connected to bus 'c1'.

Control specification:
----------------------v----------------------
%01 setto: 1.
%1x reset.
----------------------^----------------------

Continuous output connector without a name:
Has a width of 1 bit and is connected to bus 'time1'.
```


**Figure 76**
Time register

## CTU simulation

To simulate the Capture Timer Unit we have to verify the functions, the CTU should have according to the specification.

These are:

- programming internal registers
- reading internal registers
- waiting for TIME output
- capturing the time

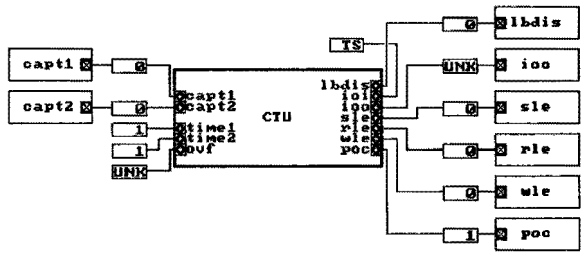


**Figure 77** CTU in POC state

We start in the POC state. After two clock ticks the CTU is initialized. Now POC can be deactivated. We start programming the prescaler with 001 00001b which is a value of 2*1=2. Then we program timer-register 1 with 10h and timer-register 2 with 20h. The TIME outputs are now LOW. After a total of 10h*2=20h clock ticks the TIME1 output will rise and after 40h clock ticks the TIME2 output will rise. We now program the prescaler with value 010 00001b which is $2^2$*1=4*1=4. It will now take 4 PCC clock ticks to make the time-base counter increment. We also program timer-register 2 with value 4 which means that the TIME2 output will go LOW and after a total of 4*4=16 ticks will rise again. Now we make the CAPT1 input HIGH and the time-base counter time is captured in timer register 1. A final check is reading the contents of all the registers of the CTU. The prescaler should still be 04h, and both timer-registers should be 0004h. All tests have been done this way, and all results are satisfying.

## 3.5 I/O ports (LIO)

The MicroDsp has ten IO ports that are located on the Left Bank of the MicroDsp memory map. All the IO ports are quasi bi-directional. Used as output this type of port has an active low drive and a resistive (passive) pull up at high state, which is assisted with an active pull up on transition to high only during a short period (time t.b.s.). When used as an input pin a resistive high level output drive is to be programmed, that can be pulled low by external hardware. At power up all ports are initialised to '1' to allow input mode in order to avoid possible signal direction conflicts. Also if ports programmed to be PCC input using XBUSE and/or XPRGM these flags are forced to select input mode.

### LIO specification

If the internal PCC split data bus and its controls are to be routed to external hardware the internal IO bus control IXBUSE must be made '1'. The FCTL monitor force controls can override the external signal XBUSE. If the internal IO bus is routed to the pins port LIO6 is used for data to PCC (IOI), port LIO7 for data from PCC (IOO) and port LIO8 for control lines. If external hardware selected an existing Left Bank address access will be disabled. This is done by disabling propagation of chip internal RLE and WLE signals.

**LIO0** (address = 0x00) is a Left Bank input/output port. On POC and on IAK this port is automatically selected, therefore in general this port should be used to read the, optionally coded, interrupt lines.

**LIO1, LIO2** and **LIO3** (addresses = 0x01, 0x02 & 0x03) are general purpose Left Bank input/output ports.

**LIO4** and **LIO5** (addresses = 0x04 & 0x05) are general purpose Left Bank input/output ports, which are changed to program instruction inputs if XPRGM is at HIGH state.

**LIO6** (address = 0x06) is a general purpose Left Bank input/output port or if XBUSE is at HIGH it is PCC IO bus input (IOI).

**LIO7** (address = 0x07) is a general purpose Left Bank input/output port or if XBUSE is at HIGH it is PCC IO bus output (IOO).

**LIO8** (address = 0x08) is a general purpose Left Bank input/output port or if IXBUSE is at HIGH it carries PCC IO bus controls:
MSB=7=SRE, 6=SLE, 5=RRE, 4=RLE, 3=WRE, 2=WLE, 1=LBDIS, LSB=0=RBDIS

**LIO9** (address = 0x09) MAIL is a IICC debug mailbox port to the controlling monitor. This port allows monitoring data via the PC simulator/emulator "mailed" by the user program without interrupting program execution.

## LIO design

In IDaSS it was not possible to design quasi-bidirectional ports because a resistive pull up can not be implemented. Therefore the inputs and outputs are separate in the IDaSS design. LIO has been divided into three parts.

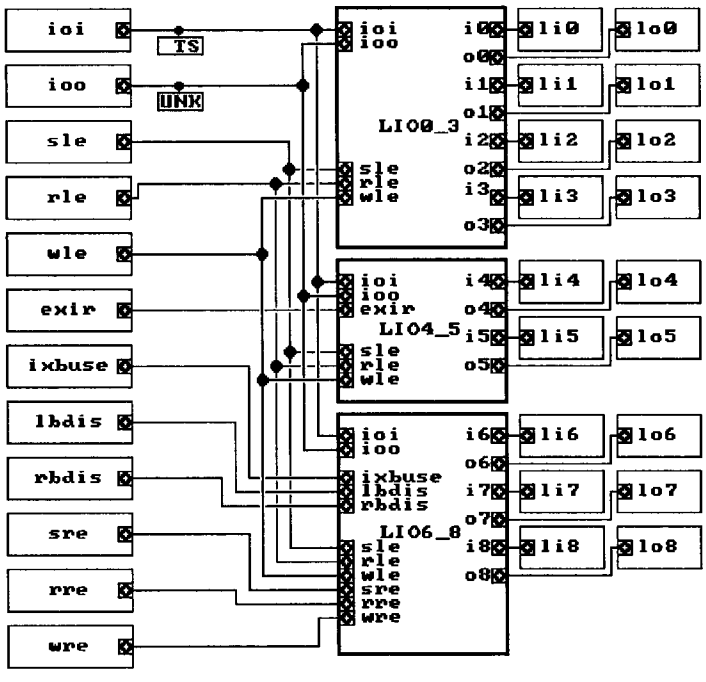


**Figure 78** LIO

LIO0_3 is a block that only has general IO ports. A port is selected with SLE and can be read with RLE and written with WLE. The IRDMUX operator is a multiplexer for the IOI bus signals i0 until i3. LBA captures the contents of the IOO bus at the moment the SLE signal is active and thus keeps the LIO address. The IRDMUX operator uses this address for the multiplex operation on the LIO inputs and the CONTROL operator uses it for the selection of an output register in case of a write.

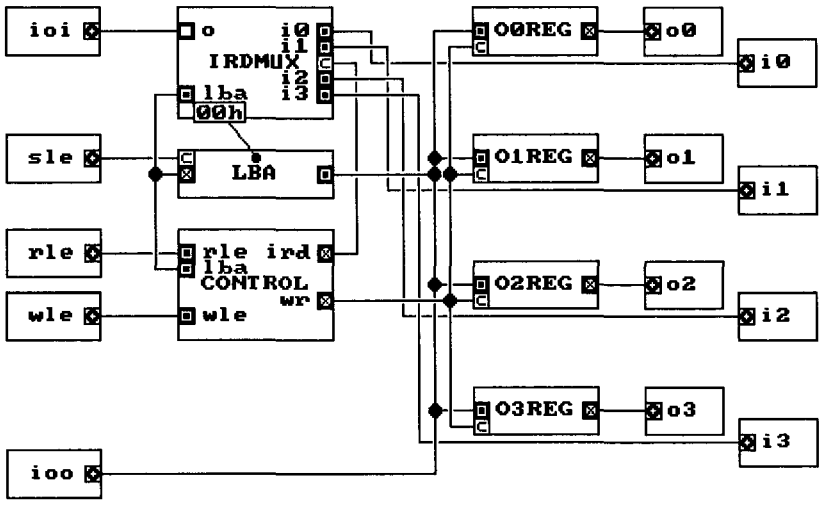


**Figure 79** LIO0_3

LIO4_5 is almost the same as LIO0_3 with the exception of the EXIR output. This output always carries the i4 and i5 and is used as external instruction bus if this mode is selected by the PRGEXT signal in the instruction multiplexer DMUX.

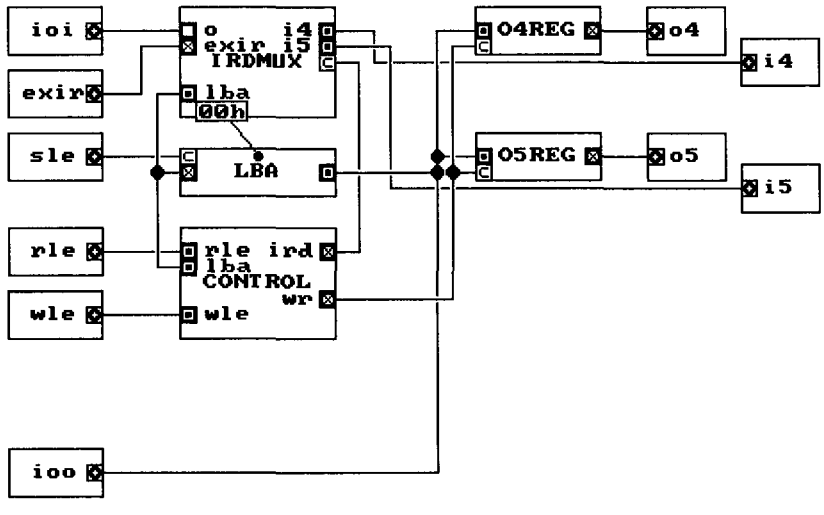


**Figure 80** LIO4_5

LIO6_8 also is a general IO port but in this case with the exception that if IXBUSE is active, LIO6 is used for IOI, LIO7 for IOO and LIO8 for the IO control signals SLE, RLE, WLE, SRE, RRE and WRE. LIO8 also provides the signals LBDIS and RBDIS.

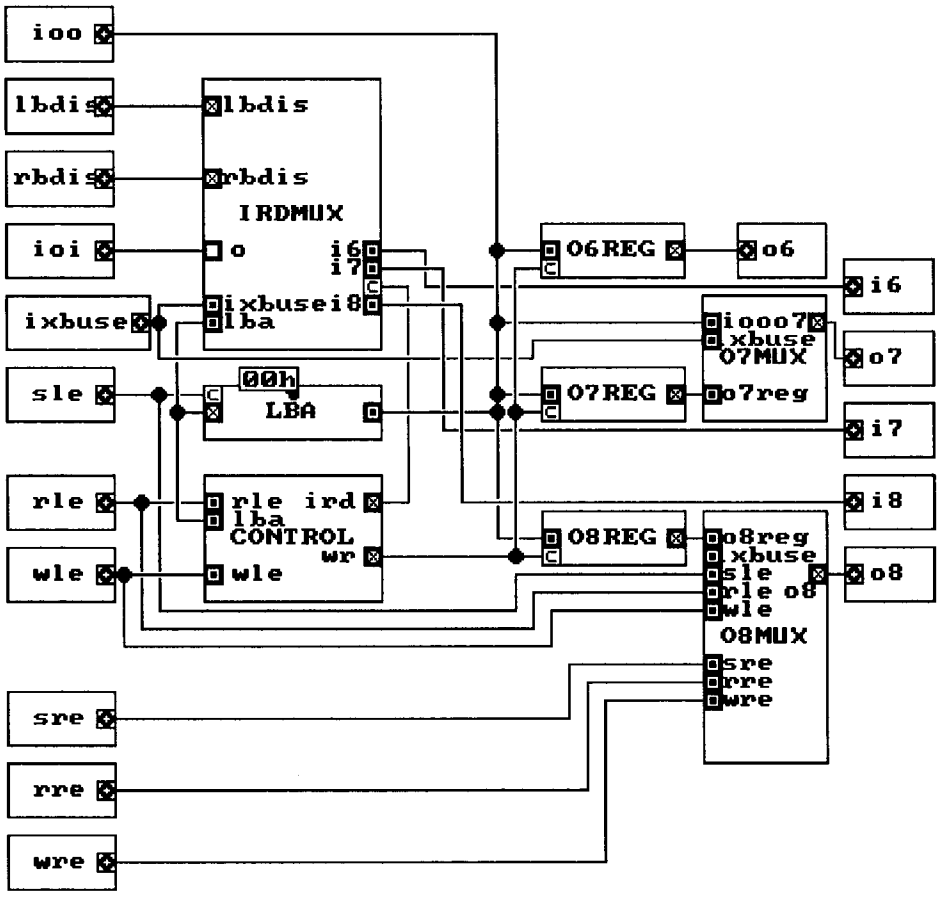


**Figure 81** LIO6_8

## 3.6 Data RAM (DRAM)

### DRAM specification

The MicroDsp test chip provides a data RAM of 256 bytes on the Right Bank of the internal IO bus (IOI, IOO, SRE, RRE, WRE). This RAM includes a RAM address register that keeps the selected address on the Right Bank. This register includes an automatic increment on access mode that is controlled from the PCC by RB address selection : select RB 0XFE = "ON"; POC or select RB 0XFF = "OFF". While in auto-increment mode the RAM may only either read or write, so never read-modify-write.
If external hardware selected (BUSEXT = '1', see 9.4.10) an existing Right Bank address and indicates this by pulling RBDIS HIGH then internal Right Bank access will be disabled. This is done by disabling propagation of chip internal RRE and WRE signals.

### DRAM design

The data RAM has been designed by A. Verschueren. He partitioned the RAM into two halves: BUSSIMU0 and BUSSIMU1. This is done to provide burst access for reading and writing and to provide access direction switching in one cycle. Burst access requires the auto-increment mode to be on.

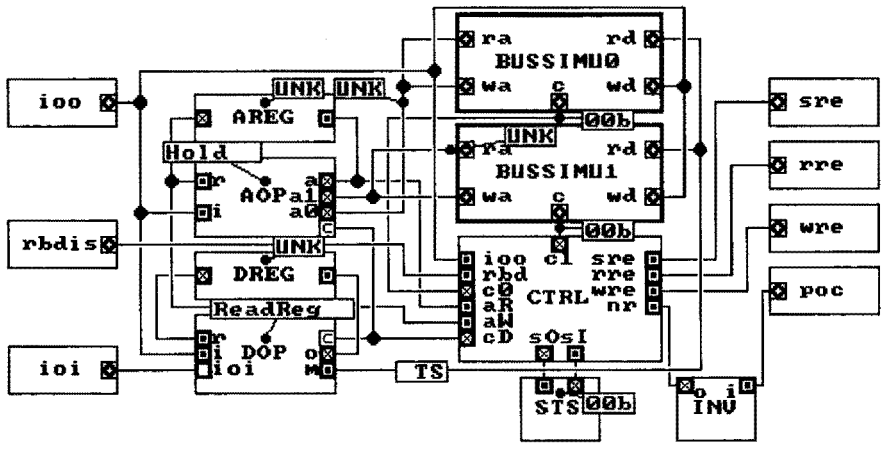


**Figure 82** DRAM

### DRAM simulation

The DRAM has been simulated by A. Verschueren. According to him, his implementation is corresponding the specification [MDSP93].

## 3.7 Program RAM (PRAM)

### PRAM specification

For PCC user program an internal static RAM of 4128 words by 16 bits is available. On board program memory avoids chip-to-chip communication, so that the system can run on maximum speed. Use of static RAM eliminates refresh cycle interrupt and hardware. Size of the RAM of 4160 words by 16 bits will not increase the chip size as minimum silicon size is probably determined by number of pins. Note that the top 64 words are reserved for use by the IICC monitor PCC routines.

If extended program memory size in number of words or in number of bits are required, the PCC assembler supports up to 8192 words by 32 bits : 16 bit PCC instruction code and up to 16 bit extension, additional memory is to be connected externally to the PCC address bits. Extended memory content management is not supported by the MicroDsp test chip and requires additional user hardware. If external program memory is enabled (PRGEXT at HIGH) the full PCC address range of instructions are entered via the pins of ports IO4 and IO5 and the IICC cannot control the PCC. In this mode the chip runs in autonomous operation and starts up at address 0x0000 after POC.

With PRGEXT at LOW internally 4160 * 16 bit on board PCC program memory RAM (generated by ADMC, Natlab) is available. User program can be loaded into PCC program memory space 0x0000..0x0FFF. For monitor actions 64 memory locations are available on 0x1000..0x103F. If the PCC accesses external program memory at an address exceeding 0x103F always external memory will be assumed, independent from the XPRGM input level and external instruction input is done via input IO ports LIO4 and LIO5.

### PRAM design

In IDaSS a RAM can contain only 2048 words. Therefore the IDaSS design of the PRAM is not exactly according to the simulation. But for simulation purposes 2048 words are sufficient. The RAM has an read address input RA and a read data output RD. If RD always gives the data at address RA. This data is delayed one clock in the DBUF register and output on the INIR output. This register is in fact the instruction register of the PCC. For data uploading via the IICC the RDIR output is used. For data downloading address WRAD and data WRIR are used on the moment WREN enables writing in the RAM.

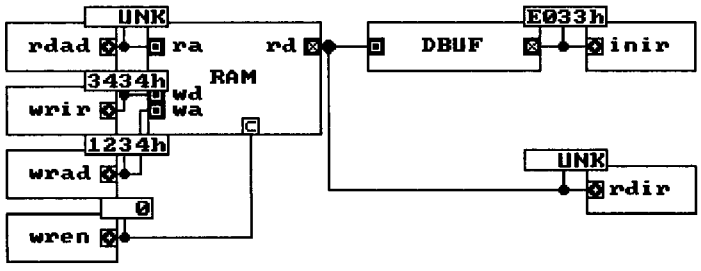


**Figure 83** PROGRAM

The program RAM is accessed via two multiplexers: the data bus multiplexer (DMUX) and the address bus multiplexer (AMUX). The schematics of these multiplexers are given in the next figures.

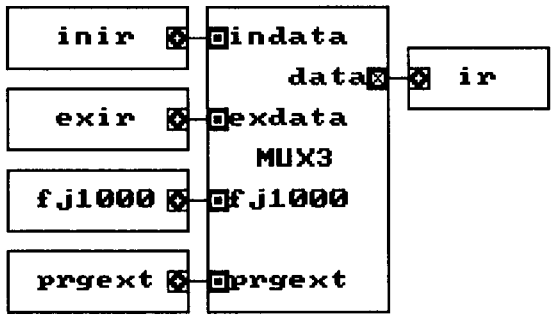


**Figure 84** DMUX

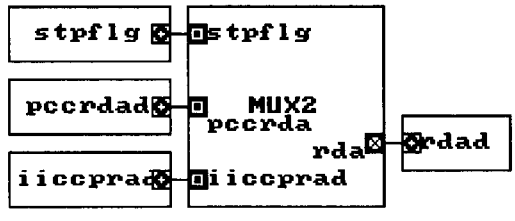


**Figure 85** AMUX

## PRAM simulation

The PRAM has been simulated briefly because it is only used for IDaSS simulation purposes. It is possible to read instructions for the PCC. Also the downloading and uploading facility has been tested.

# 4 VHDL

The VHSIC Hardware Description Language (VHDL) is a hardware description language developed, starting in 1981, by the Very High Speed Integrated Circuits (VHSIC) Program Office of the Department of Defense for use as a standard language in the microelectronics community. This language represents a new step in the evolution of language support for hardware design. The recognized need for managing the complexity of information needed for digital design has driven the development of VHDL.

VHDL is a comprehensive language that allows a user to deal with design complexity. Design, and the data representing a design, are complex by the very nature of a modern digital system constructed from VLSI chips. VHDL is the first language to allow one to capture all the nuances of that complexity, and to effectively manage the data and the design process. A major power of VHDL is that it is a standard. Thus, industry can more easily communicate designs among participants in a design process. This ability to communicate designs is equally important in the research field, since, with VHDL, collaboration between researchers at various institutions becomes easier. The scope of VHDL covers the description of architectural description to gate level description. The language is hierarchical and mixed-level simulation is supported. The concepts embodied in the timing model for the language mirror real hardware -- the VHDL models of designs behave like real hardware. Because VHDL is an IEEE standard, the language will have a significant effect on life-cycle support of product described in VHDL. At high levels of abstraction, the language makes an good specification medium for future designs to be created in new technologies or with alternative architectures. At lower levels of abstraction, the language serves well as a specification of what is to be fabricated.

### IDaSS to VHDL

The MicroDsp has been designed in IDaSS. Philips wants the IDaSS design to be translated into VHDL. VHDL is at this moment the language that Philips uses to describe hardware. The IDaSS design has been simulated with the built-in IDaSS simulator. If the translation into VHDL is done right, VHDL code simulation should give the same results. After simulation of the VHDL design is approved, a silicon compiler can generate a chip layout of the MicroDsp.

### IDaSS to VHDL converter

At the Eindhoven University of Technology an IDaSS to VHDL converter is being developed. This conversion tool is created by W. Kruijtzer of the Digital Information Systems group. It was the intention to use the converter for the translation of the MicroDsp IDaSS description, but the tool was not ready at the moment that it was needed. At that time it only created VHDL files that described the structure of the design. However, it has been possible to use this preliminary version of the converter. The VHDL code that was not yet generated by the converter was the behavioural part of the design. This part of the code had to be entered by myself.

## VHDL simulator Leapfrog

After the VHDL code has been entered, the design has to be simulated. Philips uses the Cadence tool Leapfrog to do this. With Leapfrog it is possible to debug the VHDL code. Also timing figures can be produced.

## General VHDL elements compared with IDaSS elements

If a VHDL frame is created only the behaviour of the elements has to be entered. The VHDL language is of course different from the IDaSS descriptions but there are a lot of similarities. Next we can see some examples of IDaSS descriptions and VHDL descriptions of an eight bit register, address decoding, an operator and a three-state buffer.

register 8 bit continuous output

IDaSS:

```
'PRE_SCALER\PRESCL' is a register.

This register is 8 bits wide and is controlled by an unnamed control input.
The default function is 'hold'.
This register is loaded with unknown values after a system reset.

The value loaded for the 'reset' command is 0.

Control specification:
----------------------v----------------------
%1 load.
----------------------^----------------------
```

VHDL:

```
ARCHITECTURE behaviour OF PRE_SCALER_PRESCL  IS

SIGNAL memory : std_ulogic_vector(7 downto 0) := "00000000";

BEGIN
    controli : PROCESS(x_c, x_i, clk)
    BEGIN
        IF (clk'event AND clk='1') THEN
            CASE x_c is
                WHEN '1'    => memory <= x_i;
                WHEN OTHERS => memory <= memory;
            END CASE;
        END IF;
    END PROCESS controli;

    x_o <= memory;

END behaviour;
```

## address decoding

## IDaSS:

```
'CTRL' is an operator.

This operator has 1 function.
The default function is 'select'.

Text for function 'select' of 'CTRL':
----------------------v-----------------------
cr:=(10h = addr) /\ (r_w not) /\ en.
cb:=(10h = addr) /\ (r_w) /\ en
----------------------^-----------------------
```

## VHDL:

```
ARCHITECTURE behaviour OF FCTL_CTRL  IS

BEGIN
    -- address 10h = 0010000b
    cr <= (NOT addr(0) AND
           NOT addr(1) AND
           NOT addr(2) AND
           NOT addr(3) AND
               addr(4) AND
           NOT addr(5) AND
           NOT addr(6)
          ) AND (NOT r_w) AND en;
    cb <= (NOT addr(0) AND
           NOT addr(1) AND
           NOT addr(2) AND
           NOT addr(3) AND
               addr(4) AND
           NOT addr(5) AND
           NOT addr(6)
          ) AND r_w AND en;
END behaviour;
```

## operator

## IDaSS:

```
'PRE_SCALER\PCCTRL' is an operator.

This operator has 1 function.
The default function is 'pcctrl'.

Text for function 'pcctrl' of 'PRE_SCALER\PCCTRL':
----------------------v-----------------------
pcset:=write
        if0: ((1 width: 7) shl:(prescl from:5 to:7))
        if1: ((1 width: 7) shl:(ioo    from:5 to:7)).
ldpc:=npoc
 if0: npoc,npoc
 if1: (running
  if0: write, write
  if1: (lzero \/ write),((pzero /\ lzero) \/ write)).
----------------------^-----------------------
```

## VHDL:

```
ARCHITECTURE behaviour OF PRE_SCALER_PCCTRL  IS

BEGIN
    controlo : PROCESS (npoc, write, lzero, prescl,
                        ioo, running, pzero)

    VARIABLE m : std_ulogic_vector(2 DOWNTO 0);

    BEGIN
        IF write = '0' THEN
            m(2) := prescl (7);
            m(1) := prescl (6);
            m(0) := prescl (5);
        ELSE
            m(2) := ioo (7);
            m(1) := ioo (6);
            m(0) := ioo (5);
        END IF;
        CASE m IS
            WHEN "000"  => pcset <= "0000001";
            WHEN "001"  => pcset <= "0000010";
            WHEN "010"  => pcset <= "0000100";
            WHEN "011"  => pcset <= "0001000";
            WHEN "100"  => pcset <= "0010000";
            WHEN "101"  => pcset <= "0100000";
            WHEN "110"  => pcset <= "1000000";
            WHEN OTHERS => pcset <= "0000000";
        END CASE;
        IF npoc = '0' THEN
            ldpc <= "00";
        ELSIF running = '0' THEN
            ldpc <= (write & write);
        ELSE
            ldpc <= ((lzero OR write) & ((pzero AND lzero) OR write));
        END IF;
    END PROCESS;
END behaviour;
```

## buffer

## IDaSS:

```
'PRE_SCALER\BUFPSC' is a TS buffer.

This TS buffer is 8 bits wide and is controlled by an unnamed control input.

Control specification:
----------------------v----------------------
%1 enable.
----------------------^----------------------
The default state is disabled.
```

## VHDL:

```
ARCHITECTURE behaviour OF PRE_SCALER_BUFPSC  IS

BEGIN
    controlo : PROCESS (x_c, x_i)
    BEGIN
        IF x_c='1' THEN
            x_o <= x_i;
        ELSE
            x_o <= "ZZZZZZZZ";
        END IF;
    END PROCESS controlo;
END behaviour;
```

## 4.1 IICC and CTU

The IICC unit and the CTU unit have been converted to VHDL. This has been done with the use of the converter that at this moment is in development at the Eindhoven University of Technology. The behaviour descriptions are entered by hand. Before I could do that I had to learn the language VHDL and to master the simulator Leapfrog. This took me less than two weeks. The hand conversion of the CTU and IICC unit needed also two weeks in total. The simulation of the VHDL code has been brief but because the functionality of the code is the same as the IDaSS descriptions this is enough. Furthermore, the VHDL of the IICC is at this moment being expanded with a RESET signal and with two the phase clock signals PH1 and PH2 by E. Rotte. He will test the IICC and CTU in combination with the rest of the MicroDsp.

Figure 86 gives an example of a timing diagram generated by Leapfrog. In this case it is a logical timing diagram of the IICC subblocks FCTL and STAT. The FCTL block can override the signals USRPOC, USRSTOP, USRIRQ and XBUSE. The STAT block gives status information of the MicroDsp. Below the test stimuli for this simulation are given. In the diagram can be seen that the STAT block at address 1Bh is read and that value 40h appears at the data bus. This means that RBDIS is detected HIGH and that can be verified with the RBDIS signal itself. After 500ns the FCTL unit at address 10h is loaded with value AAh which means that the signals POC, STOP, IRQ, and IXBUSE are to be forced to '1'. In the diagram this can be verified by checking these signals.

VHDL stimuli

```
-- LOAD STAT REGISTER

    addr <= "0011011";
    r_w <= '1';
    en <= '1';
    wait for 100 ns;
    en <= '0';
    wait for 200 ns;

-- WRITE FCTL REGISTER

    addr <= "0010000";
    data <= "10101010";
    r_w <= '0';
    en <= '1';
    wait for 100 ns;
    en <= '0';
```

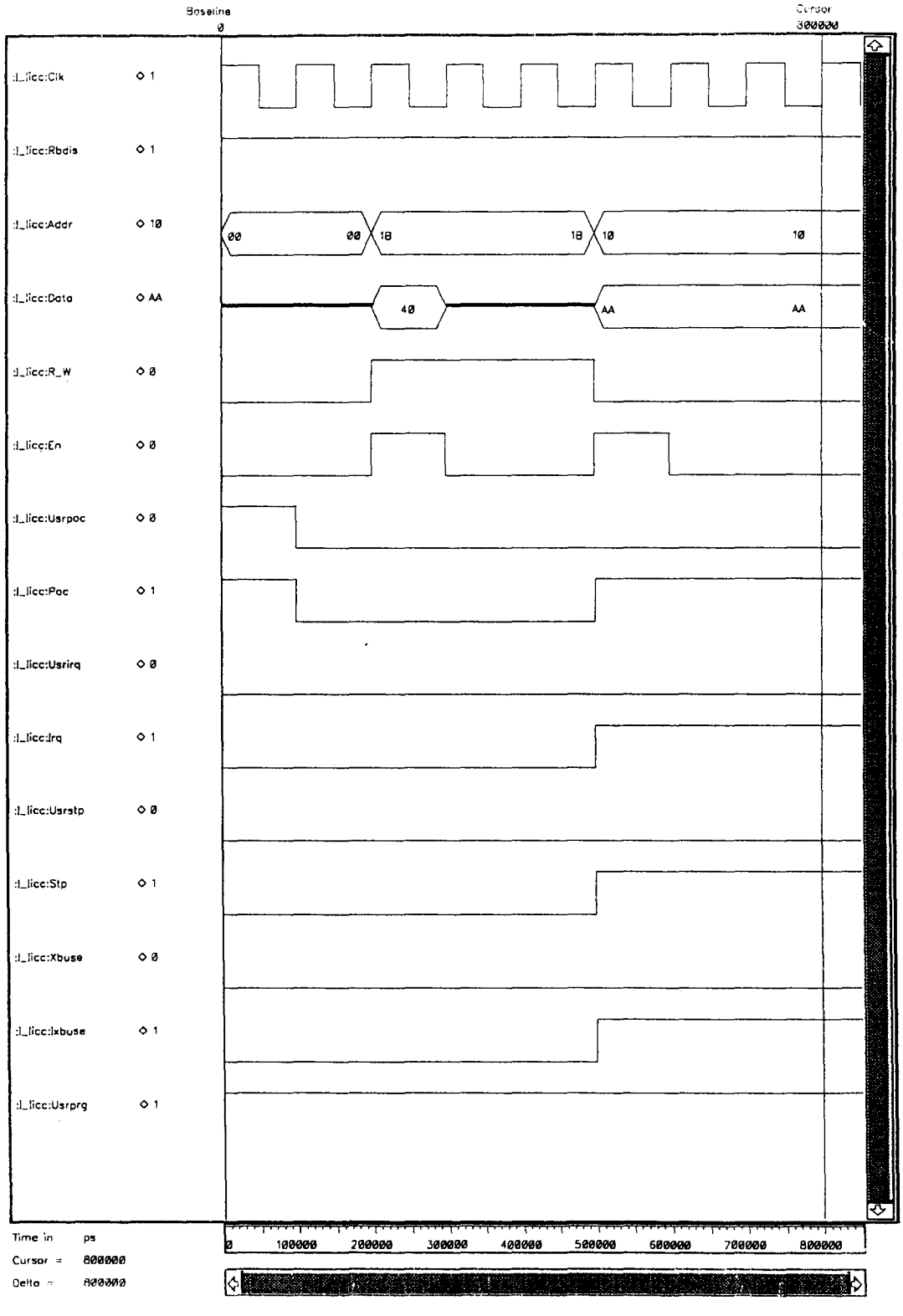


**Figure 86** VHDL simulation of FCTL and STAT units

# 5 Results

The IDaSS evaluation project has had several results.

### Specification of the MicroDsp

A new specification [MDSP94] of the MicroDsp has been made. This was done during the implementation of the MicroDsp in IDaSS. Most changes concern the $I^2C$ and Control unit and the Capture Timer unit. These changes are caused by the fact that the specification was never tested on the required functionality. Therefore a lot of changes had to be made to achieve the requested performance.

### IDaSS design of the MicroDsp

All parts of the MicroDsp have been designed in IDaSS by the Eindhoven University of Technology. The PCC processor and the program RAM have been made by Simons and Vostermans, the AAU coprocessor and the data RAM have been created by Verschueren while the Capture Timer Unit, $I^2C$ and Control, IO ports and MicroDsp toplevel have been built by me.

### VHDL generation of parts of the MicroDsp

It was the intention to describe all parts of the MicroDsp in VHDL. Therefore Simons and Vostermans have converted their IDaSS PCC into VHDL. The VHDL AAU has been made by Brand from Microtel. The Capture Timer unit and $I^2C$ and Control unit have been translated into VHDL by me. A VHDL description of the IO ports has not yet been made. For the program RAM and data RAM simple VHDL modules are used.

### Documentation

The IDaSS design of the MicroDsp is described in this report. It might be useful for everybody who has have to work on the design of the MicroDsp to read chapter 3 of this report. In this chapter the design of the $I^2C$ and Control unit and the Capture Timer Unit is described in detail.

# 6 Conclusions

IDaSS has been used to make a new specification of the MicroDsp and to simulate an implementation of the chip. The processor (PCC) has been designed by Simons and Vostermans from the TUE, the coprocessor (AAU) has been made by Verschueren of the TUE while the $I^2C$ and Control and the Capture Timer Unit were created by me.

The IDaSS implementation of the $I^2C$ and Control and the Capture Timer Unit required many corrections and additions on the MicroDsp specification. Furthermore some errors were found in the silicon versions of the AAU and PCC.

With a tool like IDaSS the efficiency and quality of the definition of new IC's can be improved considerably.

**IDaSS advantages**

- short learning period compared to VHDL
- very interactive, fast feedback on implementation ideas
- runs on a PC
- fast design input
- suitable for block definition and a first specification check
- suitable for "formal" specification by customers
- fast testing of architecture variants possible (pipelining, parallelism, ...)

**IDaSS disadvantages**

- no link yet to Philips Semiconductor tools (IDaSS to VHDL conversion is in development at TUE)
- flexibility and efficiency of implementation is dependent on the IDaSS library which has a limited set of elements compared to VHDL
- only suitable for single clock domain systems
- no asynchronous simulation yet
- no timing check yet (like back-annotation)
- no use of existing back end library blocks (bottum up design)
- does not run on apollo-unix and HP-UX systems (planned)
- packet not very "industrialized" (platforms, hooks, graphic quality, ...)

- no support by "professional" team (for solving application specific problems)
- no abstract data types (data name in stead of signal levels)
- no batch execution yet (re-run of simulation with log-file)
- no similar test-bench yet that can be used for IDaSS and VHDL

If IDaSS is going to be used by Philips as a design tool it has to be accepted by a tool support group such as for example ED&T. A lot of the above mentioned disadvantages then have to be removed.

**MicroDsp**

In any case it is a fact that IDaSS has been used to design the MicroDsp. All units of the MicroDsp have been designed with this tool. This has been done fast compared to the time necessary to develop such a processor in VHDL or in a lower level description language. For example the time needed to design the PCC in IDaSS was two weeks for two man. The design of the $I^2C$ and Control unit and Capture Timer Unit including the rewriting of the specification and learning IDaSS has taken eight weeks for me.

After the IDaSS design was completed, VHDL has been generated for the PCC, AAU, IICC and CTU. These last two blocks are generated with the use of a VHDL-frame generator and will not only be used for simulation purposes but also for the actual design of the chip. The use of the frame generator speeded up the generation of the VHDL code very much. The time that I needed necessary to generate the rest of the VHDL (behavioural code) of the IICC and CTU was less than two weeks. Considering the fact that I had to learn the VHDL language before I could start generating the code, this is rather fast.

The design of the IICC and CTU resulted in the following number of flip-flops per block.

| Block | Flip-flops |
|---|---|
| **IICC:** | |
| IIC | 26 + state controller |
| MADT | 34 |
| FCTL | 8 |
| PAD | 8 |
| IOBUS | 24 |
| XCTL | 8 |
| STAT | 8 |
| MAIL | 9 |
| MONCTRL | 1 |
| STPCTRL | 1 |
| | --- |
| Total: | 127 + state controller |

| | |
|---|---|
| **CTU:** | |
| Time-base counter | 16 |
| Prescaler | 23 |
| Timerreg 1 | 19 |
| Timerreg 2 | 19 |
| | --- |
| Total: | 77 |

Finally I conclude that IDaSS has been very useful developing designs that were not fully specified in a short period of time. These designs can be very fast converted to VHDL with the use of the VHDL converter. This VHDL will probably be synthesizable with the use of a silicon compiler like Synergy.

# Literature


[MDSP93] **J.A.A. den Ouden**, MicroDSP : PCC plus AAU test chip specification 1.1, Philips Semiconductors, 15 Sept. 1993

[MDSP94] **J.A.A. den Ouden**, MicroDSP : PCC plus AAU test chip specification 1.2, Philips Semiconductors, 18 Jan. 1994

[PCC93] **J.A.A. den Ouden**, Peripheral Controller Cell specification, Philips Semiconductors, 8 Apr. 1993

[AAU93] **J.A.A. den Ouden**, Arithmetic Accelerator Unit specification (4.1), Philips Semiconductors, 12 Nov. 1993

[$I^2C$92] **H. Schutte**, The $I^2C$-bus and how to use it (including specification), Philips Semiconductors, Jan. 1992

[IDaSS] **A.C. Verschueren**, IDaSS for ULSI, V0.08d, Eindhoven University of Technology, 20 July 1990

[VHDL] **R. Lipsett e.a.**, VHDL: Hardware description and design Kluwer Academic Publishers, 1993