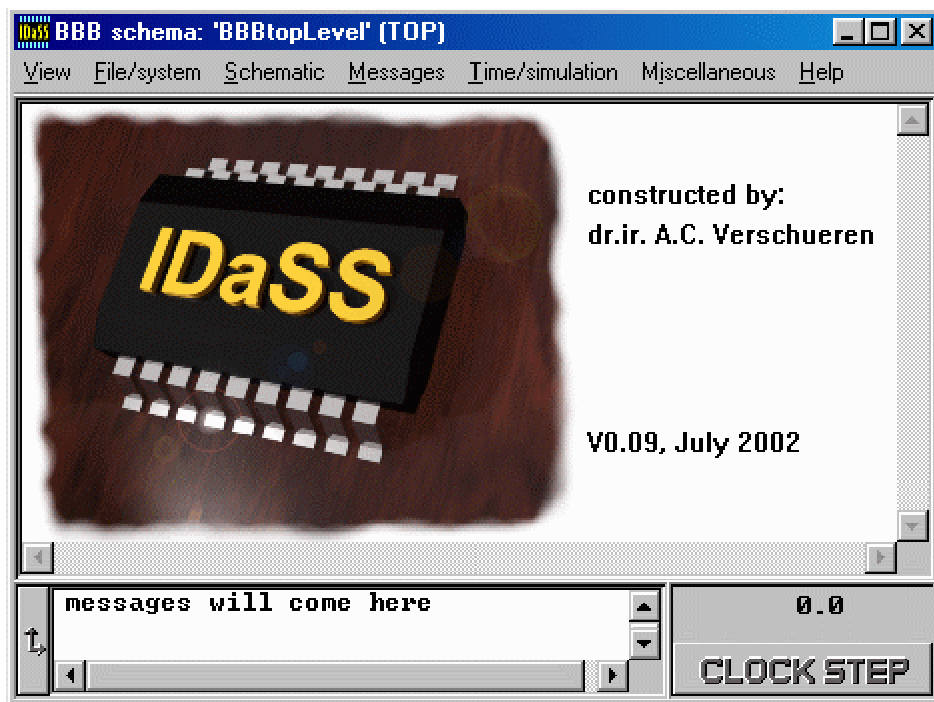


Interactive Design and Simulation System

version 0.09 for VisualWorks Smalltalk version 5i.4 and up

Short form manual



written by

dr.ir. A.C. (Ad) Verschueren

february 1999 - february 2003

Introduction

This is the so-called 'short form manual' for the *'Interactive Design and Simulation System'* (shortened to 'IDaSS' which is pronounced 'eye-dasss'), a program which allows you to design and simulate complex digital circuits like microprocessors.

IDaSS uses a mix of graphics and texts to describe a design. There are other programs which allow that too. IDaSS uses special languages for the textual descriptions. There are other programs which also use so-called 'proprietary' languages, but those used by IDaSS are highly specialised and focused to describe specific forms of behaviour - simple, yet powerful. The main difference lies in the 'I' of 'IDaSS'. Design and simulation are integrated and totally interactive, there is no fancy button to compile a system and start simulating it. In IDaSS, simulation is continuous, you cannot even turn it off!

With IDaSS, you can use two methods for describing your design, one works like a standard programming language (only with special variable types and operators), the other works at the 'Register Transfer Level' (where you work directly with registers, memories, combinatorial logic 'operators', buses and finite state machines). Both of these methods can be used in combination, but only Register Transfer Level designs can be converted into real hardware. IDaSS can generate the necessary files in 'standard' languages like VHDL and Verilog for you. You will need other programs to convert these files into chip layouts or program files for programmable logic devices, though.

This is the 'short form manual', so is there a 'long form manual'? Well, no. To describe everything available in IDaSS and tell you how to use it would give a boring document at least four times as thick as this one. IDaSS invites you to try it out and learn by doing (and hitting the **[F1]** button a lot to get help or just more detailed information). If you make a mistake, don't despair. You are well protected against losing your work with a single keystroke or mouse click.

This manual is intended to give you a basic understanding of how IDaSS works and wet your appetite for starting your own designs and exploring the capabilities of IDaSS further than described here. We do assume, however, that you have a basic knowledge of digital circuit design (and know an FSM when you see one). In fact, IDaSS was used successfully in a second trimester, first year course at Eindhoven University to let students design their own (working!) microprocessor, so it can not be that hard...

If you are in a hurry, just start reading this manual up to section 5.9 ('Building a system part 3: buses and commands'), skipping those parts you already know or just don't care about. Then jump to chapter 7 for a do-it-yourself demo and come back later for more leads and overview information.

Have fun! Ad Verschueren, july 2002

Contents:

1.	LEGAL STUFF	2
1.1	Bug reporting	2
1.2	Important places and files on the internet.....	2
2.	IDaSS INSTALLATION	4
2.1	The stand-alone version.....	4
2.2	The VisualWorks 'parcel' version.....	5
2.3	Known problems.....	6
2.4	Files in the distribution library	6
3.	GENERIC IDaSS FEATURES	10
3.1	Some more detailed features.....	11
4.	USER INTERFACE SPECIFICS.....	13
4.1	Floating and position sensitive menu's.....	13
4.2	The IDaSS help window	13
5.	USING IDaSS.....	15
5.1	Starting IDaSS.....	15
5.2	Stopping IDaSS.....	15
5.3	Saving and restarting the working environment.....	16
5.4	Main window components	17
5.5	Loading a system	18
5.6	Saving a system (part).....	18
5.7	Building a system part 1: blocks.....	19
5.8	Building a system part 2: connectors.....	19
5.9	Building a system part 3: buses and commands.....	20
5.10	Comments, comments and documents.....	21
5.11	Saving screen space and working time.....	21
5.12	Debugging and trace file generation.....	22
5.13	Timing simulation and optimisation	22
5.14	Converting an IDaSS design into real hardware.....	25
5.15	'Properties'	26
6.	BLOCKS WHICH CAN BE USED IN A DESIGN.....	27
6.1	Registers	27
6.2	Operators.....	28
6.3	State controllers	30
6.4	Memories	32
6.5	Three-state buffers	34
6.6	Constant generators.....	35
6.7	Algorithmic Level ('AL') blocks.....	35
6.8	Sub-schematics	36
6.9	Multiple schematics.....	37
6.10	Control inputs (not a block, but important)	38
7.	DO-IT-YOURSELF DEMO.....	40

1. LEGAL STUFF

The '**Interactive Design and Simulation System**' is
1988-2001 Eindhoven University of Technology
2002 dr.ir. A.C. Verschueren

Written by dr.ir. A.C. Verschueren,
E-mail: ***idass@xs4all.nl***

This software is distributed on an as-is basis, without any further implied or express warranties. We do not accept any responsibilities regarding it's use in any way.

...but we do our best! IDaSS is a research tool, which means that it is regularly updated with new features, but also that these new bells and whistles may have some childhood bugs (as like in every program, but we do admit that we are not perfect...).

1.1 Bug reporting

If you suspect a bug, check if you are using the latest version by comparing your ***errorlog.txt*** file with the one distributed from the download page in the main IDaSS distribution files ***idass5i4.exe*** and ***source5i4.zip***. If your version is older, it might be that your bug has been found and fixed (the file contains a description of bugfixes in chronological order). We welcome bug reports, send them to the e-mail address indicated above. If you're using the 'stand alone' version of IDaSS, a system crash will generate an ***error.log*** file - include this one with your error report!

1.2 Important places and files on the internet

The IDaSS 'homepage':

<http://www.xs4all.nl/~averschu/idass>

The IDaSS download page with links for downloading (sorry, no direct FTP possible for now):

<http://www.xs4all.nl/~averschu/idass/idassdownload.html>

The installation program file which contains IDaSS itself as a stand-alone version running under Windows 95/98/NT/ME and also in a Windows emulator under Linux:

idass5i4.exe

The library file which contains the IDaSS 'parcel' (including source code) and all the other files needed for installing IDaSS inside an existing VisualWorks Smalltalk (version 5i.4 and up) environment:

source5i4.zip

A separate text file explaining how to get both versions up and running:

instal5i4.txt

The VisualWorks Smalltalk site, from which you can download (complete) evaluation versions (and also buy commercial licences) of the VisualWorks system. VisualWorks can run on a large number of computer platforms, including Windows (95/98/NT/ME), Linux, Power Macintosh and a number of UNIX boxes (HP-UX, Sun Solaris and others):

<http://www.cincom.com>

The Xilinx site, from which you can download the (freeware) Xilinx WebPACK FPGA synthesis toolbox which is targeted by the ***spartan2e.aft*** Verilog source code generator delivered with IDaSS:

<http://www.xilinx.com>

2. IDaSS INSTALLATION

IDaSS can be brought up-and-running in two different ways:

- 1) As a 'stand-alone' system, without access to the VisualWorks Smalltalk environment in which it was built.
- 2) As an application 'parcel' to be installed in an existing VisualWorks Smalltalk environment.

Both of these have their own advantages and disadvantages:

- The stand-alone version is a lot smaller (approximately 9 Megabytes) than the VisualWorks environment with IDaSS installed as application (which easily exceeds 20 Megabytes). The stand-alone version is also easier to install.
- The stand-alone version only runs on a Windows (95/98/NT/ME/XP) equip-ped PC, or in a Windows emulator under Linux. VisualWorks Smalltalk can be obtained for a lot of other 'platforms' - IDaSS should run on all of them when installed within this environment as an application parcel.

First, some general warnings:

Warning for Windows-NT, Linux and UNIX users:

Filenames are case sensitive in your operating system! IDaSS looks for an ***idass.ini*** file (all lowercase), so you will have to change the name of that file if it is not all lowercase (it should be so after installation). Also, the files mentioned in this configuration file must match the actual file(path)s casewise.

Remark for all users:

We have seen that VisualWorks is not very consistent in what it calls its 'home directory'. If IDaSS can not find the ***idass.ini*** file, you will be asked to find it using a Windows-95 style file opening dialog. This also allows you to select a configuration file which is not all lowercase.

2.1 The stand-alone version

- 1) Download the ***idass5i4.exe*** file from the 'download' section of the IDaSS homepage <http://www.xs4all.nl/~averschu/idass>.
- 2) Run this file to start the automatic installation procedure (which unpacks all files, creates an icon, program group and uninstaller).

During this procedure, you can choose between a basic IDaSS installation or one preconfigured to generate Verilog targeted towards the Xilinx WebPACK tools for either Xilinx Spartan-II/200-PQ208 or Spartan-IIE/300-PQ208 FPGA's. At the end of this process, it is advisable to edit the *idass.ini* file to select (at least) the correct time zone.

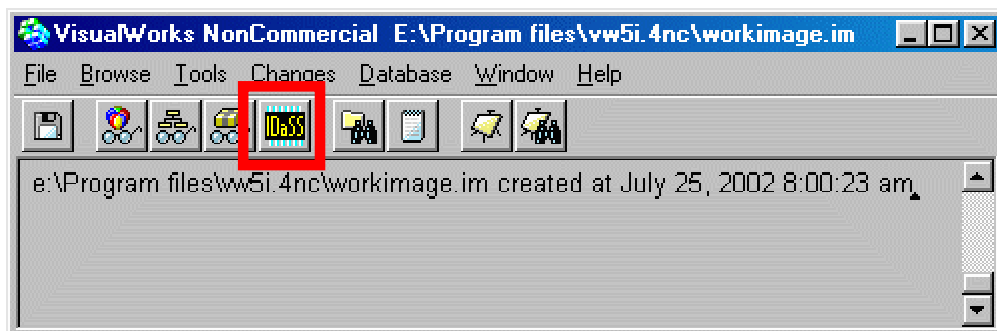
- 3) Click on the IDaSS icon created in step 2) to start the system. This should immediately open a top level schematic editor window. Pull down any menu from the menu bar and select '**install IDaSS from idass.ini**' to complete the installation and save the system with the settings chosen in the *idass.ini* configuration file.

You can manually (re)initialise IDaSS by choosing 'miscellaneous', 'system management', 'initialise IDaSS' from the menu bar of this schematic editor window.

2.2 The VisualWorks 'parcel' version

- 1) Obtain a version of the Visualworks Smalltalk 5i.4 environment from <http://www.cincom.com> which can run on your machine.
- 2) Get VisualWorks Smalltalk up and running, preferably using all their default settings. Do not forget to set the Visualworks home/installation directory (launcher window - the one with the buttons - '**File**' menu, '**set VW home**').
- 3) Download the *source5i4.zip* file from the 'download' section of the IDaSS homepage <http://www.xs4all.nl/~averschu/idass/> and unzip it to your main VisualWorks Smalltalk directory (which creates a few new directories, make sure the 'use path names' option is set).

Then follow instructions in the text file *install5i4.txt* contained in the *source5i4.zip* library (as usual, the *idass.ini* file should be modified, if necessary after copying it from *spartan2.ini* or *spartan2e.ini*!). If all went OK, a new button should appear in the launcher window:



Click this button to open a top level IDaSS schematic editor window. Pull down any menu from the menu bar and select '**install IDaSS from idass.ini**' to complete the installation and save the system with the settings chosen in the *idass.ini* configuration file.

You can manually (re)initialise IDaSS by choosing 'miscellaneous', 'system management', 'initialise IDaSS' from the menu bar of this schematic editor window.

2.3 Known problems

- Some machines have problems with the menu's. Clicking the right mouse button does not give the menu for the sub-window or schematic object pointed at, but the window menu instead. Pressing the [CTRL] button while clicking solves this - installing another mouse driver may also help (under Windows-95, that is). If that doesn't help, you can swap both menu's with a setting in the '**miscellaneous**' / '**environment settings**' menu from the menu bar of any IDaSS schematic editor window.
- The help file *idass.hlp* is not complete, most of the descriptions of window parts and lower level menu's are not containing very meaningful information yet.
- If you change the setting of the IDaSS text window font size, fixed-size windows depending on this size do not change their size automatically. You have to close and re-open them. Some platforms have problems with font size changes and offer only a single size.
- Opening the help window with a menu popped up (which provides help on this menu) closes the menu immediately, while it would be more logical to keep the menu on the screen - most windowing systems regard a menu as something extremely transient which should be removed from the screen when a window opens.

2.4 Files in the distribution library

To be placed in the main IDaSS installation directory (or 'home' directory of VisualWorks Smalltalk, if you are installing IDaSS as an application parcel):

idass.ini configuration file telling IDaSS where to find other files during installation (renamed to *idassbasic.ini* if one of the other *.ini* files replaces this one during stand-alone installation)

- spartan2.ini*** ditto, but now with settings to install a Verilog code generator targeted towards the Xilinx WebPACK tools for generating Xilinx Spartan-II/200-PQ208 FPGA configuration files. Replace ***idass.ini*** with this file before running IDaSS for the first time if you want to have this by default (can be done automatically during stand-alone installation)
- spartan2e.ini*** as ***spartan2.ini***, but now for generating Xilinx Spartan-IIE/300-PQ208 FPGA configuration files. Replace ***idass.ini*** with this file before running IDaSS for the first time if you want to have this by default (can be done automatically during stand-alone installation)
- idass.im*** initial image file containing a 'stripped' VisualWorks system with IDaSS installed (only used for stand-alone version)
- errorlog.txt*** text file containing release notes: capabilities added and bugs removed with each release
- instal5i4.txt*** text file explaining the installation processes
- licence.txt*** text file holding the licence information: READ THIS ONE!
- uninst000.exe*** program to uninstall IDaSS (only for stand-alone version)
- uninst000.dat*** data file for uninstallation (only for stand-alone version)

To be placed in a ***bin*** sub-directory of main IDaSS installation directory:

- visual.exe*** VisualWorks 'non-commercial' execution engine (only for stand-alone version)
- herald.bmp*** graphic image used both for the splash screen as well as empty IDaSS schematic editor windows (may overwrite file with same name in VisualWorks base system)
- IDaSS.ico*** icon image file for stand-alone version
- linkmaster.dll*, *vwft16.dll* and *vwft32.dll*** module libraries used by ***visual.exe***, licenced (and included) only for running the IDaSS stand-alone version

To be placed in a ***support*** sub-directory of the main IDaSS installation directory:

- idass.hlp*** hypertext linked help file for IDaSS (*not* Windows help compatible)
- shortman.pdf*** 'short form' manual - this text

<i>idass.tec</i>	default technology file with time delay calculations (based on a 2 nanoseconds basic NAND gate delay) and memory size and memory port restrictions
<i>compass1.aft</i>	'Alien File Template' for generating Compass VHDL equivalents of an IDaSS design - including RAM/ROM macrocells and ASIC pad ring
<i>spartan2e.aft</i>	'Alien File Template' for generating Verilog equivalents of an IDaSS design targeted towards Xilinx WebPACK tools to generate Xilinx Spartan-II(E) FPGA implementations
<i>spartan2e.pdf</i>	manual explaining how to use the <i>spartan2e.aft</i> Xilinx WebPACK Verilog generator
<i>example.lfd</i>	simple simulation trace file format description
<i>vhdlenv.lfd</i>	simulation trace file format description which generates VHDL test environments (well, almost)
<i>verilog.lfd</i>	simulation trace file format description which generates a file compatible with the Verilog <code>\$readmemb</code> command

To be placed in a ***designs*** sub-directory of the main IDaSS installation directory (this part may change!):

<i>up8048n.des</i>	an almost complete implementation of the Intel 8048 processor core, minus I/O instructions and I/O hardware. It is a good example of a complete and not too simple Register Level system
<i>al8048.des</i>	also an 8048 but this time including I/O, built around a single Algorithmic Level block
<i>rom8048.hex</i>	Intel HEX file containing a small test program to be loaded in the 8048 example designs
<i>ram8048.hex</i>	Intel HEX file with initial RAM contents for the 8048 designs
<i>communic.des</i>	an example which shows the use of sub-schematics, re-use of schematics, use of parameters and the concept of communicating finite state machines
<i>alexmple.des</i>	is intended to show all the constructs available in the Algorithmic Level block language. The comment for the complete design (named 'AL_test') contains instructions on how to view these constructs in full execution.
<i>pipeline.des</i>	contains a fairly complete and synthesizable pipelined processor with a 5 stage pipeline (see resulting ASIC on http://www.xs4all.nl/~averschu/idass/pipechip.html)

pipeline.hex contains a small test program to be loaded in the ROM of the ***pipeline.des*** design

3. GENERIC IDaSS FEATURES

This is a slightly modified version of the information on the IDaSS homepage.

In short:

"IDaSS is an Interactive Design and Simulation System for digital circuits, targeted towards VLSI and ULSI designs of complex data processing hardware."

Its main features are the following:

Graphics to describe system structure.

IDaSS describes a design as a hierarchy of schematics. A schematic contains elements like registers, RAM's, ROM's, combinatorial (ALU-like) 'operators' and associated Finite State Machine controllers.

Data transfer is described by drawing lines representing data buses. Complete schematics may be re-used in a system.

Textual languages to describe behaviour.

IDaSS uses specially tailored languages to describe the behaviour of system elements which can have complex behaviour, like combinatorial operators, FSM controllers and local control signal decoders. These languages have a lot of common constructs and are kept as simple as possible (for instance, there are no reserved words).

Completely integrated design and simulation.

IDaSS simulates while designing. Placing an element on a schematic immediately simulates it's behaviour. Saving a textual description immediately compiles this into behaviour and simulates it. There is no need to restart simulation if a design error is found, just fix the error, bring the system into a usable state and continue simulation.

IDaSS can be compared to an electronic breadboard without its usual shortcomings:

- *Never shortage of parts, the parts you use are always of the correct size and functionality.*
- *Complex parts and wide datapaths form no problem at all.*
- *Power-on insertion and removal without destroying anything.*
- *Errors lead to a simulation halt, not a blown up device.*
- *System operation can be observed through an unlimited supply of test probes.*

- *System state can be modified easily using interactive editing facilities.*

Integrated higher abstraction level design.

A so-called 'Algorithmic Level' block allows you to design (and simulate) complete datapaths and associated controllers with a description resembling a structured programming language like Pascal or C.

Such a block contains a set of local variables (operating like registers or memories) and a set of procedures operating on those variables. These blocks are placed in the normal Register Transfer Level IDaSS design environment and can communicate and synchronise with this environment.

Friendly to both novices and experienced designers.

IDaSS is completely menu driven and its textual languages are easy to learn. Yet, the system is powerful enough to handle very complex designs and allows libraries of reusable components and sub-systems to be built. A comprehensive help system with over 1200 hypertext linked pages is delivered with the system (just press **[F1]** to open the help window).

On-the-fly syntax and semantics checking.

IDaSS does not allow any operation which can be regarded a syntax error (like connecting buses of different widths). IDaSS will warn the user if he/she tries to do something which may result in system failure (like asking a system element to perform a function which has not been defined yet).

Direct path to hardware.

IDaSS design files can be converted into a format suitable as input for silicon compilers. Synthesizable VHDL and Verilog output is available with the 'alien file generation' functionality which uses a rulebase ('template') file for total flexibility.

3.1 Some more detailed features

- The maximum width (number of bits transferred or stored in parallel) of datapath elements is not fixed, it can be changed from the '**miscellaneous**', '**environment settings**' menu on the menu bar of any schematic editor window.

NOTE: although the width can be set well above 64 bits, viewers (virtual test probes) and memory editors will only handle widths up to 64 bits (for now).

- You can perform time-consuming operations (like a long simulation run, generating documentation or VHDL/Verilog generation) in the background. Limitations are that this can be done for one operation per simulation session (a system edited) and that only one file in/out or documentation generation operation can be going on at the same time. Window operations (except closing) are allowed while running an operation in the background.
- Text cut, copy and paste runs through the platform copy/paste buffer (the 'clipboard' under MS-Windows). This allows you to exchange text (fragments) with other programs.

Schematics (either as a whole or just the part shown in the window) can also be placed *as graphics* in this buffer. Direct printing of schematics (or text) is not possible, though.

- For total cross-platform compatibility, IDaSS can read any kind of text file (DOS, UNIX or Mac). By default, it will generate the kind of text file fitting to the platform it is running on. This can be overruled with an entry in the '**environment settings**' sub-menu of the '**Miscellaneous**' menu on the schematic window menu bar.
- Automatic, periodic saving of a design is possible to prevent loss in case of a system crash (see '**Miscellaneous**', '**auto save settings**').

4. USER INTERFACE SPECIFICS

The VisualWorks Smalltalk environment in which IDaSS is implemented automatically adjusts its user interface to match that of the host operating system under which it is running (and you can even overrule the automatic switching, if you want). Window handling and text editing operations are totally standard and should need no further explanation. Still, there are several things worth mentioning here.

4.1 Floating and position sensitive menu's

Although all IDaSS windows have a menu bar to provide access to the most often used menu's, it was sometimes necessary to provide a higher level of selection between different menu's.

In schematics, each schematic element (block, connector, bus segment or viewer) has its own menu. You access this menu as a floating menu of your operating system (point the mouse at the element and click a specific button - the right button under Windows-95, for instance).

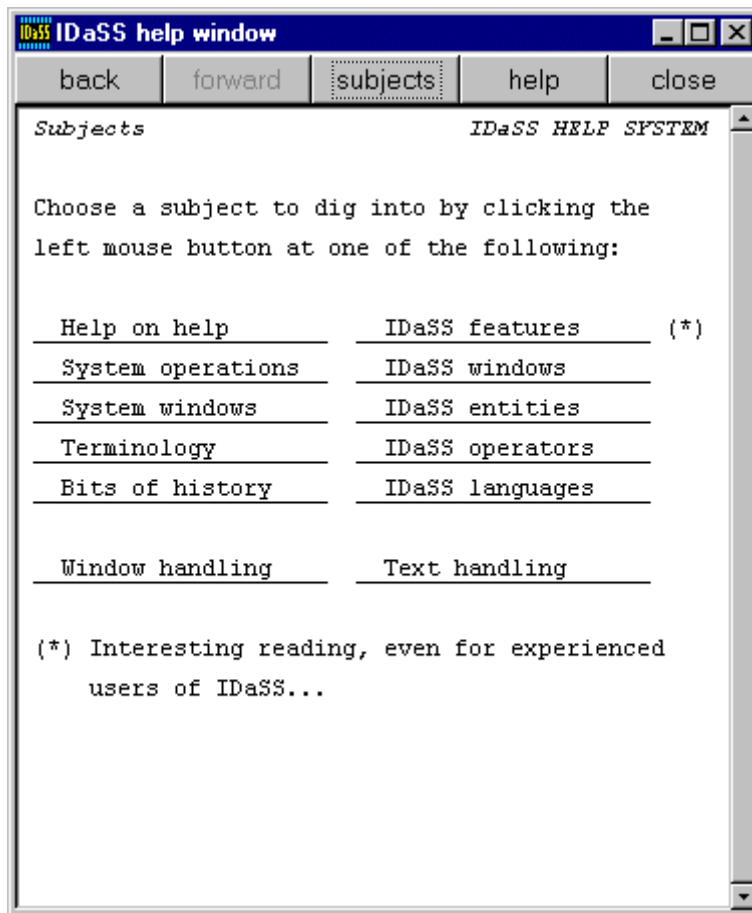
If clicking this way gives the window menu, try holding down the [CTRL] key while clicking. You can permanently swap the window menu and the intended floating menu with an entry in the 'Miscellaneous', 'environment settings' menu from the menu bar of any schematics editor window.

Sometimes, the exact position of the mouse when the menu is called up is important, for instance when making a side connection to a bus segment.

In text lists of objects which also show the object states, selecting an entry in this list will immediately call up the menu for this entry.

4.2 The IDaSS help window

Pressing [F1] with a menu or prompter visible will open a help window containing information on this menu or prompter. If no menu or prompter is visible, pressing [F1] will open a help window containing information for the screen element the mouse was pointing at that time.



Pages contain links to other pages denoted by pieces of underlined text. You can go to a linked page by clicking the select button (in most cases the left one) while pointing at the link.

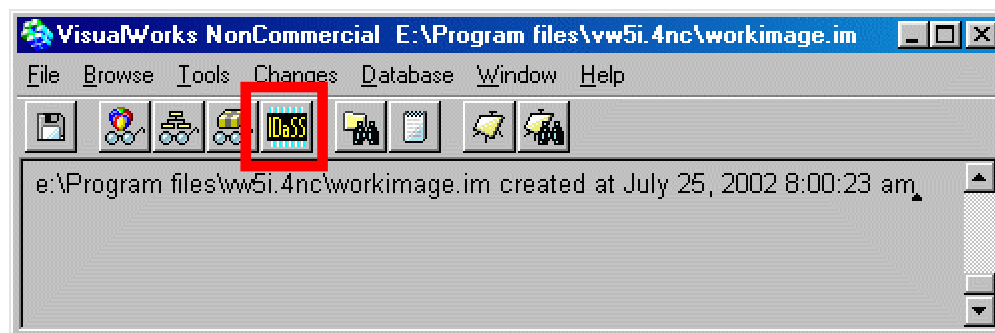
5. USING IDaSS

This chapter provides an introduction on how to start and stop IDaSS, load and save designs, create new (or modify existing) designs and a lot of other interesting features.

5.1 Starting IDaSS

On the stand-alone version, clicking the 'IDaSS' icon (set up during installation) will open a top-level schematic window in which you can start editing and simulating your design.

After installing the IDaSS parcel in an existing VisualWorks Smalltalk environment, the Smalltalk 'launcher' window should have a button labeled 'IDaSS' which performs the same function:



With IDaSS running, you can always open a new (empty) top-level schematic editor with from the menu bar menu '**File/system**' entry '**new session**'.

5.2 Stopping IDaSS

Closing a top-level schematic editor window will stop an edit/simulation session and also close all other IDaSS windows related to this session. IDaSS will ask if you are sure you want to close this window because you will lose your work if you did not save the design.

On the stand-alone version of IDaSS, closing the last top-level schematic editor window also exits the complete IDaSS environment automatically.

In case IDaSS is installed within an existing VisualWorks Smalltalk environment, quitting this environment is done with '**File**', '**Exit VisualWorks**' from the menu bar of the Launcher window.

5.3 Saving and restarting the working environment

The VisualWorks Smalltalk 'image' contains the complete working environment, including IDaSS and any IDaSS design sessions you are performing. When saving this image in a file, it will record system mode setting changes you have made, windows you have opened and even the exact simulation state of the systems you're working on.

If you change system settings (for instance the maximum datapath element width), you must save the 'image' to record these changes for the next time you start the system!

For the stand-alone version, you can save the image from with **'File/system', 'save image'** from any top level schematic editor window menu bar.

If you installed IDaSS as 'parcel' in a complete VisualWorks environment, you can also save the image file with **'File', 'Save as'** from the launcher window or when you exit VisualWorks Smalltalk with **'File', 'Exit VisualWorks', 'Save then exit'** from the same window.

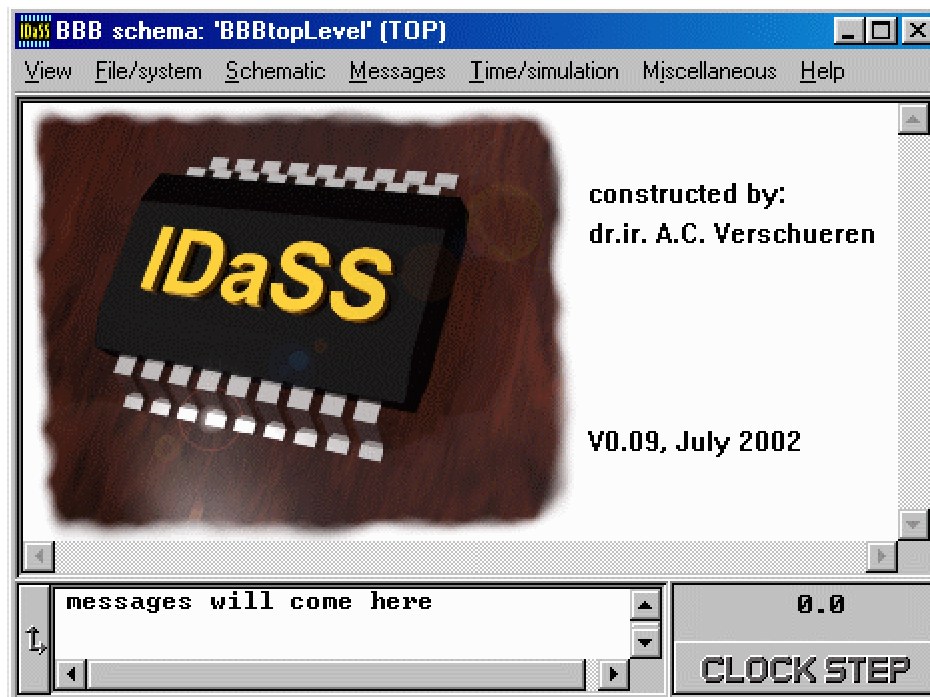
After this you should give the base name for the image file (with as default the name of the image file which is being used, do *NOT* give the **.im** extension). Using different names allows you to have several image files (for instance each containing work on a different project).

Image files are rather large, well over 5 megabyte - although convenient, they are NOT meant as a medium to store IDaSS designs!

If you want to restart VisualWorks with a specific image file, you should indicate this image file (full path name!) as parameter on the command line which starts the VisualWorks executable (for the Windows versions, this is file **visualinc.exe** or **visual.exe** in the **bin** subdirectory of the main VisualWorks Smalltalk installation directory).

If you are using a 'quick start' method for VisualWorks (for instance with a desktop icon under Windows), do not forget to add the intended image file to the command line stored there. Under Windows, use the **'properties'** selection from the right mouse button menu of the icon - and do not forget to set the working directory to the VisualWorks home directory too, with full path name (this forms part of the stand-alone version's installation process).

5.4 Main window components



This window contains a menu bar and three main sub-windows:

- The top one is the actual schematic workspace in which the system structure is defined graphically with blocks, connectors placed within blocks and data buses drawn between connectors. Blocks and buses can be given 'viewers' (a kind of virtual test probes) to provide continuous information regarding the system state.
- The bottom left one showing 'messages will come here' is a small (read-only) text window used to display information of all sorts. A number of previous messages is kept in this window, which can be scrolled back into view. The graphic button on the left margin enlarges the sub-window to the full window size in order to view more messages in the list (click once more to reduce the sub-window again).
- The bottom right part is the focal point in the timing of the simulation (which is always going on, even while editing the system!). The text on the button shows what will happen if you click your mouse there:

CLOCK STEP simulates a single clock cycle

EVENT STEP advances time to the next scheduled event

TIME STEP advances time by a specific amount

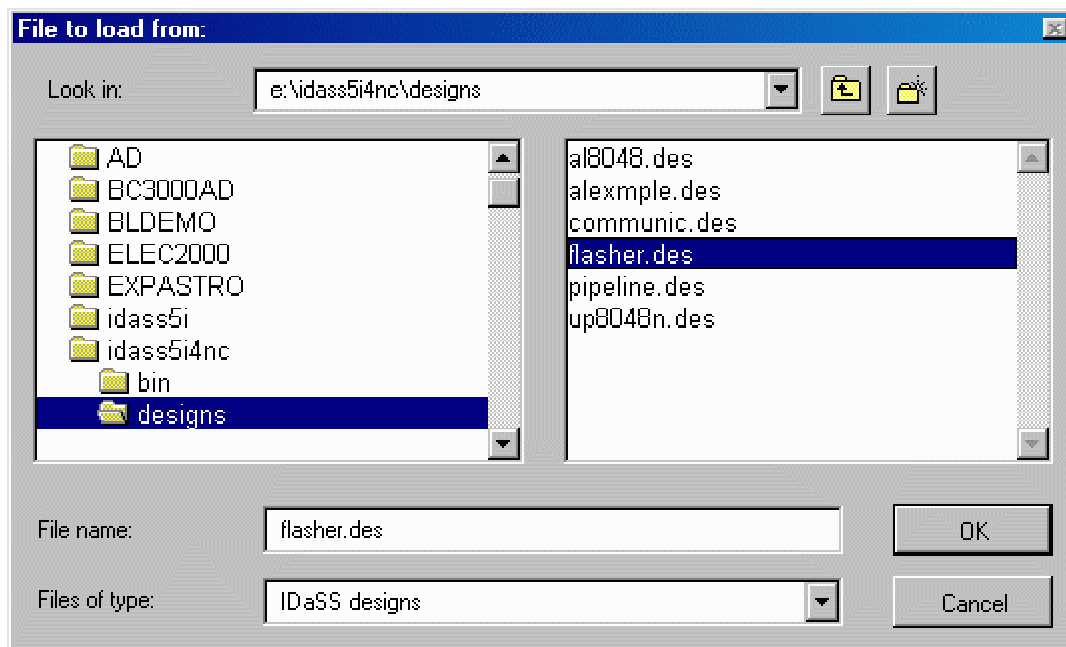
ABORT stops a long (simulation) operation

The single line text window above the button shows either the clock cycle number (with one fractional digit as shown in the picture, a '-' sign indicates *'just before the start of the clock cycle shown'*) or the

actual simulation time. Changing settings and modes controlling this, as well as resetting the simulated system is done from the **'Time/simulation'** menu on the menu bar.

5.5 Loading a system

You can load an existing IDaSS design by selecting **'load system'** from the **'File/system'** menu on the menu bar (but only if you have an empty top schematic). You are first asked whether you want to get the file from the IDaSS designs clipboard (called **'temp'**) or from a normal file. The latter will open a Windows-95 style file search window which targets an IDaSS design (**.des**) file. The file you select will be loaded and immediately starts simulation at simulation time 0.



5.6 Saving a system (part)

Saving the complete system is done by selecting the **'save system'** entry from the **'File/system'** menu on the menu bar. Likewise, all important blocks in a design have a **'file out'** entry in their menu. In both cases, you are presented with the choice to store the system or element onto a normal design file, the IDaSS designs clipboard or using an installed 'alien file format'. The latter allows you to save an IDaSS design as a synthesizable VHDL or Verilog (or other language) file. Installing alien file generation is done from the **'Miscellaneous'** menu on the menu bar.

5.7 Building a system part 1: blocks

System building starts by placing blocks on a schematic. This can be done from the '**Schematic**' menu on the menu bar, entry '**add block**'. The menu which pops up allows you to add a standard block, or one loaded from a file or the designs clipboard. The selected element must be placed on the schematic by moving the mouse to position the upper-left corner, then holding down the left mouse button to position the bottom-right corner after which the left mouse button can be released. After this, some extra information must be provided (name for new blocks, width in bits for registers, etc.). Note that you may use any name for a block you like, as long as there are no two blocks in a single schematic with the same name.

Names in IDaSS start with a letter, which may be followed by any combination of letters, digits and underscores - IDaSS is case insensitive and names can be as long as you like (provided they fit in the block symbol), with all characters checked in name comparisons.

After a block has been placed, you can start extending and modifying its behaviour - always from the menu you get by popping up the floating menu for the block. Standard operations like moving the block on the schematic (with '**frame**'), removing the block and adding 'viewers' to monitor the state of the block are directly available on this menu, while most blocks have an '**edit**' entry for more in-depth changes. Some editing operations require the use of a separate edit window, like defining the states of a state machine controller.

5.8 Building a system part 2: connectors

Only in a few rare cases, connectors are created when the block is initialised (a three-state buffer makes no sense without input and output connectors). In all other cases, you have to add the connectors yourself (mostly from the '**edit**' menu mentioned above). If the system asks for an *optional* name, you can leave the name blank. If the system asks for a *mandatory* name, you have to enter a name - as with blocks, any name will do, as long as the connectors on a block are named differently.

Each connector has its own menu, which allows operations like starting a bus from a connector or disconnecting the connector again. IDaSS uses five types of connectors:



Inputs which simply make a value on a connected bus available within the block.



Continuous outputs which place a value generated by the block on a connected bus.



Three state outputs are also outputs, but it is possible here to '**disable**' the output so that another one can drive the bus. After initialisation, a three-state output is disabled by default and needs an '**enable**' command to drive the bus.

By sending '**enable**' and '**disable**' commands to blocks, all three state outputs which respond to those commands will change their state. Individual connectors can be targeted by appending '\\\ ' and the connector name after the block name.



Control inputs whose main task is to translate values on the connected bus into commands for the block it is placed in, using a textual syntax which resembles a 'Programmable Logic Array' specification.

These are described in section 6.10 in some more detail.



Bidirectional 'feedthrough' connectors are only allowed for subschematic blocks - they form a direct connection between a bus connected at the schematic symbol block and a bus connected to a special connector block on the underlying schematic (with the same name as the connector at the symbol).

5.9 Building a system part 3: buses and commands

Data transfer between blocks on a schematic is done with actually drawn data buses. A new bus must be started from an existing (as yet unconnected) connector with '**add bus**' from its menu. Once you made this menu selection, you can draw the bus by moving your mouse away from the connector. You can make a turn by clicking the left button on the mouse at the turning point and moving the mouse again. If you want to continue in the same direction, click the button again without moving the mouse. Another click at the same location will end the bus without making a connection.

Connecting the bus to a connector is done by drawing the dynamic bus line through that connector and clicking the left mouse button. This is only possible if this connector was not connected yet and has the same number of bits as the bus you are drawing. Adding a new connection to a bus is done from the bus itself (not by connecting two buses together!). Just open the floating menu of the bus at the spot where you want to make the connection and select '**connect**', after this you can start drawing the new connection.

Note that IDaSS does not regard a value on a bus or in a storage location as a collection of bits which can all be 0, 1, 'unknown' etcetera individually. For IDaSS, a value is either known (from 0 up to the maximum value allowed for that number of bits) or a special value indicating something else - 'UNK' for UNKnown, 'TS' for Three-State or 'OVL' for OVerLoad (more than one driver on a bus).

To reduce clutter on a schematic, commands coming from a state machine controller or Algorithmic Level block and test values used by these elements are not transferred with graphic buses. Commands in IDaSS are abstract names, given to named blocks or connectors. Tests are performed by naming the tested object in a test expression inside a state description or Algorithmic Level routine.

Example design files *communic.des* and *up8048n.des* contain several state machine controllers which show how this is actually used while *al8048.des* contains an Algorithmic Level block using these capabilities.

5.10 Comments, comments and documents...

It is a good practice to provide lots of comments when designing a complex system. Within a textual specification, you can always place comments between double quotes. For blocks, connectors, buses and complete schematics, you can add comments using a special '**comment window**', which you can open from the '**File/system**' menu on the menu bar of a top level schematic editor. You can specifically ask for '**comment**' in a lot of menu's attached to design objects, but you can also simply click the left button of the mouse on a schematic element to get an explanation in the message window and comment in the comment window. Text you type in the comment window will be saved automatically when you request comment for another object.

IDaSS can generate textual documentation automatically. This documentation will provide information present in the design database, and will also include any comment attached to design elements. As the amount of documentation can be very large, it is possible to fine-tune what should be included and left out using the '**document settings**' sub-menu of the '**Miscellaneous**' menu in the menu bar. Documentation can be generated for complete systems (again, from the '**Miscellaneous**' menu), sub-systems or individual blocks - just look out for '**document**' menu entries.

5.11 Saving screen space and working time

Normally, editing textual specifications is done in separate windows attached to the objects these specifications belong to. Opening and closing these windows takes time, keeping them open all at once creates a mess on the screen. You can open a '**universal compiler**' window from the '**File/system**' menu of a top level schematic editor which can be used for all textual

specifications in the system (the elements which can use it enable a '**compile...**' entry in their menu structure).

The universal compiler window also allows a 'test compile' of the whole design or a sub-system of it. IDaSS text compilers generate warnings (instead of breaking off compilation) when they detect a construct which is syntactically correct but which uses/addresses system elements which are not built yet. A test compile can then be used to check if these kinds of errors still exist in the system you assumed to be complete.

5.12 Debugging and trace file generation

Complex systems can be a nightmare to debug. It becomes almost impossible to keep an overview of the operation during a simulation because the values you want to check are scattered all over the system structure.

Using 'probe sets' should help here - they allow a central window to show values from all over the system, using a textual specification. The syntax resembles a test expression in a state machine controller, extended with special tests even a state machine controller can not perform. Each separate probe in a probe set can also combine several test points using the normal IDaSS expression operators. Multiple probe sets can be defined and individually activated or made 'dormant'. Probe sets as well as individual probes can be given meaningful descriptive names and attached comments.

With the probe set targeted at the system, it is possible to define messages, warnings and breakpoints to be generated when specific values are detected. It is also possible to record the values to a simulation trace file during a simulation run. The format of these trace files is not fixed, so it is possible to generate a trace file which suits a specific post-processing program or which just highlights specific events in the system. One of the trace file formats delivered with IDaSS comes close to generating complete VHDL test environments for the traced system, another one generates a format suitable for loading in a Verilog simulator with the `$loadmemb` command.

For more information on probe sets, select 'IDaSS features' from the 'subjects' page in the help window (this page also contains information on other subjects discussed here).

5.13 Timing simulation and optimisation

IDaSS performs full timing simulation with a simulation timestep of 10 femto-seconds (light travels 3 micrometers in that time, so it should provide ample time resolution...). The maximum delay which can be specified in the timing simulation is one full day, but the actual simulation time has no limit (so you can simulate forever).

This timing simulation is well hidden. You will not notice it when you are using only the CLOCK STEP button, until you get 'setup time violation' errors which indicate that your design is becoming too slow (you are placing too many asynchronous logic between clocked elements). If this happens, you can perform major cheating in two ways:

- Simply turn full timing simulation (and setup time checking) off by selecting the '**fast BBB simulation**' mode in the '**Time/simulation**', '**settings**' menu.
- Make the clock period longer by changing the '**current clock interval**' in the same '**settings**' menu (better also change the '**reset clock interval**' to the same value to prevent the same errors after resetting the simulated system).

Another way to cheat is by changing the timing values calculated for the designed system. The calculations are stored in the technology file which is attached to the system (automatically when IDaSS was installed or manually from the '**Miscellaneous**' menu).

The default technology file *idass.tec* uses an abstract timing model in which an inverting logic gate has a delay of 2 nano-seconds and a non-inverting logic gate has a delay of 3 nano-seconds. All other delays are derived from these values where possible (or using calculated guesses if no gate-level structure was known). With a default clock cycle of 100 nano-seconds, this translates to a maximum of 50 logic gates in the 'critical path' of a design - which is a rather low number (comparable to the critical path lengths reported for DEC alpha processors!).

You can use rule-of-thumb multiplication factors to get from IDaSS indicated timing to the real technology timing. For instance, 1 micron CMOS ASIC processes generally achieve a timing 5 (five) times better - if you can get your system to run within the default 100 nanoseconds clock of the abstract technology, you should not be surprised to reach 50 MHz clock speed with a 1 micrometer process.

The timing simulation uses separate delays for different parts of a design, and sums these delays during actual simulation - all of these delays have separate sections in the technology file:

- *Input value and command setup times to the clock* for synchronous elements (these are all checked during simulation, violations by default abort a simulation run).
- *Clock to value and command stable delay times* for synchronous elements.
- *Clock cycle time and reset-to-first-clock delays* for the complete system. The reset-to-first-clock is separate and generally much longer than the clock cycle to give the system time to stabilise after reset. IDaSS forces a lot of UNKnown values into the system after reset to

check whether it will actually reach a stable state, most of these UNKnowns should have resolved by the time the first clock is given.

- *Fixed asynchronous transfer delay times*, for instance address-to-output on an asynchronous memory read port, test value-to-command delays on a state machine or command-to-enable delays on a three state output. A special case is the delay assumed to be present in the multiplexer which selects the outputs of different functions in the ALU-like 'operator' block.
- *Basic combinatorial operator delay times* which are combined into source-to-result delays for complete expressions (using critical path analysis) to save simulation time.

All of these delays can be changed manually (from the '**Time/simulation**', '**settings**' or element-local '**edit**', '**delays**' menus), which is a way to get out of the 'setup time violation' errors. Only use this, though, if you are certain that the actual implementation will be faster than indicated by IDaSS - this is very well possible because IDaSS assumes worst-case timing with no optimisations at all done in the implementation path.

The best way to solve timing problems is to change your design in an effort to decrease the critical path length. Because of the worst-case timing model, any improvement in IDaSS timing should never worsen the timing in the actual implementation. Aside from major structural changes (inserting extra registers, moving combinatorial logic around or combining combinatorial logic), sometimes simply re-writing combinatorial logic expressions can also help (for instance to reduce the number of bits actually added, as wide adders are slower than small adders).

The default technology file 'knows' that a compare (not) equal to a constant can be built with highly optimised logic (for instance a single NAND gate for a 'fourBitValue ~= 1111b') and has a lot of other tricks like these built-in. Beware that major re-writes of expressions may make them unreadable, so add a lot of comments to explain what you are doing.

Although IDaSS does not provide true critical path analysis, it maintains a list of setup time margins during simulation (can be placed in the message sub-window with the top entry of the '**Time/simulation**', '**settings**' menu). Each list entry indicates the exact setup time checked, the minimal margin encountered on this setup time during the simulation so far and the clock cycle in which this happened (actually, the clock cycle before the indicated number). By replaying that clock cycle using the EVENT STEP button (enabled from the '**settings**' menu), you can follow asynchronous signals as they travel through the system and figure out where you can improve your design.

5.14 Converting an IDaSS design into real hardware

Although IDaSS is a relatively abstract hardware design 'language', it was designed in such a way that hardware synthesis should be possible. There is, however, no (commercial) synthesis tool which can use an IDaSS design file as its input description. They all use (more or less) standard hardware description languages, sometimes even proprietary languages only understood by a single tool.

To bridge the gap, IDaSS is fitted with an 'alien file generation' system which directly produces a textual description from the design database maintained by IDaSS during a design/simulation session. The actual language/dialect/text layout to be generated is defined in a 'template' file, which combines a rulebase and simple expert system capable of a lot of conversion and optimisation operations. Before it is possible to save a system (or sub-system) in an alien file format, the rulebase must be attached to IDaSS. This can be done during installation (remove a comment doublequote from the *idass.ini* file) or using an entry in the '**Miscellaneous**' menu from the menu bar.

The *compass1.aft* template delivered with IDaSS is targeted towards the Compass synthesis system, which uses VHDL as input language. This template file not only generates the VHDL file needed, but also several other files needed to generate macrocell layouts for RAM's and ROM's. You need to set some switches in the template file menu before this is enabled, though (which can be found from '**Miscellaneous**', '**alien file template...**' menu entries).

The *spartan2e.aft* template delivered with the system is targeted towards the (freeware!) Xilinx WebPACK system (see www.xilinx.com) and generates a Verilog source code file with accompanying 'User Configuration File' for synthesizing IDaSS designs into Xilinx Spartan-II and Spartan-IIE FPGA's. See *spartan2e.pdf* in the IDaSS *support* directory (after installation).

Some hints and tips:

- Alien file generation will stop with an error if an Algorithmic Level block is present in the system (part) to convert.
- Alien file generation also aborts when the converted system part still contains textual specifications which generate a warning during compilation.
- All memories (and their ports) must be given a technology restriction as defined in the technology file. Note that the 'dual port RAM' cannot be converted by the template files given above. See the description of memories in chapter 6 for more information.

- Although three-state outputs can be used and will be synthesized correctly, try to avoid them *inside* the system and use multiplexers instead. Three-state buffers are slow and large, their use should be limited to interconnect modular systems on a chip and for off-chip bus connections.

5.15 ‘Properties’

‘Properties’ allow additional information to be attached to IDaSS design elements. They are stored in the design database and design files, will also be shown in automatically generated documentation. At the moment, they do not influence the actual simulated design behaviour in any way – they can, however, be accessed by the ‘alien file generation’ rulebases described in the previous section. These can use this extra information, for instance, to control ASIC pin assignments, output pin power drive levels or engraving of copyright information on the chip surface. If a template file uses properties, this will be clearly indicated in the documentation coming with that file.

Multiple properties may be attached to a single design element, distinguished by a primary key name and an optional secondary key which can be another name or a numeric value. The actual property value is either a numeric value (in the range $0..(2^{64} - 1)$, can also be given as UNKnown) or a sequence of characters (also known as a ‘string’).

To enter a ‘string’ value, type a single quote followed by the characters you actually want to store. The starting quote is not stored – do *not* put a single quote at the end unless you really want one there.

To check, enter, edit and remove properties, just click on the ‘**properties...**’ entry in the main menu of the design element whose property set you want to access. This will pop up a recurring menu which shows the currently attached properties (with their values) and allows new properties to be defined. Clicking on an existing property allows removal, renaming and changing the value of that property. Properties which have a secondary key are indicated as such and pop up a new recurring menu to access the set of secondary keys and attached values.

6. BLOCKS WHICH CAN BE USED IN A DESIGN

The number of 'Basic Building Blocks' available to design a system is fairly limited. They can be called from the **'add block'** menu entry of the **'Schematic'** menu in the menu bar, with all types of memories in a separate sub-menu. We only give a very short introduction here. For more information, open a help window, click 'subjects' and then 'IDaSS entities' for a list of links to complete descriptions of all the blocks (and other design elements).

Data path elements like registers, memories, connectors and buses all have a variable 'width' (number of bits they store or transfer in parallel). By default, this width may vary between 1 and 64 bits, but the upper limit can be changed from the **'Miscellaneous'**, **'environment settings'** menu. Calculations done *within* a block can always handle up to twice the number of bits of the maximum data path element width (so, by default up to 128 bits).

NOTE: at the moment, viewers and memory editor windows will not show values wider than 64 bits!

6.1 Registers

These have a designer-selectable width, and are equipped with a semaphore bit which indicates they have been written. An asynchronous system reset value can be defined (by default, this is UNKnown, so the register is not initialised). One input and/or one (three-state) output connector can be placed in the register for loading new values respectively outputting its current value.

Each clock, a register performs a default operation unless instructed otherwise by sending it a command. After initialisation, this default operation is **'hold'** which simply indicates the register will hold its value. Other functions a register can perform on given commands are:

'load' a value from the input connector on the next clock (also sets the semaphore bit to 1/**'W'**ritten then).

'inc' increment the current value on the next clock.

'dec' decrement the current value on the next clock.

'ldinc'

load incremented value from the input connector (and set semaphore) on the next clock.

'lddec'

load decremented value from the input connector (and set semaphore) on the next clock.

'write: <value>'

load the given value into the register and set the semaphore on the next clock.

'setto: <value>'

load the given value into the register on the next clock but do *NOT* set the semaphore.

'reset'

load the register with a specific value (default 0) on the next clock, overruling all commands above - does not influence the semaphore.

'ressem'

resets the semaphore bit to 0/'R'eset on the next clock, is overruled by a command which sets the semaphore.

State machine controllers and Algorithmic Level blocks can test the current value of the register by simply using its name in a test expression. The semaphore bit is tested by using the register name followed by a question mark. Using two question marks in a row sends a '**ressem**' command to the register when the test is performed (a kind of 'test-and-reset').

6.2 Operators

These blocks contain most of the combinatorial operations in a datapath, ranging from a simple bus splitter or -combiner to a floating point operation (and beyond). They can be given any number of input and (three-state) output connectors, with a designer-selectable number of bits on each of these.

The functionality resembles that of a normal Arithmetic Logic Unit in a processor - an operator can perform several functions (like an ALU can perform addition, subtraction, shift left...), but only one at a time. Functions are defined by a set of expressions entered (and then '**save**'d) in a separate text edit window.

Each function has a designer-defined name, which is also the command to be given by state machine controllers and Algorithmic Level blocks to let the operator perform that function. A default function can be defined to be executed when no command is received. For simplicity, the first function entered in an operator becomes this default function.

The expressions which define a function operate with the names of input connectors as variables and assign to the names of output connectors, using the standard Smalltalk syntax and a set of over 60 (hardware implementable)

operators. These include standard arithmetic and logic operations, but also bit field extraction and concatenation, comparison, priority, multiplex, merge, constant generation etcetera. For a complete list (and an explanation of the Smalltalk expression rules) select 'IDaSS operators' from the 'subjects' list in the help window.

Expressions in an operator function can also assign to a 'temporary' variable (a name preceded by an underscore). Following this assignment, this temporary variable can be used as source value in other expressions. An example of this is the following:

```
"Add two 8 bit inputs 'a' and 'b' with a
single bit carry in 'ci', producing an 8
bit output 'y' and single bit carry out
'co' (comments are placed between double
quotes, as is done here):"
```

```
_sum :=
  a add: b cin: c.
```

```
"The 'add:cin:' operator performs an 'add-
with-carry' operation, resulting in a
value one bit wider than the values which
were added. This extra bit is the carry
out. The period signals the end of an
expression."
```

```
y :=
  _sum from: 0 to: 7.
```

```
"The 'from:to:' operator extracts a bit
field from a value. All operators have
a well-defined number of bits they will
generate - there is no possibility to
create a value with a run-time variable
number of bits in it."
```

```
co :=
  _sum at: 8
```

```
"The 'at:' operator extracts a single
bit from a value. The last expression
does not need a period at its end."
```

Note: the exact language syntax of all the languages used by IDaSS is placed in the help file and can be accessed from the 'subjects' list entry 'IDaSS languages'.

6.3 State controllers

These blocks form the main centralised control elements in a register level design (decentralised control is implemented with 'control input' connectors, which will be discussed later). Each state machine controller can test and control all other blocks on the schematic it is placed in itself, as well as all sub-schematics of this schematic (using 'path' names like 'subschema\blockname').

There is no limit to the amount of state machine controllers in an IDaSS design, they are even allowed to jointly control data path elements as long as they do not send contradicting commands at the same time (for instance one controller telling a register to 'load' while another tells that same register to 'inc'rement).

The functionality of a state machine controller is entered in a textual form (in a separate text edit/compile window), with a separate text for each of the states of the controller. Although IDaSS numbers the states and the default state transition is to go to the next numbered state (with the first state following the last one), names must be used to label the states when a non-default state transition must be specified. An unlabeled state must start with a colon (':'), while a labeled state starts with the state name immediately followed by a colon.

Commands are given to blocks by placing an abstract command name after the (path) name of the block. Commands with numerical parameters (like 'write:' for a register) must be followed by a numerical constant, commands with a name parameter (state machine controllers themselves know a 'goto: <statename>' command) must be followed by an allowed name for that command. Lists of commands are separated by semicolons.

State transitions are defined in a 'graphical' way and use state names where needed. A state machine controller can be fitted with a subroutine stack which allows call and return commands to be used. After initialisation, the 'stack depth' is set to zero indicating 'no stack'. Within a state text, a state transition is always the last command which will be executed. The following state transition commands are available:

- '-> targetStateName'
go to the indicated target state at the next clock.
- '<<' hold the current state at the next clock.
- '>>' go to the next numbered state at the next clock (first state if this is the last one).
- '=> targetStateName'
call the target state at the next clock, pushing the next numbered state on the subroutine stack at the same time.

'=> targetStateName , returnStateName'
call the target state at the next clock, pushing the indicated return state on the subroutine stack at the same time.

'<=' return to the state at the top of the subroutine return stack at the next clock and pop this state from the stack at the same time.

'<= targetStateName'
pop the subroutine stack at the next clock but throw away the state which was at the top of the stack - use the given target state instead as the next state.

Condition tests and chains of conditional commands (possibly ending with a state transition) are placed in test blocks. These test blocks can be nested and syntactically take the place of a single normal command sent to a block. A test block is in principle a CASE-like construct, but has the peculiarity that (unless a state transition is executed) *all* tests are performed in the order of the state text. Several branches of the CASE can match the test value - all these branches will then be executed.

A test block starts with a square opening bracket ('['), followed by a test expression. This expression uses the same base syntax and operators as expressions in an operator function. The operands are (path) names of tested blocks and no assignment is done. A test expression is not allowed to generate a constant value.

Special tests (like a register semaphore value) can be performed by placing question marks after the block name. To test a bus value, target a connector attached to that bus by following the block name with '\\\' and the connector name (placing a question mark after an output connector name will test the value generated internally by that block). Like operator functions, it is possible to use temporary value assignment expressions before the actual test expression - these temporary value expressions are local to the test block (must follow the square opening bracket).

Following the test expression, the test block contains one or more CASE branches starting with a vertical bar ('|'), a specification of the values which activate this branch and the list of commands to be executed in the branch. The value specification may be empty, indicating that any non-zero value activates the branch. Test values may be specified as normal constants ('12', '%1001', '0A5h'), values with don't care bits ('01x0b', '4xh') and/or ranges of values ('13..21'). Multiple test values may be given in a single branch, separated by comma's. A branch is executed if the test expression result matches at least one of its test values or -ranges.

A test block ends with a closing square bracket (']'), followed by a semicolon if more commands follow after it.

A good example of a modestly complex state machine controller with nested test blocks can be found in the *up8048n.des* file coming with the IDaSS

distribution. The *communic.des* file shows how state machines can cooperate and control elements in sub-schematics.

Features worth mentioning here are that IDaSS state machine controllers have a set of commands of their own (and therefore can control each other, see the help page of the state machine controllers). Also, they can use a set of single bit 'signals' (one of the 'IDaSS entities' of the 'subjects' list) for communication and synchronisation purposes. These are defined at the top level of a design and can be used by all the controllers present within the design.

6.4 Memories

IDaSS provides five fundamentally different types of memories and four types of 'ports' which can be used within these memories for reading and/or writing. When a technology definition file is attached to IDaSS, this file contains restriction rules for implementable memories (maximum sizes, maximum number of ports, port timing restrictions etc.). It is possible to select 'IDaSS default' technology for a memory, which removes all implementation restrictions (but also removes the capability to actually implement the memory via the alien file generation facility).

The following port types are available:

'read-only' ports have an address input and a (three-state) data output. Asynchronous as well as synchronous (and even pipelined) reading is possible, the output can be latched. For synchronous reading, a **'read'** command must be sent to the memory (activates all read ports which can handle it) or to the address input of the port (activates this port only). Reading can also be done by default, in which case a **'noread'** command must be issued in the same ways to *stop* reading.

'write-only' ports have an address and a data input. They always write synchronously, but the number of clocks (and pipeline overlap) for a write cycle may vary. To control writing, **'write'** or **'nowrite'** commands must be used just like the commands for a read-only port.

'read/write' ports have an address and a data input, as well as a (three-state) data output. They combine the capabilities of a read-only and a write-only port, adding automatic reading if no writing is done and restrictions on the timing between read and write cycles.

'fixed' ports only have a (three-state) data output, which reads a fixed location in the memory. The technology file may restrict the address to be read to a single address or range of addresses.

The actual memories modelled by IDaSS make use of these ports, but can also add specialised inputs and/or outputs:

RAM's can be read and written, the unrestricted technology ones can be given any combination of all the port types available. The default technology file defines three implementable RAM memories:

'Single ported' with one read/write port which reads and writes synchronously with automatic output enable, maximum 64 kilobits in several width/#words combinations.

'Dual ported' as single ported but now with two read/write ports (not always present in actual technology libraries and currently not supported by the standard VHDL/Verilog generator rulebases).

'Register file' with a maximum of 6 ports, of which 4 may be single clock synchronous write-only ports while the remainder should be asynchronous read-only ports, maximum size is 1 kilobit in several width/#word combinations.

ROM's can only be read, the unrestricted IDaSS technology allows any number and combination of read-only and fixed read ports. The technology file defines two implementable ROM's:

'Synchronous' with a single cycle synchronous read-only port, maximum size is 128 kilobits in several width/#words combinations.

'Asynchronous' uses one asynchronous read-only port, also with a maximum size of 128 kilobits.

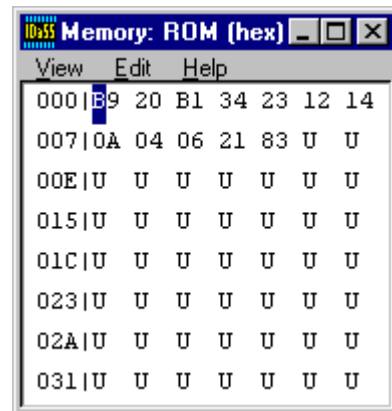
FIFO's allow read-only and fixed read ports and provide a special write data input. The **'read'** command can be used to remove the word at the 'head' of the memory (address 0), which shifts all other words down by one address. The **'write'** command writes the value input at the data input to the 'tail' of the memory. The word at the head and the number of words currently in the FIFO can be tested by a state machine controller or Algorithmic Level block. The technology file defines a single (register file based) FIFO with a fixed read port for address 0 and a maximum size of 1 kilobits. Note that if you create a FIFO with a synchronous read-only port, you can only send **'read'** and **'noread'** commands to this port via its address input (the FIFO itself uses these commands for other purposes). Reading and writing a FIFO can be done by default which requires **'noread'** and **'nowrite'** to prevent them.

LIFO's are like FIFO's but work as a stack instead of the queue behaviour of FIFO's. Writing as well as reading are done from the head of the memory (address 0). A LIFO can be given a large number of commands to **'push'**, **'pop'** and **'replace'** data to/from/at the head of the memory (see the help file). The technology file defines a single

(register file based) LIFO with up to two fixed read ports (addresses 0 and 1) and a maximum size of 1 kilobits.

CAM's (Content Addressable Memories) are a very special type of memory which allows matching stored words against a bit pattern (with don't care bits), after which all words in the CAM can be modified by setting and/or resetting bits based upon the match results - and all this in a single clock cycle. This type of memory allows read-only and fixed ports to be added, although the technology file does not allow these to be present for the implementable CAM (with a maximum size 1 kilobits). As a very complex memory, a CAM defines a large number of special inputs and outputs and has a large number of commands and options - see the help file for more information.

Memory contents can be viewed and modified in special windows, in a variety of display formats. Due to simulation time constraints, these windows can only show a small part of a memory and cannot perform normal scrolling operations - scrolling is done with the cursor keys (together with **[page up]** and **[page down]** where available). Within the window, the mouse can be used to move the cursor.



The '**Edit**' menu allows placing the cursor at a specific address, filling and moving blocks of memory. Typing valid (hexadecimal) digits will change the memory contents. To insert a new word in a FIFO or LIFO, type **[i]**. To remove a word, type **[r]**. To give a word an UNKnown value (indicated with 'U' in the window, type **[u]**.

6.5 Three-state buffers

These very simple blocks must have a single data input and a single three-state data output, which can be controlled with '**enable**' (for a default disabled output) and '**disable**' (for a default enabled output) commands. The width is (as usual) designer-selectable.

The same '**enable**' and '**disable**' commands can be sent to any block with three-state outputs, and will then control all outputs with a matching default state. By directly targetting the output (add '\\\' and the connector name after the block path name), each three state output can be controlled individually.

6.6 Constant generators

These blocks must have a single (three-state) output and know only a single '**setto: <value>**' command which places the given value onto the output (with automatic enabling if the output is a three-state connector). A default value can be specified to be output when no command is received. The standard use of these blocks is to let a state machine controller or Algorithmic Level block force a value onto a bus. Another use is to place a control input (see section 6.10) in a constant generator, forming a very powerful Programmable Logic Array (PLA) type module.

6.7 Algorithmic Level ('AL') blocks

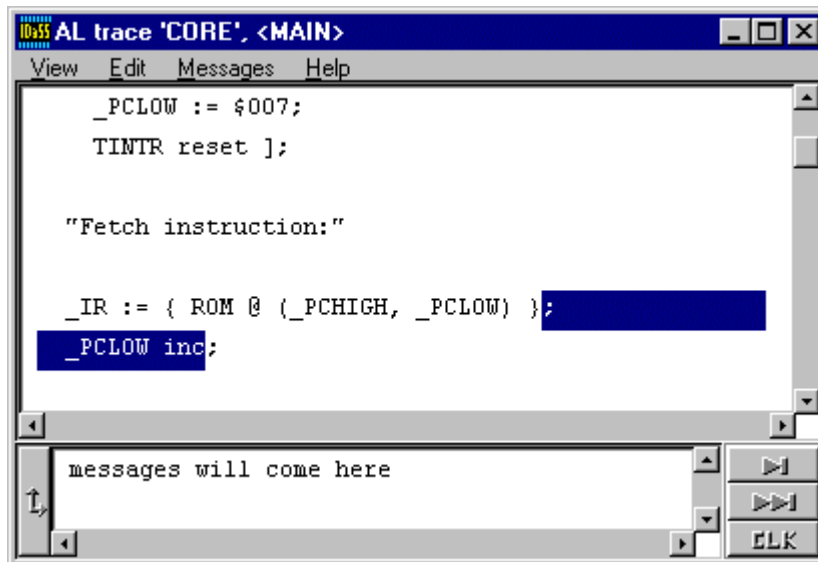
These blocks combine data processing, data storage and control in one. They contain local variables in the form of registers and memories, with functionality defined by a set of routines written in a language supporting all the standard 'structured' programming language constructs.

An AL block can test and control elements in its (register level) environment just like a state machine controller - it can even write directly into external RAM's and registers. An interrupt-like mechanism is available by allowing external '**call:**' commands which start one of the internal (so-called 'global') subroutines - this command is also available on state machine controllers which are fitted with a stack. Synchronising to the clock is done by 'wait statements' which come in a number of forms ranging from very simple ('*wait here for the next clock*') to extremely complex ('*stop executing here - after two clocks, start comparing the values of two external registers, repeat this test each clock until they are equal or a maximum of five clocks have passed, return the last test result when resuming*'). The main drawback of AL blocks is that they cannot be converted (yet) with the 'alien file generation' facility into standard hardware description languages.

The edit window for AL blocks is very powerful. It has to be, as it is functioning as a complete software development environment (with as special feature that editing and execution of a program are done at the same time). The window can switch between four main modes - multiple windows may be opened on a single AL block to show different aspects at the same time:

- 1) *local variable editor*: allows definition, monitoring and modifying of local register- and RAM-like variables. Normal memory editor windows are used for basic RAM variable contents editing. Comment lines may be inserted for better overview.
- 2) *routine editor*: allows editing and compiling of main, local and global (sub-) routines.

- 3) *execution stack editor*: allows inspection and modification of the execution stack of the AL block, shows exactly how the program flow is progressing (for instance loop count values).
- 4) *program tracer*: shows routine and statement within this routine which is in execution. Single stepping is possible using three buttons at the bottom-right part of the window (top-to-bottom: 'execute next statement', 'execute until next wait statement' and 'perform clock')



Two example files are distributed with IDaSS to illustrate the use of AL blocks (for more information, consult the help file):

alexmple.des explains and demonstrates all the constructs possible in an AL block. Load it, then open a comment window from the '**File/system**' menu and ask for '**comment**' from that same menu - then follow the instructions given.

al8048.des gives an actual design where a single AL block is used to implement a complete processor core.

6.8 Sub-schematics

These provide hierarchy and re-use within a complex design. A sub-schematic can be placed as symbol in another schematic, with multiple schematic symbols in a design using the same sub-schematic (with different simulation states, of course). When creating a sub-schematic and re-use is possible, IDaSS will automatically ask whether you want to re-use the symbol, contents or symbol AND contents of an existing schematic (or simply start with a new one).

Data communication through a schematic boundary is done with bidirectional connectors at the symbol and within 'connector' blocks placed onto the sub-

schematic (these are interconnected by having the same name). Testing and controlling elements of a sub-schematic is possible by using 'path'-names in a state machine controller or Algorithmic Level block. Connector blocks may be tested to obtain the value on the buses they are connecting.

IDaSS provides named parameters to give re-used design elements different behaviour even though they have the same specification. Parameters are attached to schematic contents and schematic symbols (with the latter overruling the former if they have the same name). Most parameters are numeric (although special purpose 'time' and 'string' parameters are also available) and they can be used as 64 bit value inside any expression by preceding the parameter name with a double underscore. Numeric parameters can also be used to choose default commands or define special values which would normally be constants (for instance the value loaded in a register after system reset). See the help file ('subjects', then 'IDaSS features') for more information.

6.9 Multiple schematics

These behave as a set of numbered ('tagged') identical sub-schematics with all data communication connections connected in parallel. Each tag is a 64 bit value, the user can specify the set of tag values to be used (with each tag value creating a complete sub-system!). Tag values within a single multiple schematic must be different.

Testing and controlling elements on the sub-schematics of a multiple schematic requires the specification of the tag number in the 'path'-name of that element, for instance as in 'MULTIPLE[34]\ACCUREG'. This means that test and control channels are *not* connected in parallel at a multiple schematic symbol.

System elements within a multiple schematic can obtain the tag number of their sub-schematic via a parameter called 'tag'. To obtain the local tag number of higher level multiple schematics (nested multiple schematics are allowed!), use 'tag1' for the next higher level, 'tag2' for the level above that, etcetera. For example, suppose that a state machine controller placed on a schematic inside a multiple schematic wants to test whether the 16 bits 'addr' connector block matches the lower 16 bits of the tag value of this schematic:

```
"Enable local OUT register's three-state
output when the 'addr' connector block
matches bits 0..15 of the local tag:"
```

```
[ (__tag from: 0 to: 15) = addr
| OUT enable ];
```

```
"Note the double underscore to access the tag as a parameter. We must use 'from:to:' to select the correct bits from the tag because the '=' operator can only compare equal width values (and returns a single bit binary 1 if the comparison holds, which is enough to select the CASE branch)."
```

6.10 Control inputs (not a block, but important)

As mentioned before, local control can be implemented with 'control inputs' (see help page 'subjects', 'IDaSS entities', 'connectors') placed within the block to be controlled. The textual specification of such a control input looks like a simplified state machine controller test block:

```
"Local control of a register (presumably the program counter of a processor). The control input is 8 bits wide:"
```

```
(7, 3, 0..1)
```

```
"Selects which bits of the input to test (this is optional, all bits are tested if not specified). In this case, bits 7,3,1 and 0 form the test value (in that order):"
```

```
"7310 <- bits tested"  
%1xxx reset; ressem.
```

```
"If bit 7 is a %1, the remainder is a don't care and we reset the register value AND semaphore (a period separates the tests here - no path name is needed because the control is local anyway)."
```

```
%01xx setto: (10 zeroes, 4..6, 2 ones).
```

```
"If bit 7 is %0 and bit 3 is %1, we load the register with %0000000000xyz11, where 'xyz' are bits 6, 5 and 4 of the input value. A state machine controller can only use constant values or parameter names for numerical commands like these. A control input has even more capabilities than shown here, see syntax in the help file."
```



```
%0000,  
%0011 load.
```

```
"Test values are specified as in a test  
block of a state machine controller..."
```

```
%0001 inc
```

```
"The last command needs no period..."
```

Multiple control inputs may be present in a single block. These may control the block as long as they give no contradicting commands. Combinations of control inputs, state machine controllers and Algorithmic Level blocks may also be used with the same restriction.

7. DO-IT-YOURSELF DEMO

The best way of learning to use a tool is to start it up and try it out... But with a tool like IDaSS, a 'guided tour' may even be better. After you've done this, you can either look in the *designs* directory for more complex examples of designs or simply start creating your own systems.

If you follow the steps below, you will create a simple design (we call it the 'running light demo') which contains a register, some combinatorial logic in an operator and a state machine controller. Don't worry if you make a mistake - IDaSS is very forgiving and will not let you destroy your work with a single mouse click. If you get lost in a menu structure, just click outside the menu or click 'cancel' in an entry field window to start over again.

We give the steps in a rather terse format here, assuming you have read the sections called 'Building a system, part X:...' earlier in this text. Here we go:

- 1) Open an empty top level IDaSS schematic editor window as indicated in section 5.1.
- 2) From the **'Schematic'** menu, add a register block, name it 'REG', make it 8 bits wide and place it in the schematic. Don't make it too small, but also leave ample room for other blocks (use **'frame'** from the register's floating menu to change its size and position on the schematic, if needed).
- 3) Attach a binary value viewer to the register from its floating menu.
- 4) Change the value of the register to 12 by using the **'edit'/'current value'** selections from the register menu.
- 5) Let's play a little: click the CLOCK STEP button several times and see the clock cycle indicator change (the '-' in this window indicates the simulation stopped just before the indicated clock) - the register holds its value! Now edit the default command of the register to 'increment' and perform some more clock steps...
- 6) Use the **'edit'** menu of the register to add an input and a continuous output connector. You may call them 'in' and 'out', but that is not really necessary (name may be left blank). Click your mouse's select button on the register and its connectors and see what happens in the message sub-window.
- 7) Add a new operator block named 'SHIFTER' to the schematic, just like you added the register. Place this new block either to the side or top/bottom of the register block.

- 8) From the **'edit'** menu of the operator, add an 8 bits wide input connector named **'in'**. Place this connector within the block symbol at such a position that it will be easy to connect to the register's output.
- 9) In the same way, add an 8 bits wide continuous output named **'out'** to the operator, to be attached to the register's input.
- 10) Connect the input of the register with the output of the operator with a bus. Start by selecting **'add bus'** from the register's input connector menu, choose a nice name for the new bus and draw the bus by moving the mouse, clicking the mouse select button to make a turn, ending the bus by placing the last segment of the bus through the operator input and clicking the mouse button again.

If you cannot draw the bus in the direction you want, click the left mouse button while pointing at the starting point of the movement. In case you want to end a bus in 'mid air' (without connecting), click a second time.

- 11) Add another bus between the two remaining connectors and attach a binary value viewer to the bus at the register input (each bus has its own floating menu!). This viewer shows **'UNK'** for an unknown value because we did not define the actual functionality of the **'SHIFTER'** operator yet - let's do that now:
- 12) From the operator's menu, select **'edit'**, followed by **'functions...'**. Select the **'???'** which pops up (to indicate we want to create a new function) and use the name **'Left'** in the entry field which opens. After this, you can open the function editor window. To keep an eye on the schematic, try not to overlap both windows.
- 13) In the text editor top-part of the operator function editor, type the following expression:

```
out := in shl: 1
```

In a comment (between double quotes), you can indicate this assigns the input value, shifted left over one bit, to the output.

- 14) Select **'save'** from the **'Edit'** menu on the menu bar of the function editor window. This compiles the function and immediately installs it in the system - the viewer at the bus should now indicate the shifted value (try changing the register value!).

If the **'save'** operation gives an error, check whether the names of the connectors on the operator match those in the expression and whether you did not make a typo, then try again...

- 15) In the register, edit the default command to **'load'**. Play a little with the register value and the **CLOCK STEP** button - you have created a shift register!

- 16) In the function editor window, select '**manage functions**' from the '**Edit**' menu, followed by '**copy**' and name the copied function 'Right'. The label of the window should reflect the change. In the expression, change 'shl:' into 'shr:'. Modify your comment to state that this is a shift to the right over one bit and '**save**' the function text.
- 17) Clicking on the operator symbol in the schematic reveals the operator now has *two* functions - but it is still performing the shift left. Attach a viewer to the operator to confirm this. Play with the register value and '**change function**' from the operator's '**edit**' menu until you are certain both functions work as they should. Note that, when you click the CLOCK STEP button, the operator function automatically falls back to the default (which should still be 'Left').
- 18) Time to start controlling the machine... From the '**Schematic**' menu, add a '**state controller**' block named 'CONTROL' to the schematic. Attach a viewer to this block. The top line (with '@') shows the current state number, the middle line (with '>') shows the next state number, the bottom line indicates the machine is actually active (running) and will show other information later on.
- 19) Open a state editor window for the state controller block with '**edit states**' from its '**edit**' menu entry. The text editor in this window shows the initial text of state number 1, which contains a single colon to indicate the state has no label name yet.

We want to create a system in which a register value shifts to the left until bit 7 (the leftmost one) becomes a binary 1, after which the register should shift to the right until bit 0 becomes a binary 1, at which point the shift direction reverses again. This requires two states in the state controller, one which controls shifting to the left and one which controls the shifting to the right.

- 20) Edit the state text for state #1, until you have the following (make sure you named the register 'REG' and the operator 'SHIFTER', or you will have to change the names in this text...):

```

goLeft: "State label..."

[ REG at: 7      "Test bit 7 of register"
| 0             "IF it is a zero:"
  SHIFTER Left; "  Shift to the left"
  <<           "  and hold this state"
| 1             "IF it is a one:"
  SHIFTER Right; "  Shift to the right"
  -> goRight   "  and change state..."
]

```

If you '**save**' this text from the '**Edit**' menu, the message sub-window will indicate '1 WARNING GIVEN'. If you scroll the message text one line, you can see what went wrong - state 'goRight' has not been

defined yet! Still, the text compiles and is taken into execution (watch the state controller block viewer, it now works with the name and shows at the bottom left which kind of state transition is made). Playing with the register value should show that the decision taken by the state controller is indeed based upon bit 7 of the register. If the controller viewer indicates '> ???' on the second line, this means that a transition command is in execution which uses a not defined state - this is allowed as long as you do not hit the CLOCK STEP button...

- 21) In the **'Edit'** menu of the state editor window, select **'ins./del. states'**, followed by **'insert after'**. Notice in the window label that you have now started editing state number 2. Change the text until it becomes a mirror image of the first state (all 'left's changed into 'right's and vice versa, testing bit 0 of the register instead of bit 7):

```

goRight: "State label..."

[ REG at: 0      "Test bit 0 of register"
  | 0            "IF it is a zero:"
  SHIFTER Right; "  Shift to the right"
  <<            "  and hold this state"
  | 1            "IF it is a one:"
  SHIFTER Left;  "  Shift to the left"
  -> goLeft     "  and change state..."
]

```

Saving this state text should produce no warnings or errors because state 'goLeft' has already been defined. A '> ???' in the state controller viewer should now disappear.

- 22) The system is almost complete. Change the value of the register to 1 and click the CLOCK STEP button several times. The value in the register should shift left and right as intended. By selecting **'show current state'** from the **'Edit'** menu of the state editor window, you have an extra indication of the progression of the states.
- 23) Let's see if the system can start up... Select **'reset all'** from the **'Time/simulation'** menu of the schematic window and confirm that you actually want to to this.

Oops! If you followed the instructions so far, the register does not get a defined value after system reset - this makes it impossible for the controller to decide what should be the next state, with as side effect the sending of unknown commands to the operator.

From the register's **'edit'** menu, change the **'system reset value'** to a **'specific value'** and choose 1 as this value. The message sub-window should confirm your choice. Now you can reset the system safely with **'reset all'** as before. It looks like this does not help much...

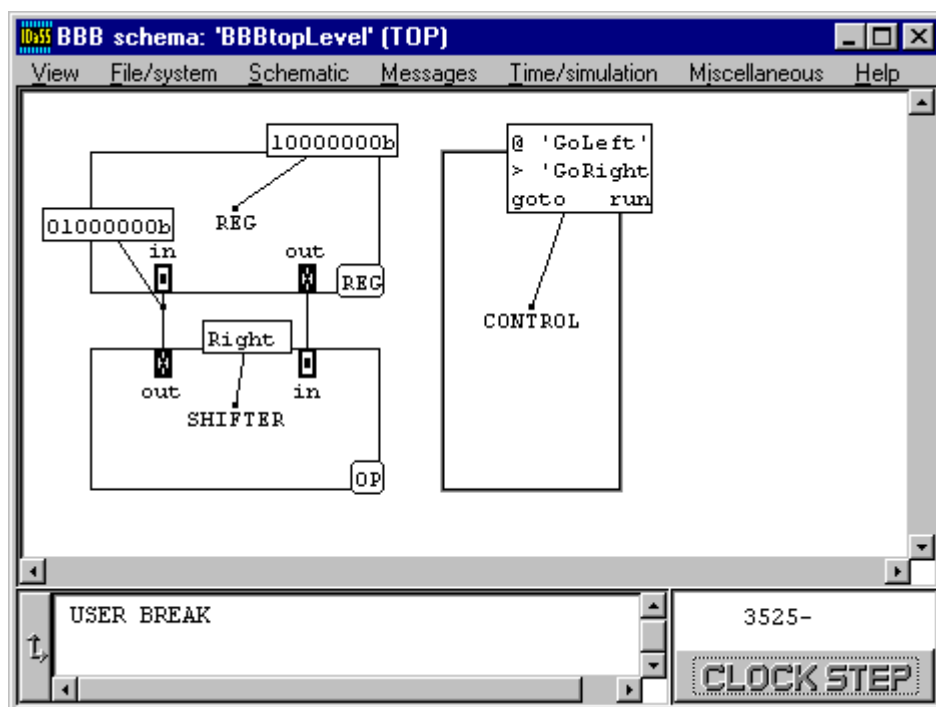
If you look at the clock cycle indicator, it shows '0.0' - meaning the system has just been released from the simulated reset signal at simulated timepoint 0. The simulated logic inside the state controller block did not have time yet to perform the actual test and set the internal state transition or external commands!

Clicking CLOCK STEP once advances the simulated time to just before the first actual clock step (indicated by '1-'), by which time the system should have stabilised and all error indications should have gone. Clicking some more indicates all is well and the system is fully operational.

- 23) Select '**run until break**' from the '**Time/simulation**' menu. You will see the clock cycle counter increment, but in a default installed system, the windows will not be updated!

You can fix this by forcing window updates at fixed simulated time intervals. First, stop the running simulation by clicking the red ABORT button, then select '**settings...**' from the '**Time/simulation**' menu, followed by '**screen update interval**'. Set this interval to 100 nanoseconds by typing '100 ns' in the entry field (as the default clock cycle time is 100 nanoseconds, this will update the screen each clock cycle). During simulation, the screen should now be updated - but you may not be able to see all the information flashing by. You can slow simulation down by inserting a '**demo delay**' with '**Miscellaneous**' menu entry '**environment settings...**'.

By now, the window should look like shown below. Congratulations if you ended up here!



This concludes the 'guided tour'. You can save your design with '**save system**' from the '**File/system**' menu. Stopping an IDaSS design session is done by simply closing the top level schematic window.

Have fun!