Tutorial

IMPLEMENTATION OF A CORDIC ALGORITHM USING THE ACTEL CORTEX M1 DEV-KIT

Acknowledgement

This Tutorial as been created in co-operation between Doulos and the Fachhochschule Oldenburg/Ostfriesland/Wilhelmshaven, the Laboratory for Computer Architectures and Programming of peripheral sub-systems: Prof. Dr.-Ing. Gerd von Cölln, Hennig, Maxim, Janssen, Stephan.

Table of contents

INTRODUCTION

The aim of this practical project in computer structures is to demonstrate how to implement microprocessors in FPGAs and how to combine those with user-defined hardware functions (developed in VHDL). The implementation discussed in this paper was based on an Actel Cortex M1 Enabled ProASIC3 Development Kit. The Cortex M1 is a microprocessor, developed by ARM and Actel specially for use in FPGAs.

This Development Kit includes a PCB, populated with a ProASIC 3 M1A3P1000 FPGA, 16 MBytes of flash memory, 1MByte of SRAM, a USB JTAG programmer and some additional I/O peripherals. A comrehensive software suite with tools for the hardware implementation, the simulation and the programming are also part of the Development Kit.

1 SOFTWARE

The software included in the Development Kit (on three CDs) represented an older version of the tools. We recommend to download the latest version of the software, available at www.actel.com. The following versions have been used in this project:

- Libero v.8.1
- Core Console v.1.4
- Actel SoftConsole v.2.0.0.13

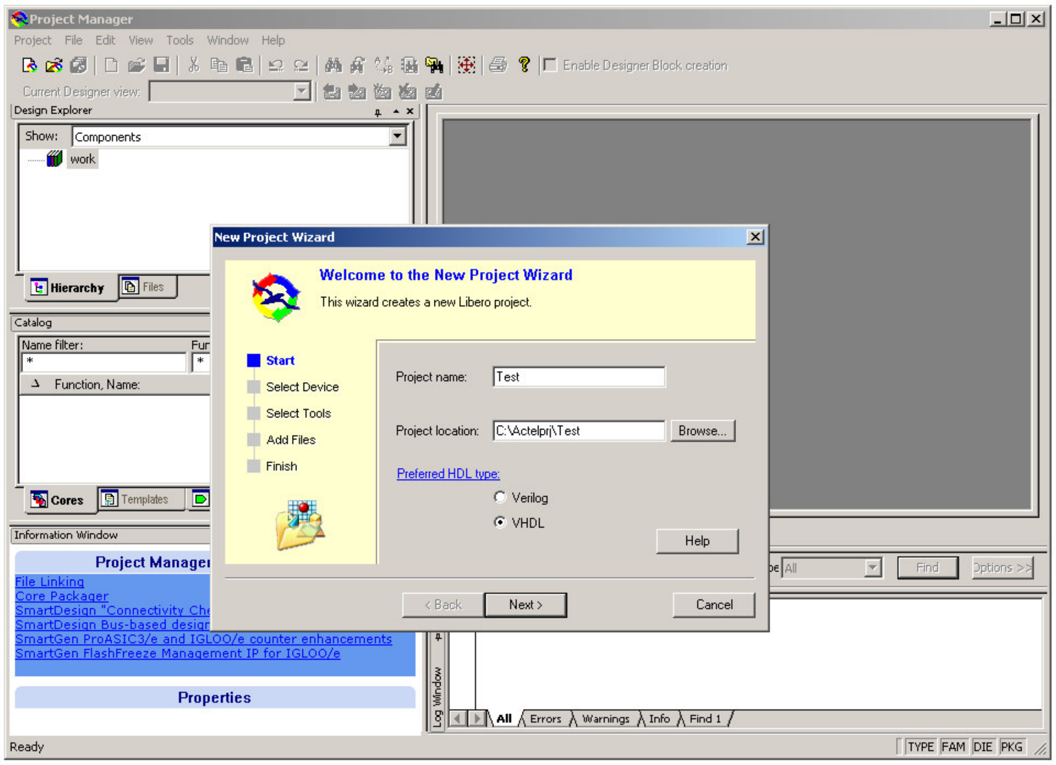
2 IMPLEMENTATION METHOD

The purpose of this project was to implement a Cordic algorithm for the calculation of sine and cosine functions on the given hardware. One part of this algorithm would be performed by external peripherals.

The following sections will present the detailed method by which this was realised.

2.1 Creation of a new project

The Libero IDE was used for the definition and implementation of the hardware. A new project was created using the `Project/New Project...` menu option. In the dialogue box that opened up as a result, the project's name and location were specified, together with the HDL used, before clicking on `Next`.



Welcome to the New Project Wizard

This wizard creates a new Libero project.

Project name: Test

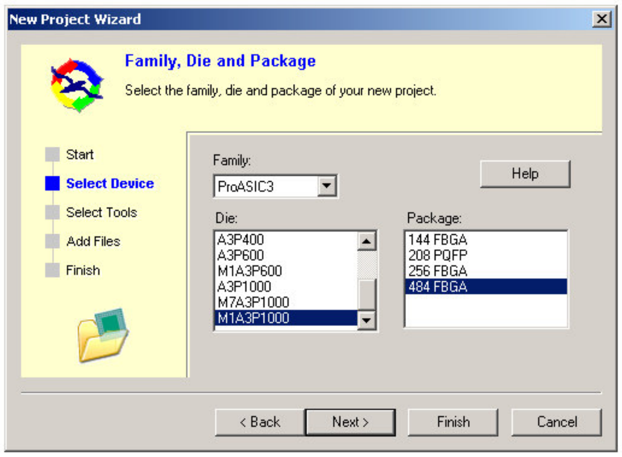
Project location: C:\Actelprj\Test

Preferred HDL type:

Verilog [✓ VHDL]

Fig. 1: Creating a new project (Step 1)

The FPGA device, needed for the implementation, was then selected.



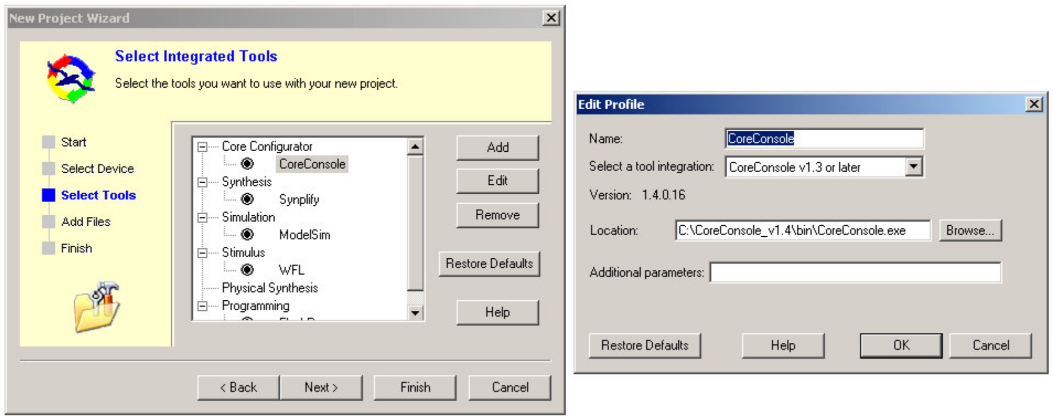
Family: ProASIC3

Die: A3P400, A3P600, M1A3P600, A3P1000, M7A3P1000, M1A3P1000 (selected)

Package: 144 FBGA, 208 PQFP, 256 FBGA, 484 FBGA (selected)

Fig. 2: Creating a new project (Step 2)

Having clicked on `Next` once more, it was then possible to select the individual tools that would be used in the development of the solution. If that were the very first project created in Libero, it would, additionally, be necessary to specify the complete file paths to all these tools.



Core Configurator: CoreConsole
Synthesis: Synplify
Simulation: ModelSim
Stimulus: WFL
Physical Synthesis
Programming

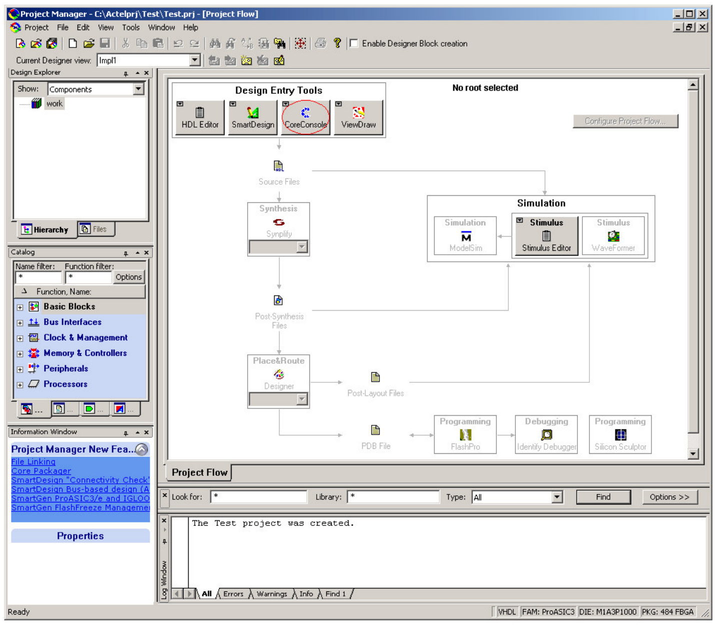
Name: CoreConsole
Select a tool integration: CoreConsole v1.3 or later
Version: 1.4.0.16
Location: C:\CoreConsole_v1.4\bin\CoreConsole.exe
Additional parameters:

Fig. 3: Creating a new project (Step 3)

The next dialogue allowed all existing source files to be added to the new project. Finally, a summary of all the selected options was presented, and the creation of the project was confirmed with a click on the `Finish` button.

2.2 Creating the microprocessor with CoreConsole

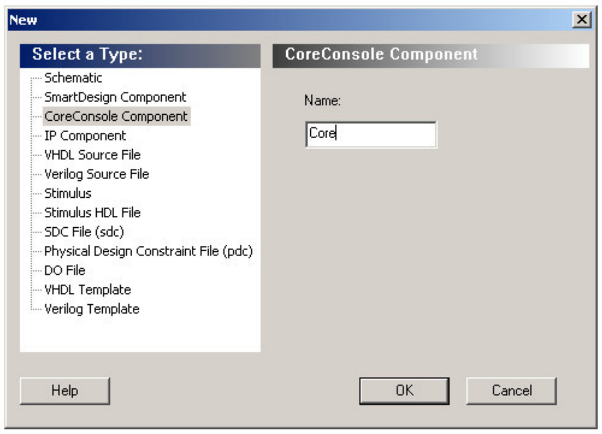
The tool CoreConsole was used for defining the microprocessor. It can be invoked by clicking on the CoreConsole icon in the Design Entry Tool.



```
The Test project was created.
```

Fig. 4: Empty project

The microprocessor was added to the project as a component and had to be identified, as shown in the following dialogue. CoreConsole was then started with a click on `OK`.



New

Select a Type:

- Schematic
- SmartDesign Component
- CoreConsole Component
- IP Component
- VHDL Source File
- Verilog Source File
- Stimulus
- Stimulus HDL File
- SDC File (sdc)
- Physical Design Constraint File (pdc)
- DO File
- VHDL Template
- Verilog Template

CoreConsole Component

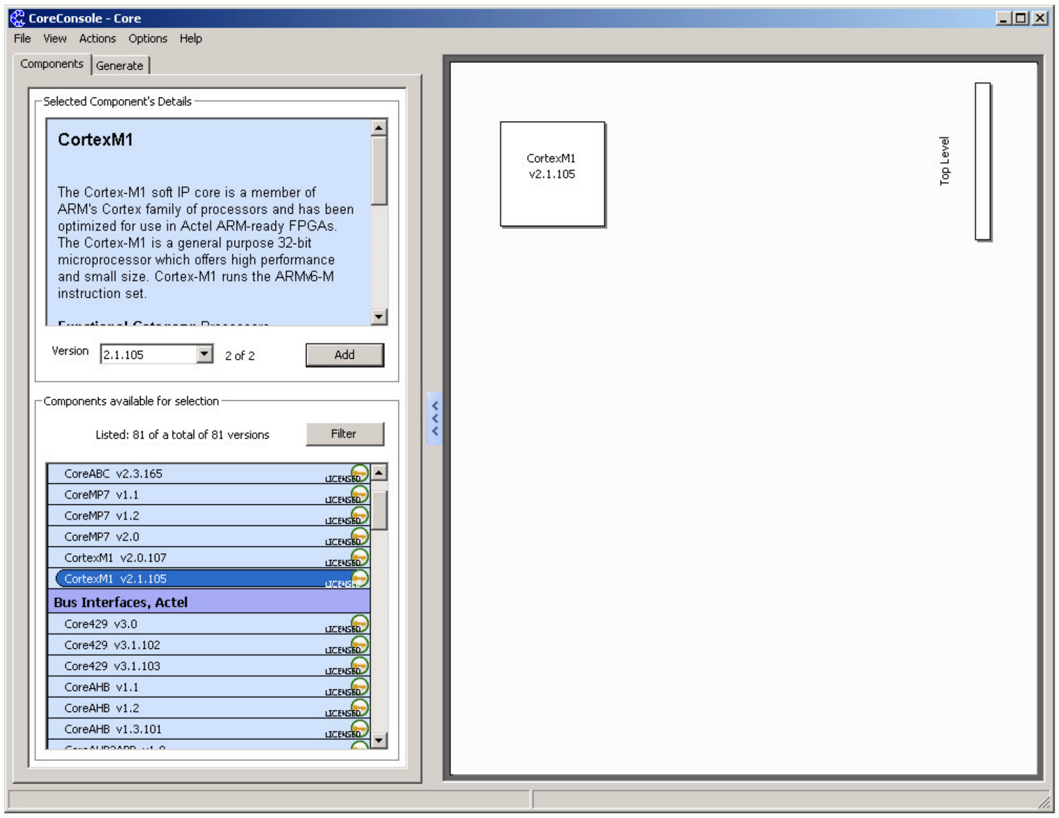
Name: Core

Fig. 5: Creating a CoreConsole component

CoreConsole is divided in three sections. The largest one, on the right, is the working area in which the microprocessor will be put together. On the left, in the lower section, there is a list of all available parts. Selecting one of those results in the corresponding short description appearing in the upper section together with associated links to the relevant datasheets.

2.2.1 Selection of parts

Any part from the list can be placed in the working area by selecting it and clicking the `Add` button. The same can also be achieved by double-clicking on the required part in the list.



CortexM1

The Cortex-M1 soft IP core is a member of ARM's Cortex family of processors and has been optimized for use in Actel ARM-ready FPGAs. The Cortex-M1 is a general purpose 32-bit microprocessor which offers high performance and small size. Cortex-M1 runs the ARMv6-M instruction set.

Version: 2.1.105 — 2 of 2

Listed: 81 of a total of 81 versions

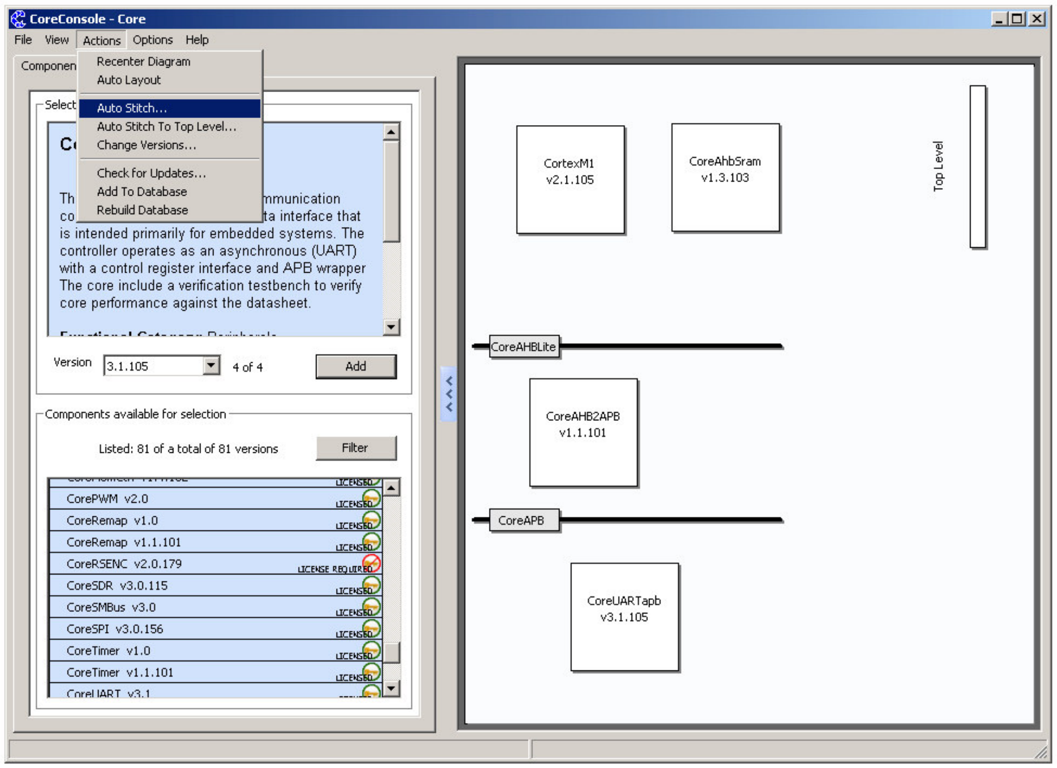
CoreABC v2.3.165 LICENSED
CoreMP7 v1.1 LICENSED
CoreMP7 v1.2 LICENSED
CoreMP7 v2.0 LICENSED
CortexM1 v2.0.107 LICENSED
CortexM1 v2.1.105 LICENSED
Bus Interfaces, Actel
Core429 v3.0 LICENSED
Core429 v3.1.102 LICENSED
Core429 v3.1.103 LICENSED
CoreAHB v1.1 LICENSED
CoreAHB v1.2 LICENSED
CoreAHB v1.3.101 LICENSED

Fig. 6: Adding new parts in CoreConsole

Memory and input and output interfaces were needed, in addition to the microprocessor core, as well as buses to ‘bind’ all those components together. The FPGA’s internal memory, associated with the CoreAhbSram part, was chosen. The bus component, CoreAHBLite, was chosen to connect the memory and the processor core together. The requirement for input and output interfaces was realized by a UART, for which the part CoreUARTapb was chosen. This was attached to the bus component CoreAPB. To allow communication between the buses, a bridge was needed, which was realized by the CoreAHB2APB part.

2.2.2 Connecting the parts together

Having chosen the required parts, it is now possible to either manually connect them together or let CoreConsole establish all connections automatically. The latter was chosen by clicking on `Actions/Auto Stich...`.

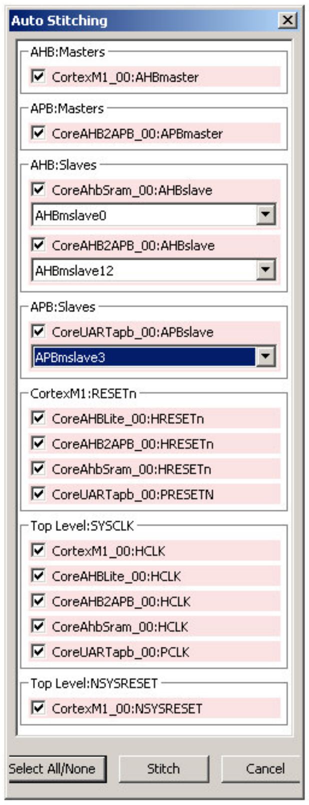


Th mmunication
co ta interface that
is intended primarily for embedded systems. The
controller operates as an asynchronous (UART)
with a control register interface and APB wrapper
The core include a verification testbench to verify
core performance against the datasheet.

CorePWM v2.0 LICENSED
CoreRemap v1.0 LICENSED
CoreRemap v1.1.101 LICENSED
CoreRSENC v2.0.179 LICENSE REQUIRED
CoreSDR v3.0.115 LICENSED
CoreSMBus v3.0 LICENSED
CoreSPI v3.0.156 LICENSED
CoreTimer v1.0 LICENSED
CoreTimer v1.1.101 LICENSED
CoreUART v3.1

Fig. 7: Connections performed automatically

The connections, which were to be performed automatically, were then selected in the dialogue box that appeared as a result. Bus masters appeared at the top of this dialogue, followed by the bus slaves, followed by other control signals such as clock and reset. In the case of the bus slaves, the bus port, to which they should be connected, could additionally be defined.



Auto Stitching

AHB:Masters
- [✓] CortexM1_00:AHBmaster

APB:Masters
- [✓] CoreAHB2APB_00:APBmaster

AHB:Slaves
- [✓] CoreAhbSram_00:AHBslave
  AHBmslave0
- [✓] CoreAHB2APB_00:AHBslave
  AHBmslave12

APB:Slaves
- [✓] CoreUARTapb_00:APBslave
  APBmslave3

CortexM1:RESETn
- [✓] CoreAHBLite_00:HRESETn
- [✓] CoreAHB2APB_00:HRESETn
- [✓] CoreAhbSram_00:HRESETn
- [✓] CoreUARTapb_00:PRESETN

Top Level:SYSCLK
- [✓] CortexM1_00:HCLK
- [✓] CoreAHBLite_00:HCLK
- [✓] CoreAHB2APB_00:HCLK
- [✓] CoreAhbSram_00:HCLK
- [✓] CoreUARTapb_00:PCLK

Top Level:NSYSRESET
- [✓] CortexM1_00:NSYSRESET

Fig. 8: Automatic connections dialogue box

A click on the `Stich` button generated all the selected connections.

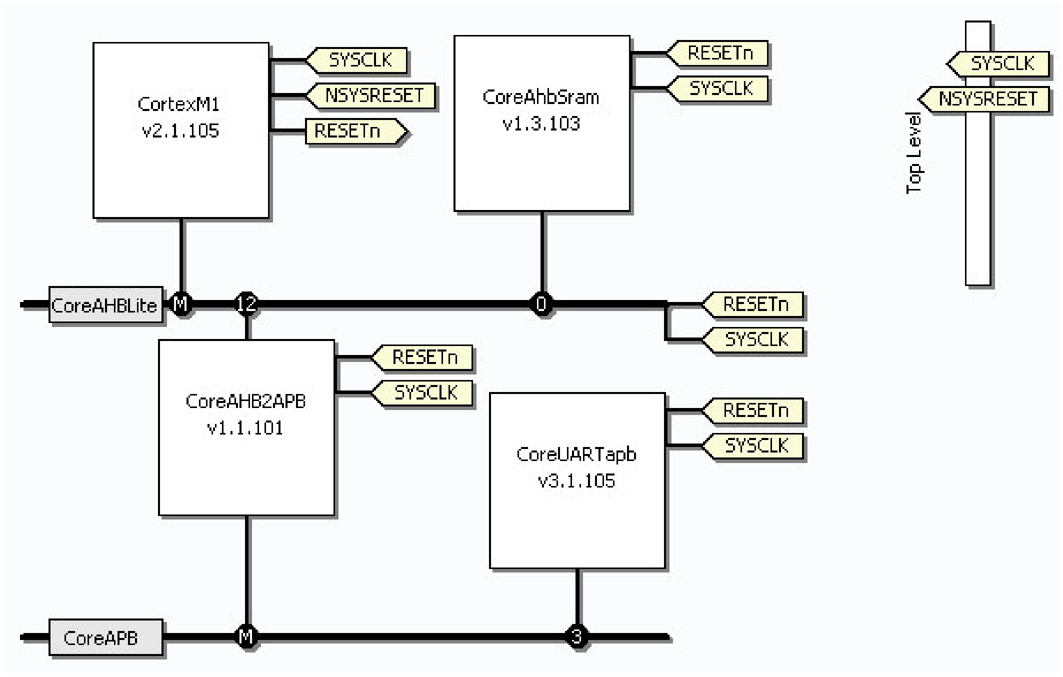
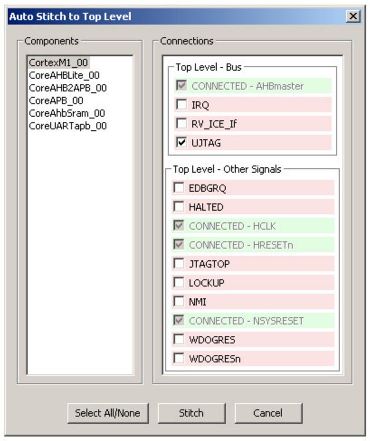


Fig. 9: The result of automatically performed connections

In addition to the two signals, Clock and Reset, already specified above, there were other signals that had to be included in the top level interfaces: the processor's UJTAG signals, necessary for loading programs and for debugging, the RX and TX signals of the UART and one port of the APB bus. To perform these top level connections the option `Actions/Auto Stitch To Top Level...` was selected.



Auto Stitch to Top Level

Components
- CortexM1_00
- CoreAHBLite_00
- CoreAHB2APB_00
- CoreAPB_00
- CoreAhbSram_00
- CoreUARTapb_00

Connections

Top Level - Bus
- [✓] CONNECTED - AHBmaster
- [ ] IRQ
- [ ] RV_ICE_If
- [✓] UJTAG

Top Level - Other Signals
- [ ] EDBGRQ
- [ ] HALTED
- [✓] CONNECTED - HCLK
- [✓] CONNECTED - HRESETn
- [ ] JTAGTOP
- [ ] LOCKUP
- [ ] NMI
- [✓] CONNECTED - NSYSRESET
- [ ] WDOGRES
- [ ] WDOGRESn

Fig. 10: Defining top level connections

The left pane of the dialogue box that appeared contained the constituant parts of the processor system defined thus far. Whenever one of those was selected, the right pane showed the corresponding signals that could be connected to the top level. Any signal on a green background indicates that it is already connected.

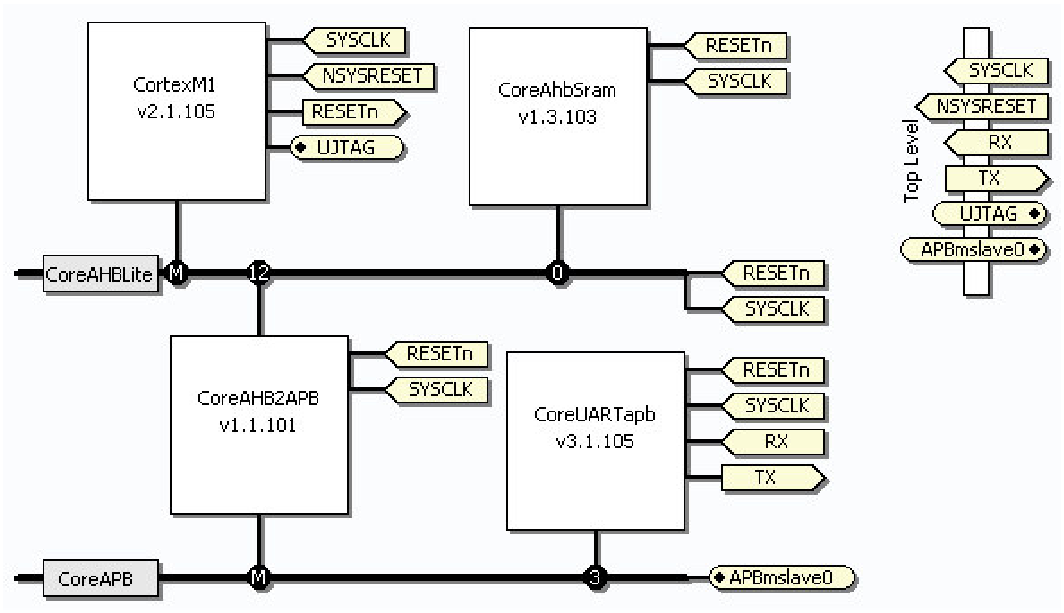


Fig. 11: The processor with all the required signals

2.2.3 Configuring the parts

Next came the configuration of the processor system's components. This was done by moving the mouse pointer over each of these parts and then selecting the `Configure` option. Right-clicking on each part would perform the same task.

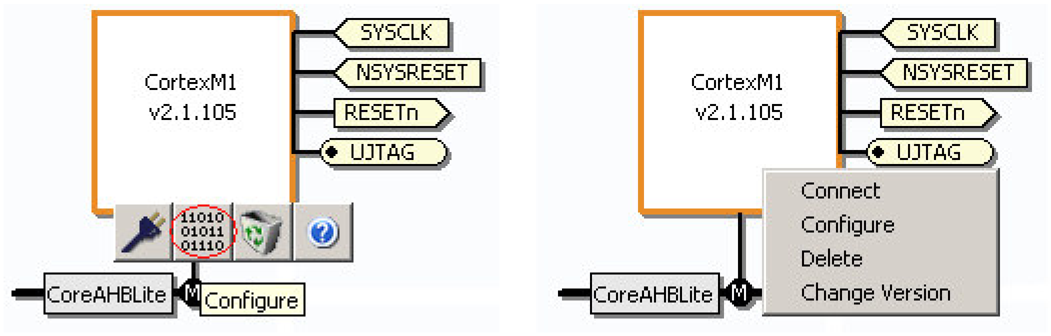
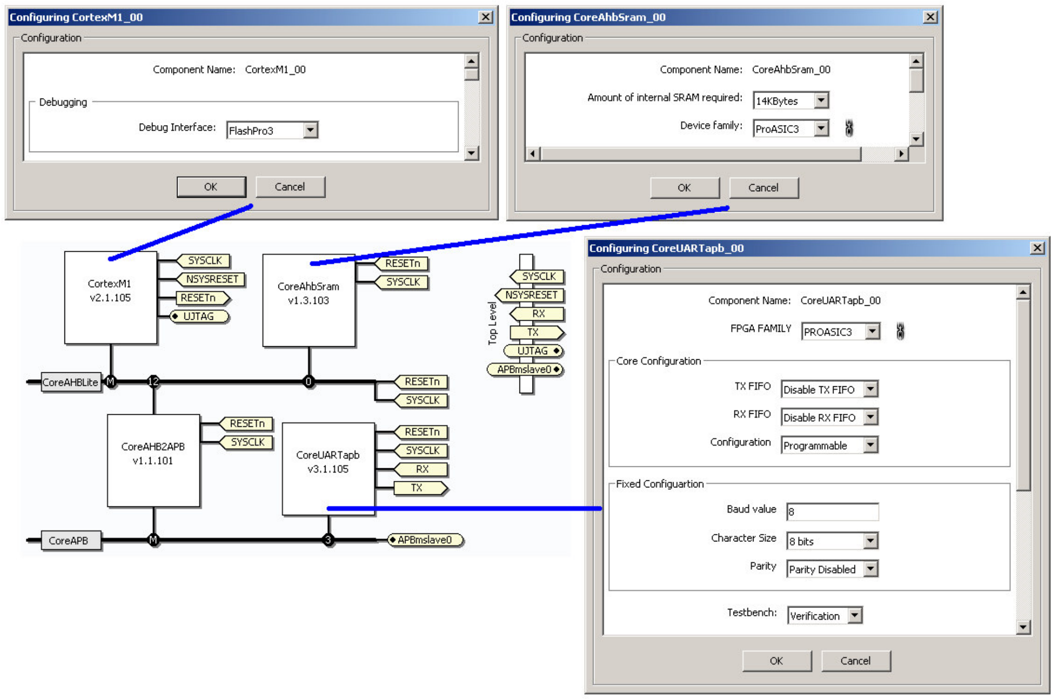


Fig. 12: Component configuration

In the software version used in this project, most of the configuration options for the Cortex-M1 processor core were not available. Only a suitable debugging interface could be selected which, when using the Actel Software, it was the Flash Pro 3.

The only option that needed to be configured for the memory component was its size, with 14 kByte being the maximum possible.

The options for the UART, shown in Abb. 13, indicate that the component was configured without any transmit or receive FIFOs, with a data size of 8 bits without parity and that the configuration could be modified at any time by software. The baud rate was configured for a clock frequency of 16 MHz and a data rate of 115200 Baut. More precise configuration details are included in the datasheet.



Configuring CortexM1_00

Configuration

Component Name: CortexM1_00

Debugging

Debug Interface: FlashPro3

Configuring CoreAhbSram_00

Configuration

Component Name: CoreAhbSram_00

Amount of internal SRAM required: 14KBytes

Device family: ProASIC3

Configuring CoreUARTapb_00

Configuration

Component Name: CoreUARTapb_00

FPGA FAMILY PROASIC3

Core Configuration

TX FIFO Disable TX FIFO

RX FIFO Disable RX FIFO

Configuration Programmable

Fixed Configuartion

Baud value 8

Character Size 8 bits

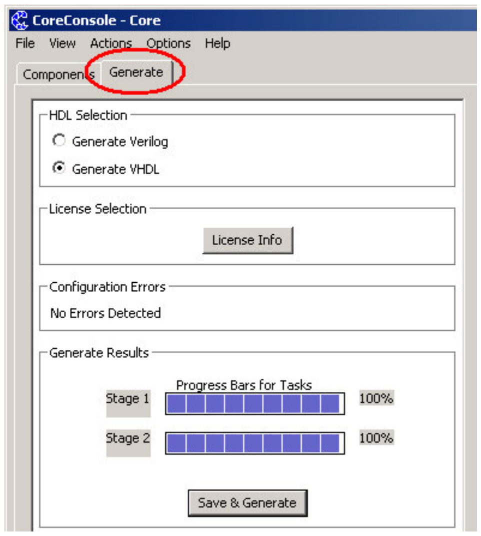
Parity Parity Disabled

Testbench: Verification

Fig. 13: Configuration options for the various components

2.2.4 Generating the processor system

Once all the options were configured, the processor system was generated. For this, under the tab `Generate`, the required HDL was selected (VHDL in this project), in which the code for the processor system should be generated. Finally the `Save & Generate` button was clicked to start the process. Once this was complete, it was possible to exit the CoreConsole.



Generate Verilog [✓ Generate VHDL]

Configuration Errors
No Errors Detected

Stage 1 100%
Stage 2 100%

Fig. 14: Generating the processor system

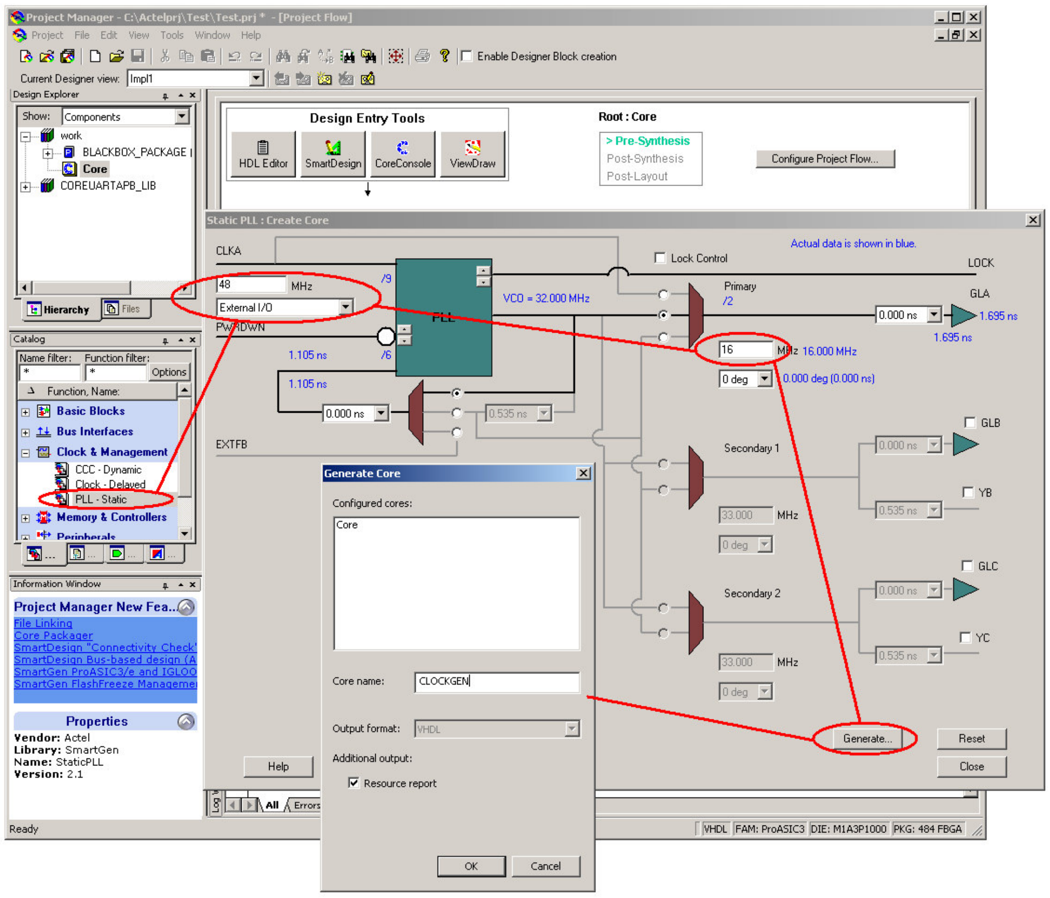
2.2.5 Memory Map

The processor's memory map is determined by the position of the individual components on the bus. The AHB bus divides the overall addressable space of 4GByte in sixteen equal sections, each with a size of 256MByte. A connected APB bus further divides such a 256MByte section in sixteen subsections, each with a size of 16MByte.

The UART occupies the third slot of the APB bus which, in turn, is attached on the twelfth slot of the AHB bus. As a result, it can be accessed at address 0xC3000000.

2.3 Clock setup

The processor system should be clocked at a frequency of 16 MHz, on the PCB, however, there was an oscillator providing a 48 MHz clock. With the Clock Control Circuits, also available on the PCB, it was possible to generate several different, individually configurable, clocks from this ‘master’ clock. To generate the required 16 MHz clock signal, the option `Clock & Management / PLL – Static` had to be selected (see illustration below). In the resulting dialogue box, the input clock was defined with a frequency of 48 MHz and the option `External I/O` was specified. The frequency of the output clock was defined as 16 MHz. A click on `Generate...` opened a new dialogue, allowing to name this clock generating component. A final click on the latter’s `OK` button started the component generating process.



Current Designer view: Impl1

Show: Components

Properties
Vendor: Actel
Library: SmartGen
Name: StaticPLL
Version: 2.1

CLKA: 48 MHz, External I/O

VCO = 32.000 MHz

Primary /2: 16 MHz 16.000 MHz, 0 deg 0.000 deg (0.000 ns)

0.000 ns 1.695 ns

Configured cores: Core

Core name: CLOCKGEN

Output format: VHDL

Additional output: Resource report [✓]

Ready | VHDL | FAM: ProASIC3 | DIE: M1A3P1000 | PKG: 484 FBGA

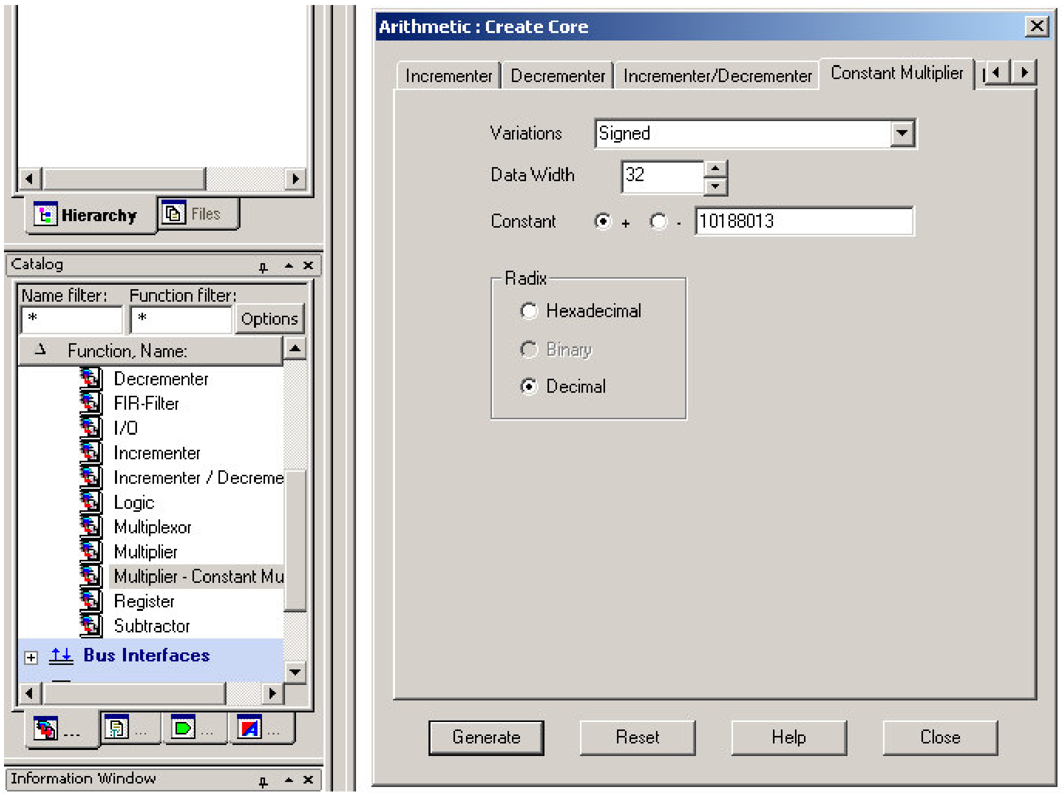
Fig. 15: PLL setup

2.4 Adding user-defined hardware

The generated result in the Cordic Algorithm must be multiplied by a constant. In this project, this multiplication was performed by additional hardware.

2.4.1 Generating the hardware

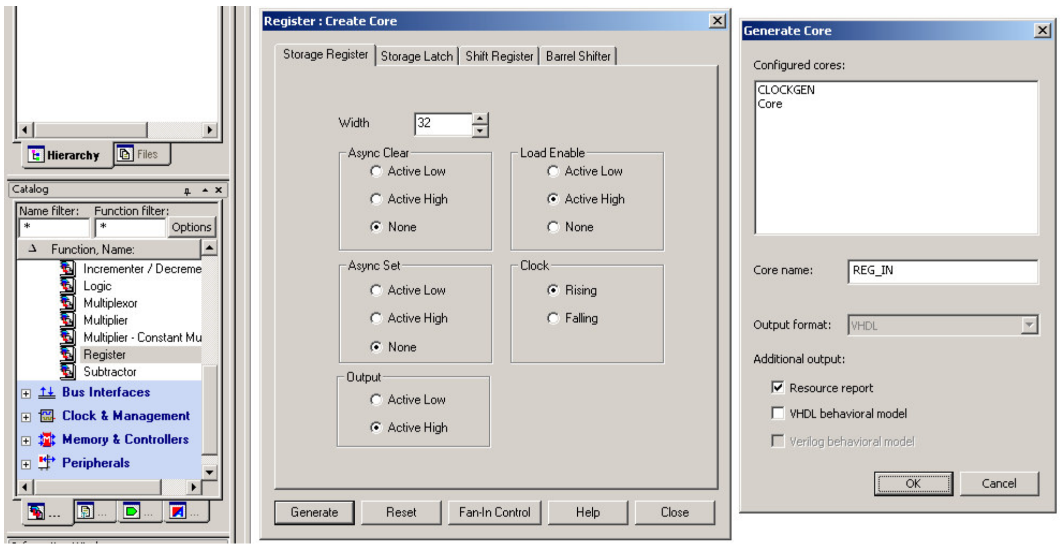
With the option `Basic Blocks / Multiplier - Constant Multiplier` it was possible to have the tools generate a multiplier.



Variations: Signed
Data Width: 32
Constant: [✓ +] - 10188013
Radix: Hexadecimal / Binary / [✓ Decimal]

Fig. 16: Multiplier setup

Two registers were also needed: one in which the processor wrote the data to be used in the calculation and a second one from which it could read its result. Both registers were generated using the option `Basic Block / Register` (see illustration below). Abb. 17 shows the configuration of the input register. The output register did not require a Load Enable input signal.



Register : Create Core

Storage Register | Storage Latch | Shift Register | Barrel Shifter

Width: 32

Async Clear: Active Low [✓ None] Active High

Load Enable: Active Low [✓ Active High] None

Async Set: Active Low Active High [✓ None]

Clock: [✓ Rising] Falling

Output: Active Low [✓ Active High]

Generate Core

Configured cores:
CLOCKGEN
Core

Core name: REG_IN

Output format: VHDL

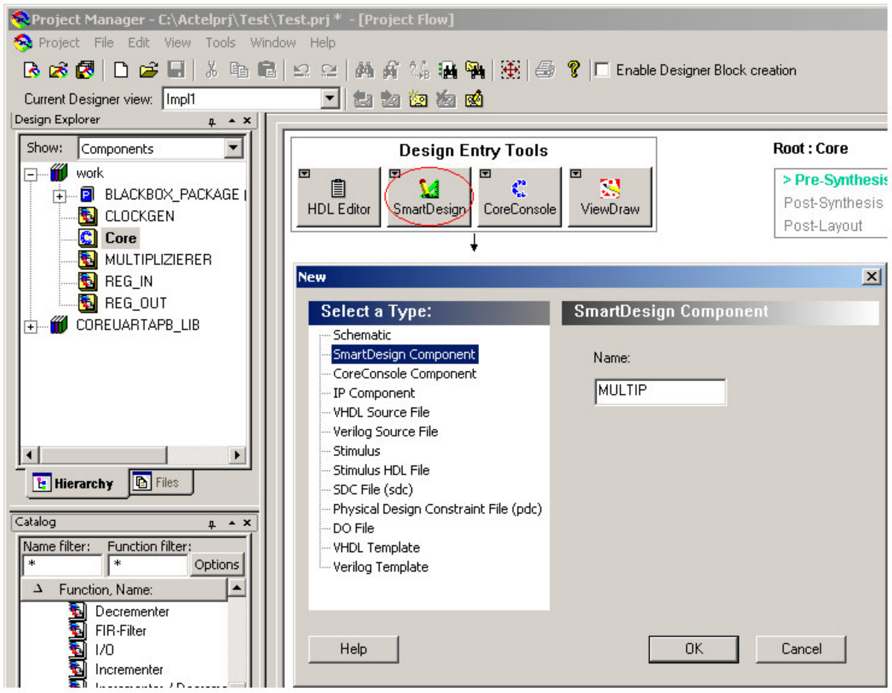
Additional output:
[✓ Resource report]
VHDL behavioral model
Verilog behavioral model

Fig. 17: Configuration of the input register

2.4.2 Working with Smart Design

It was sensible to include both the multiplier and the registers in a single block. This could be done either with VHDL or by using the Smart Design tool.

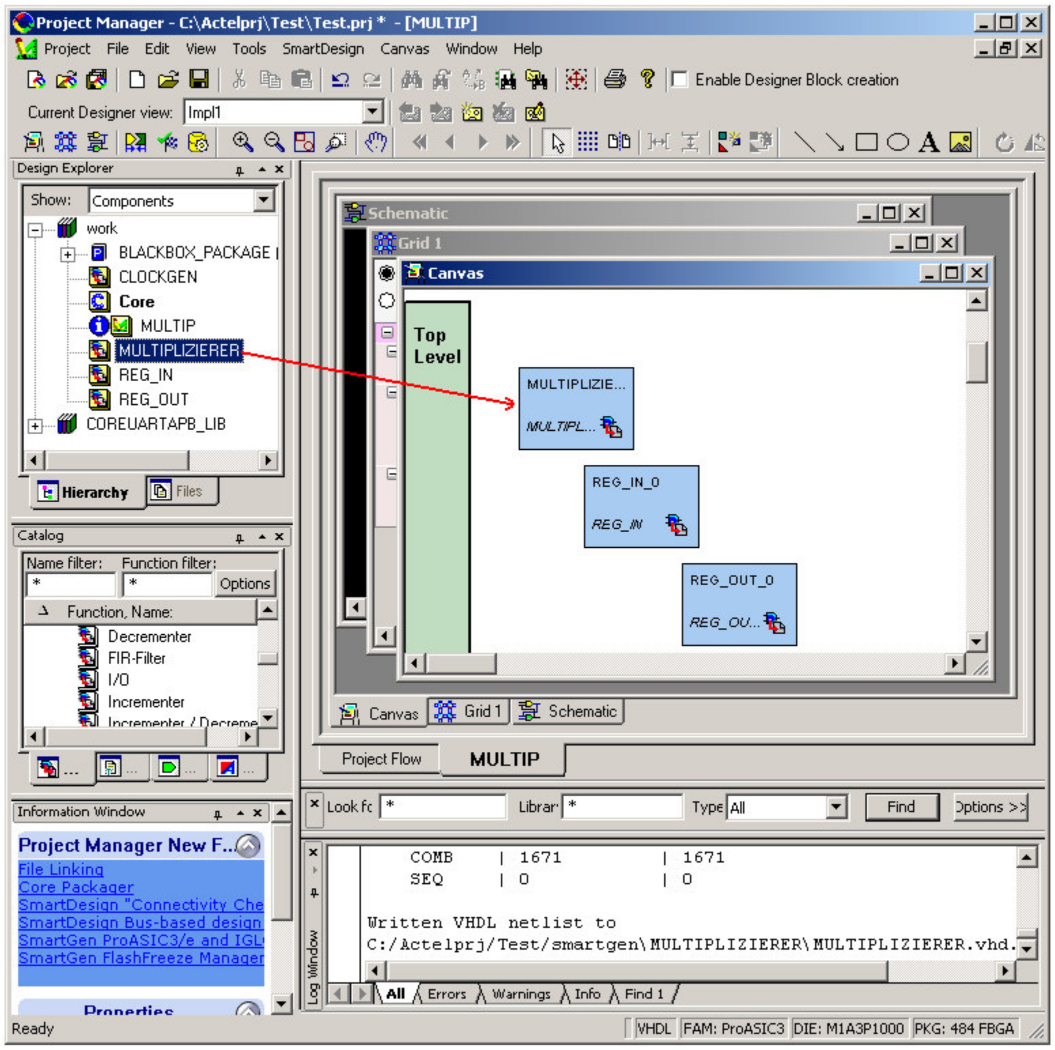
Smart Design can be found among the Design Entry Tools., The new hardware block had to be given a name when the tool was first started.



Name: MULTIP

Fig. 18: Invoking the Smart Design tool

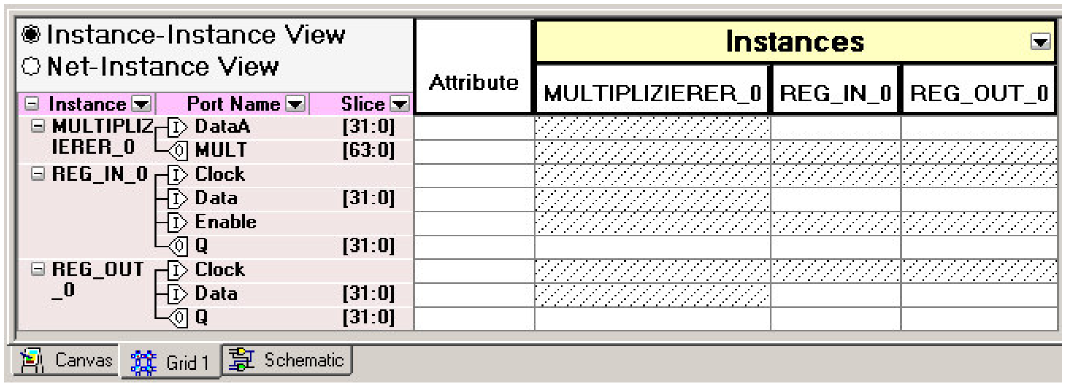
Then the new block’s individual components were added to Smart Design. This was done simply by dragging the components from the hierarchical list on the left (see illustration below) onto the Canvas window on the right. The three components were then connected to each other and to the top level.



```
    COMB     |  1671          |  1671
    SEQ      |  0             |  0

Written VHDL netlist to
C:/Actelprj/Test/smartgen\MULTIPLIZIERER\MULTIPLIZIERER.vhd.
```

Fig. 19: Adding components to the Canvas

This was done by changing to the Grid window: on the left, there is a list of all constituant components and their I/O ports; on the right, the same components are listed again, this time in tabular form. In order to establish a connection between, say, the output port of the input register (`REG_IN_0: Q`) and the input port of the multiplier (`MULTIPLIZIERER_0: DataA`), one has to click on the cell at the intersection of the row `REG_IN_0: Q` and column `MULTIPLIZIERER_0`. In this way, all possible connections between the two ports will be shown. In this particular case, there was only a single possible connection which was built with a single click.

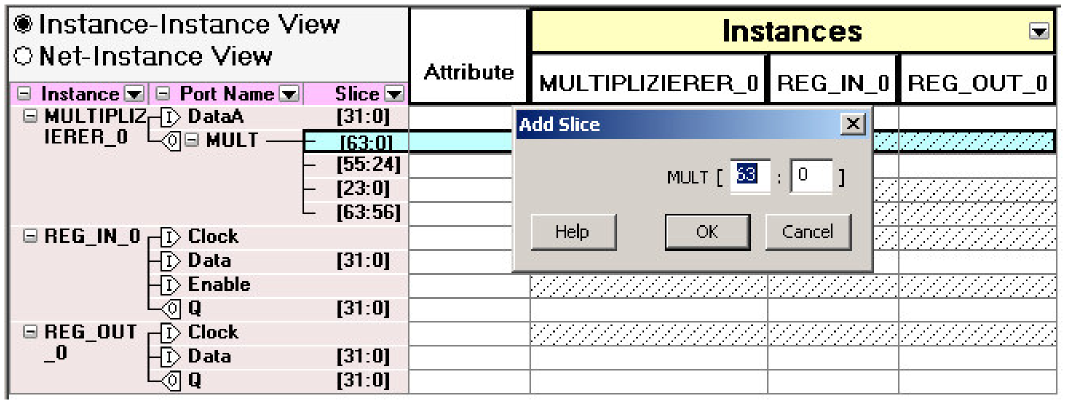


Instance	Port Name	Slice	Attribute	MULTIPLIZIERER_0	REG_IN_0	REG_OUT_0
MULTIPLIZIERER_0	DataA	[31:0]		▨		
	MULT	[63:0]		▨	▨	▨
REG_IN_0	Clock			▨	▨	▨
	Data	[31:0]		▨		
	Enable			▨	▨	▨
	Q	[31:0]				
REG_OUT_0	Clock			▨	▨	▨
	Data	[31:0]		▨		
	Q	[31:0]				

Fig. 20: The Grid window showing the still unconnected components

In addition to the new block's internal connections, described above, it was necessary to establish its 'top level' ports, to allow the block to be connected to the other components of the processor system. These were defined by right-clicking on the appropriate ports of the block's individual components (for example on the input port of the input register, `REG_IN_0: DATA`) and select the option `Promote To Top Level`. It is also possible to rename any of these top level ports by right-clicking on it and selecting the option `Modify Top Level Port...`.

The result of the multiplication of two fixed point numbers, each with 24 decimal points, is too large to fit in the output port (by a factor of 2^24). This problem can be resolved by ignoring the last 24 bits of the result. This is why only bits 24-55 of the multiplier's output port were connected to the output register's input port. This was done by splitting the 64-bit bus at the multiplier's output by right-clicking on it and selecting `Add Slice...` option.



◉ Instance-Instance View
○ Net-Instance View

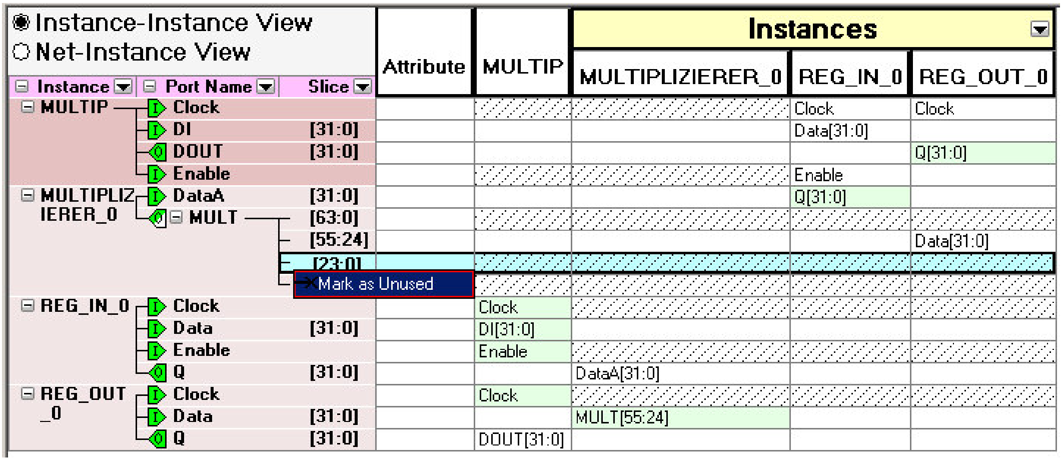
Instance	Port Name	Slice	Attribute	MULTIPLIZIERER_0	REG_IN_0	REG_OUT_0
MULTIPLIZIERER_0	DataA	[31:0]				
	MULT	[63:0]			▨	▨
		[55:24]				
		[23:0]			▨	▨
		[63:56]			▨	▨
REG_IN_0	Clock				▨	▨
	Data	[31:0]				
	Enable			▨	▨	▨
	Q	[31:0]				
REG_OUT_0	Clock			▨	▨	▨
	Data	[31:0]				
	Q	[31:0]				

Add Slice

MULT [63 : 0]

Fig. 21: Bus splitting

The remaining ports of the bus were not used. To avoid subsequent unnecessary warnings these ports were declared 'unused' by clicking on the `Attribute` column and selecting the option `Mark as Unused`.



◉ Instance-Instance View
○ Net-Instance View

Instance	Port Name	Slice	Attribute	MULTIP	MULTIPLIZIERER_0	REG_IN_0	REG_OUT_0
MULTIP	Clock			▨	▨	Clock	Clock
	DI	[31:0]				Data[31:0]	
	DOUT	[31:0]					Q[31:0]
	Enable			▨	▨	Enable	
MULTIPLIZIERER_0	DataA	[31:0]				Q[31:0]	
	MULT	[63:0]		▨	▨	▨	▨
		[55:24]					Data[31:0]
		[23:0]		▨	▨	▨	▨
			Mark as Unused	▨	▨	▨	▨
REG_IN_0	Clock			Clock	▨	▨	▨
	Data	[31:0]		DI[31:0]			
	Enable			Enable	▨	▨	▨
	Q	[31:0]			DataA[31:0]		
REG_OUT_0	Clock			Clock	▨	▨	▨
	Data	[31:0]			MULT[55:24]		
	Q	[31:0]		DOUT[31:0]			

Fig. 22: Marking unused pins

When all connections were done (as shown in Abb. 22), the new, user-defined, hardware block was ready. In the Schematic window, the newly completed block could be seen and examined for any connection errors.

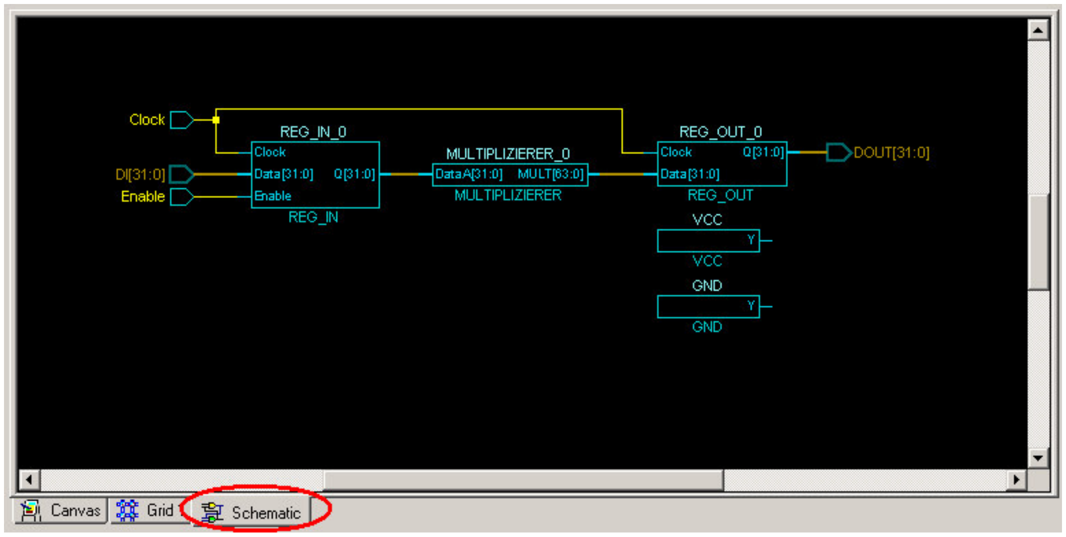


Fig. 23: The Schematic window showing the used-defined multiplier block

VHDL Code could be generated automatically by clicking on the icon `Generate SmartDesign` or thtough the menu option `SmartDesign / Generate`.

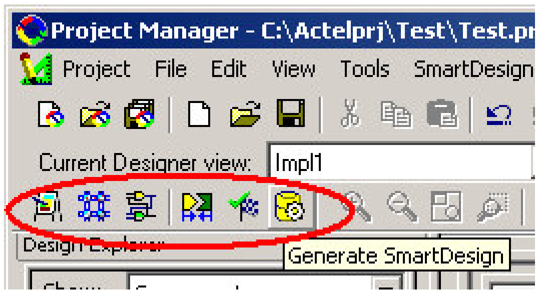
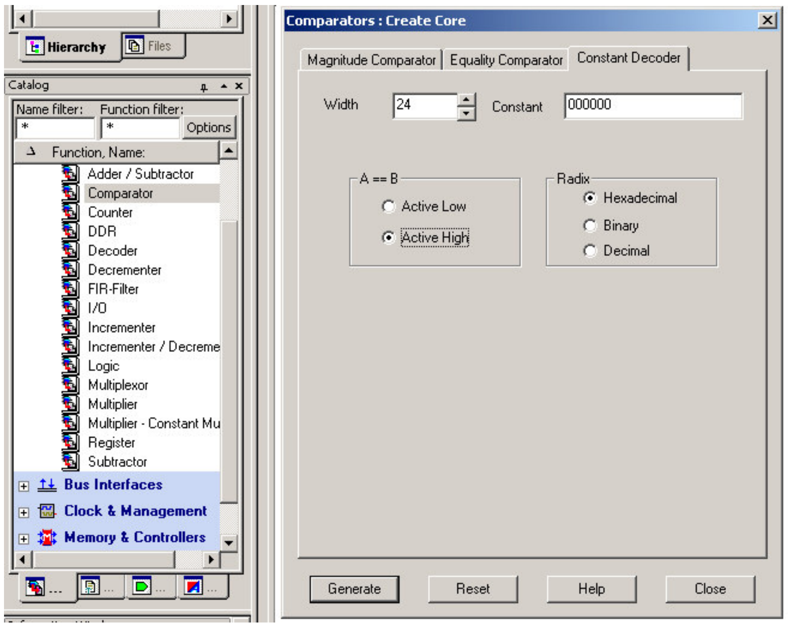


Fig. 24: Generating VHDL Code

2.4.3 Creating a simple bus interface

The newly created hardware block had to be connected to the APB bus of the processor system. A bus interface was, therefore, necessary and, in this case, it was particularly simple to develop: it only had to establish a Write Enable signal for the input register, pass the clock signal on to the multiplier and connect the bus' read- and write-data signals with the block's registers.

Using the `Comparator / Constant Decoder` option, a comparator was automatically generated and configured as an address decoder, i.e. activating a signal whenever the address on the bus 'hit' on the component's address space.

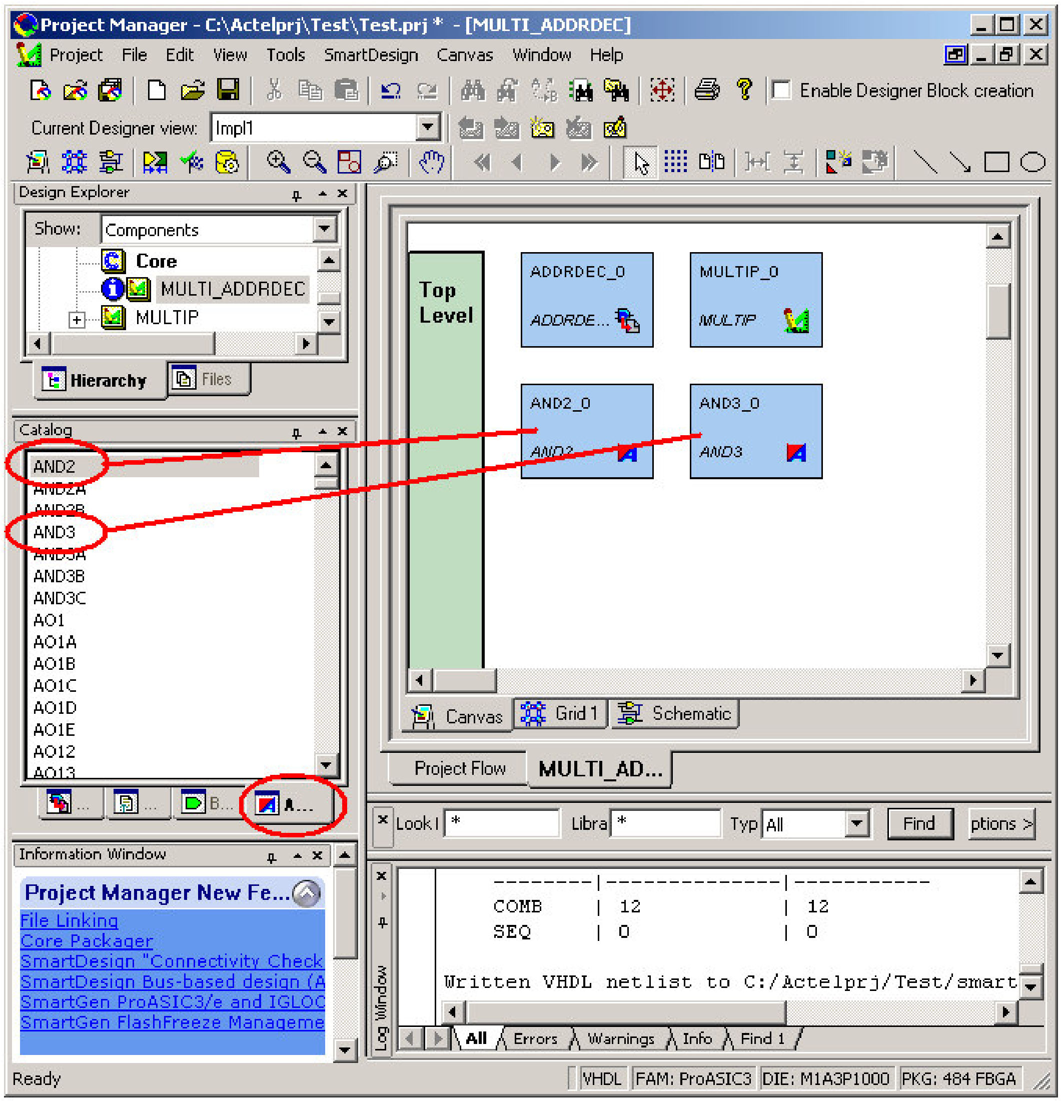


Width: 24
Constant: 000000
A == B: Active Low [✓ Active High]
Radix: [✓ Hexadecimal] Binary Decimal

Fig. 25: Defining a comparator as address decoder

This address decoder was brought together with the previously generated multiplier block in a new Smart Design.

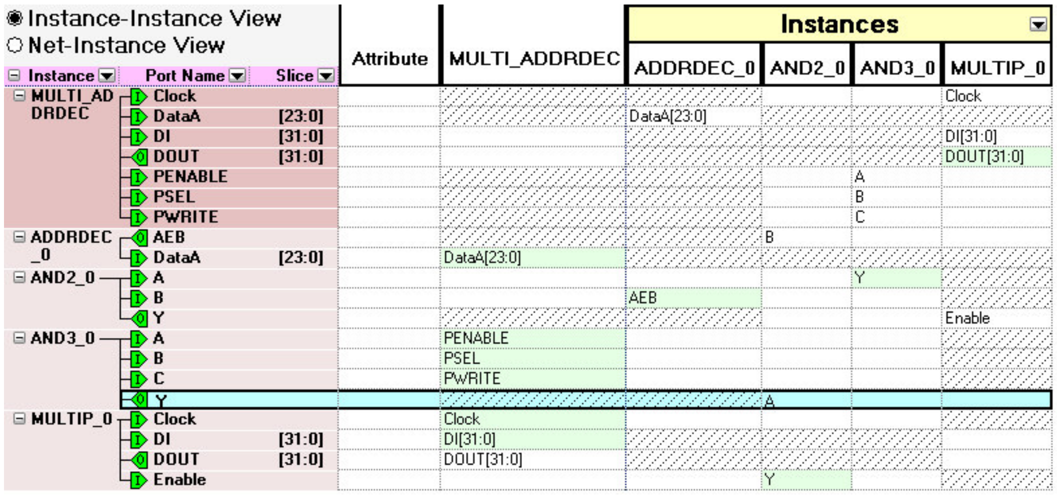
According to the bus protocol, performing a write operation to a peripheral device, involves the signals PSEL, PENABLE and PWRITE. These were combined with a AND-gate. A second AND-gate was needed to include the output of the address decoder. These AND-gates were found in the Actel Cell Library and instantiated in the design by dragging and dropping in the Canvas window.



```
        --------|---------------|-----------
        COMB    |  12           |  12
        SEQ     |  0            |  0

Written VHDL netlist to C:/Actelprj/Test/smart
```

Fig. 26:   Instantiating AND-gates in Smart Design

The connections were built as shown in Abb. 27 and Abb. 28.



[✓ Instance-Instance View]
Net-Instance View

Instance	Port Name	Slice	Attribute	MULTI_ADDRDEC	Instances			
					ADDRDEC_0	AND2_0	AND3_0	MULTIP_0
MULTI_ADDRDEC	Clock							Clock
	DataA	[23:0]			DataA[23:0]			
	DI	[31:0]						DI[31:0]
	DOUT	[31:0]						DOUT[31:0]
	PENABLE						A	
	PSEL						B	
	PWRITE						C	
ADDRDEC_0	AEB					B		
	DataA	[23:0]		DataA[23:0]				
AND2_0	A						Y	
	B				AEB			
	Y							Enable
AND3_0	A			PENABLE				
	B			PSEL				
	C			PWRITE				
	Y					A		
MULTIP_0	Clock			Clock				
	DI	[31:0]		DI[31:0]				
	DOUT	[31:0]		DOUT[31:0]				
	Enable					Y		

Fig. 27: Multiplier with Adress Decoder (Grid window)

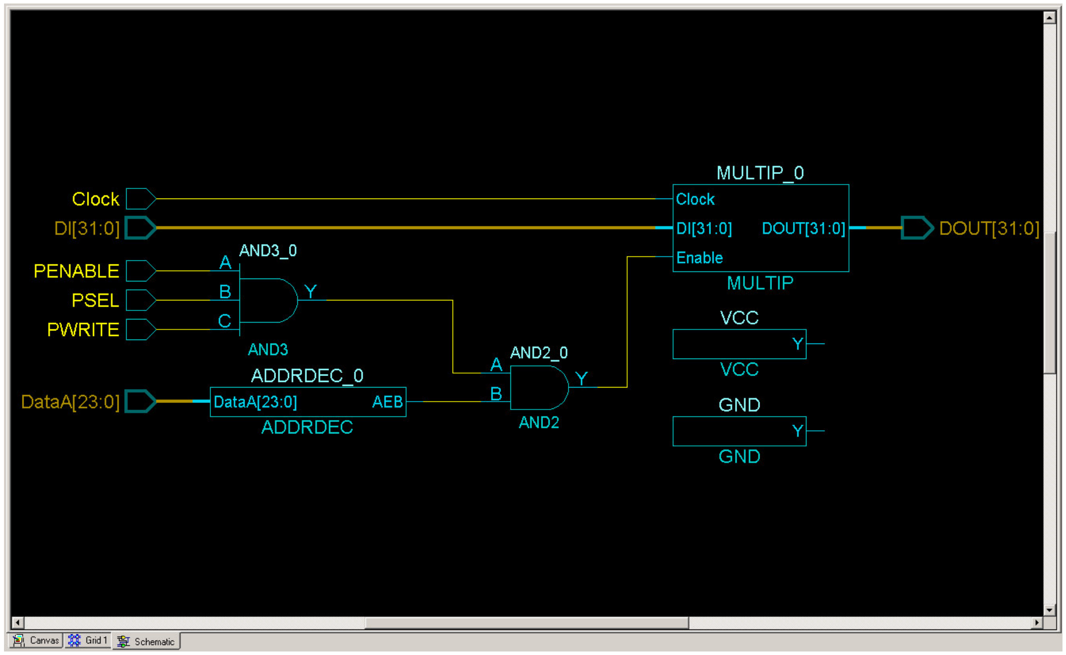
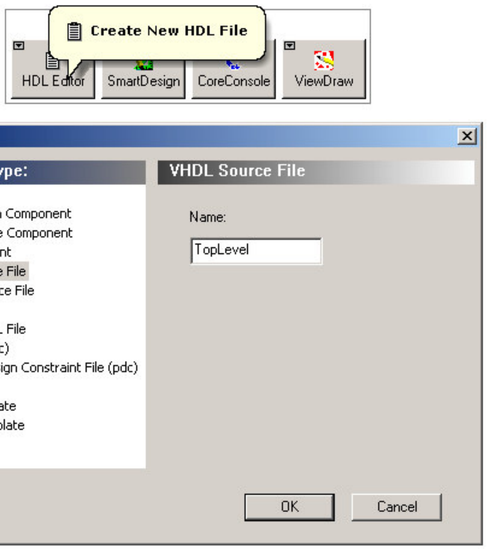


Fig. 28: Multiplier with Adress Decoder (Schematic window)

2.5 Creating the Top Level

When all the components had been generated, they were connected together and the processor system's top level ports were defined and connected to the FPGA's pins. With earlier versions of the Libero software, all this could have been done with Smart Design. With version 8.1, used in this project, there was a problem with the APB bus however.. In Smart Design, the port PRDATA was not displayed and, consequently, could not be connected. Because of this, the top level was developed in VHDL.



VHDL Source File

Name: TopLevel

Fig. 29: Creating the top level VHDL file

Any VHDL files created in this way can be checked for syntax errors. To do this, one needs to right-click on the relevant file and select the `Check HDL File` option. The results are displayed in the Log window.

2.6 Synthesis

For the synthesis, it is necessary to identify the file which represents the top level of the design. This will then be shown in bold in the overall design hierarchy. One can redefine the top level of any design simply by right-clicking on a file in the design's hierarchy, and selecting the option `Set As Root`.

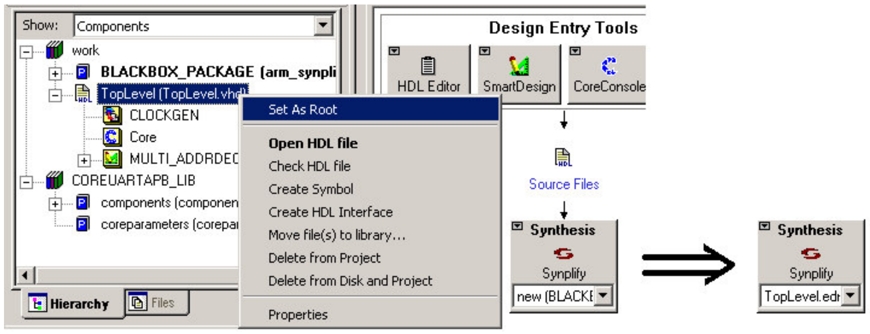
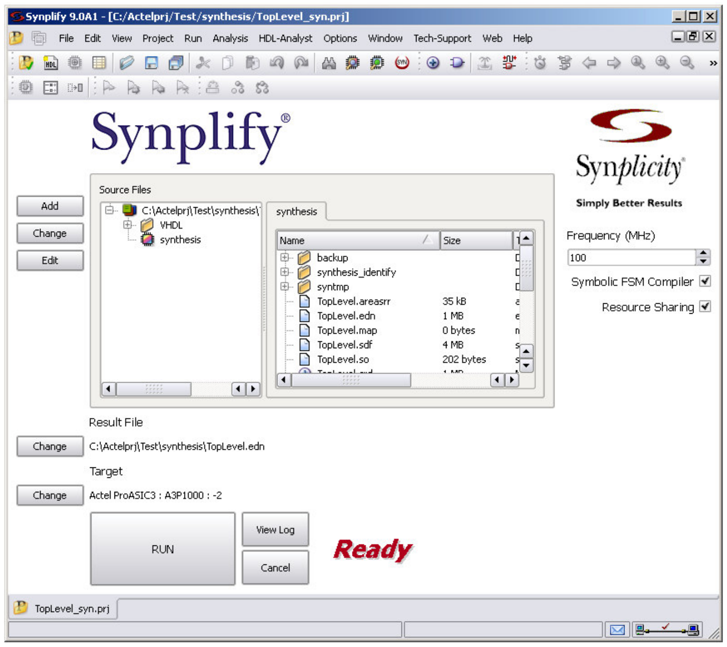


Fig. 30: Selecting the component Top Level as the top level of the project

The Synplify synthesis tool was then invoked with a double-click.

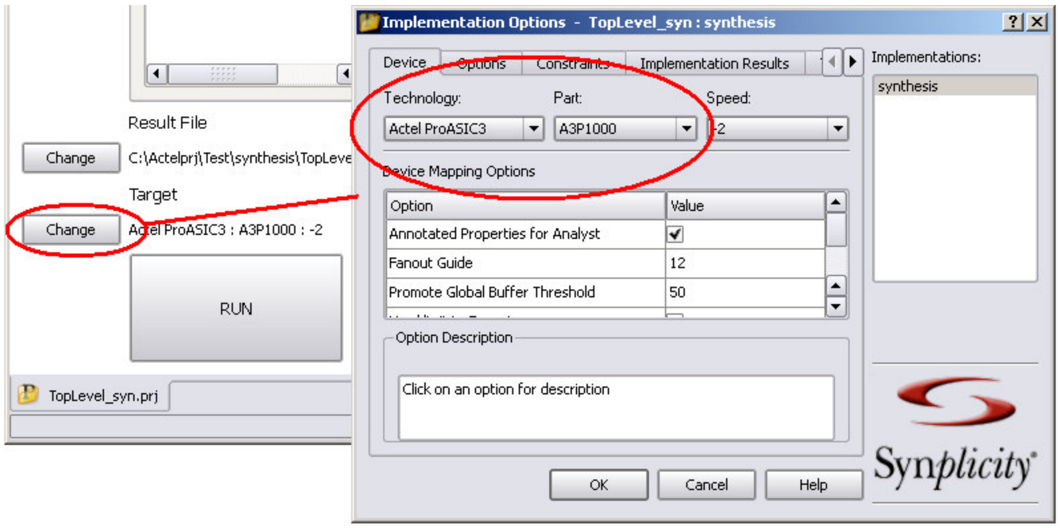


Frequency (MHz): 100
Symbolic FSM Compiler ✓
Resource Sharing ✓
Result File: C:\Actelprj\Test\synthesis\TopLevel.edn
Target: Actel ProASIC3 : A3P1000 : -2

Fig. 31: Synplify

When Synplify first started, there was an error message about the selected FPGA device not being known. To ensure a problem-free run of the tool, it was necessary to specify a different FPGA as the target device. This was done by clicking the lower of the two `Change` buttons: in the dialogue box that opened up, the Actel

ProASIC3 A3P1000 device was selected, which is, both in terms of size and resources, identical to the one on the PCB.



Technology: Actel ProASIC3
Part: A3P1000
Speed: -2

C:\Actelprj\Test\synthesis\TopLeve

Actel ProASIC3 : A3P1000 : -2

Device Mapping Options

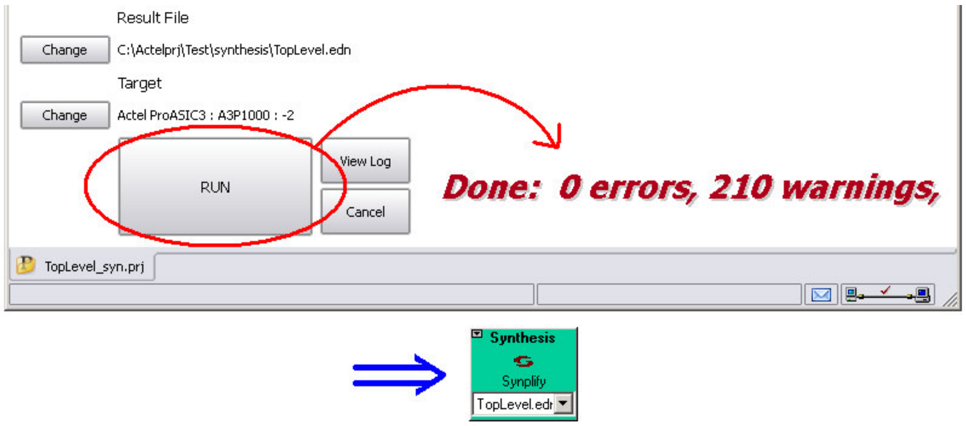
Option	Value
Annotated Properties for Analyst	✓
Fanout Guide	12
Promote Global Buffer Threshold	50

Option Description

Click on an option for description

Fig. 32: Changing the target device

The synthesis process was started with the `Run` button. Clicking on the `View Log` button, displays all warnings and error messages. Warnings in source files that were automatically generated by the tools, can usually be ignored.



C:\Actelprj\Test\synthesis\TopLevel.edn

Actel ProASIC3 : A3P1000 : -2

Done: 0 errors, 210 warnings,

Fig. 33: Starting the synthesis

2.7 Place & Route

A successful synthesis in Synplify is indicated by the colour green. At that point the Designer tool was invoked by clicking on the `Place & Route` button. The default settings were kept and confirmed with a click on `OK`. The compiler was started with a click on the `Compile` button. How long this process will take until it completes depends on the computer on which it is run.

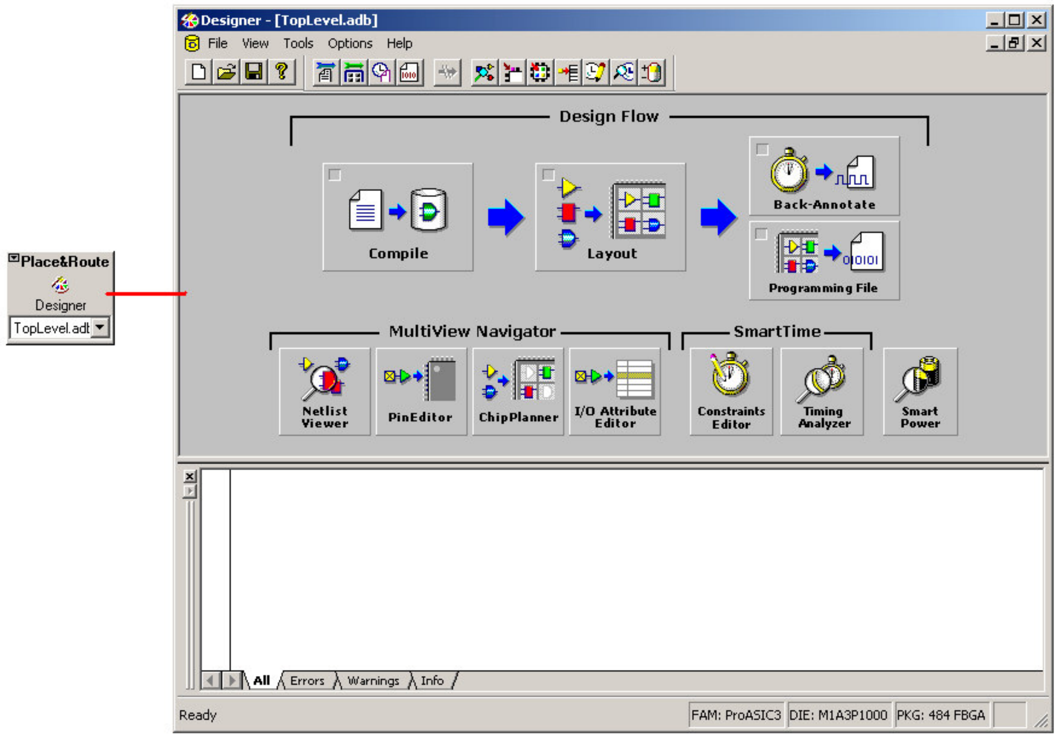
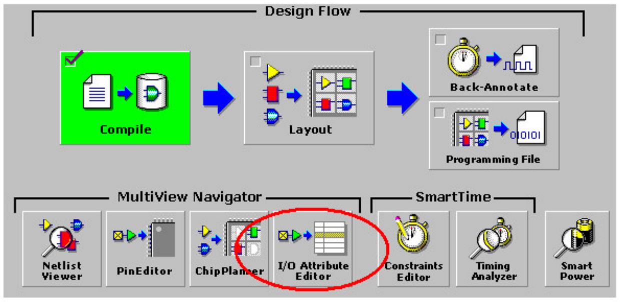
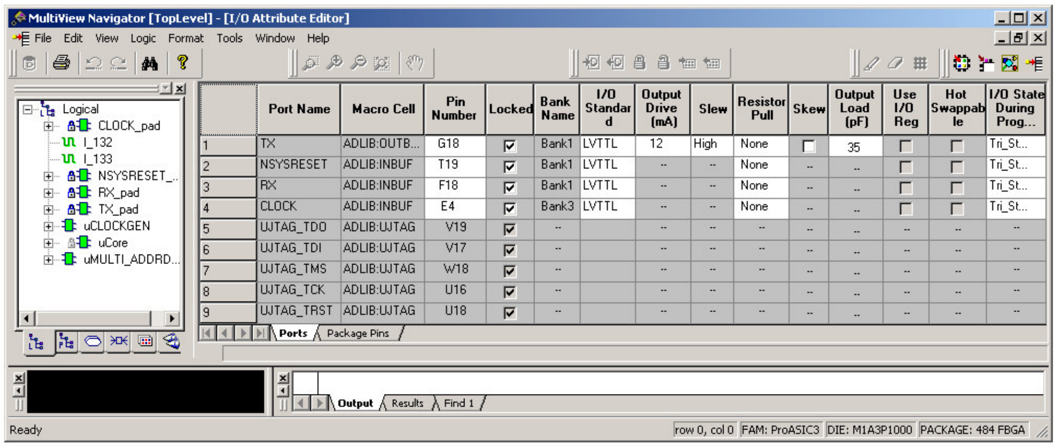


Fig. 34: The Designer tool

Following the compilation, the I/O assignment was performed using the I/O Attribute Editor.





	Port Name	Macro Cell	Pin Number	Locked	Bank Name	I/O Standard	Output Drive (mA)	Slew	Resistor Pull	Skew	Output Load (pF)	Use I/O Reg	Hot Swappable	I/O State During Prog...
1	TX	ADLIB:OUTB...	G18	☑	Bank1	LVTTL	12	High	None	☐	35	☐	☐	Tri_St...
2	NSYSRESET	ADLIB:INBUF	T19	☑	Bank1	LVTTL	--	--	None	--	--	☐	☐	Tri_St...
3	RX	ADLIB:INBUF	F18	☑	Bank1	LVTTL	--	--	None	--	--	☐	☐	Tri_St...
4	CLOCK	ADLIB:INBUF	E4	☑	Bank3	LVTTL	--	--	None	--	--	☐	☐	Tri_St...
5	UJTAG_TDO	ADLIB:UJTAG	V19	☑	--		--	--	--	--	--	--	--	--
6	UJTAG_TDI	ADLIB:UJTAG	V17	☑	--		--	--	--	--	--	--	--	--
7	UJTAG_TMS	ADLIB:UJTAG	W18	☑	--		--	--	--	--	--	--	--	--
8	UJTAG_TCK	ADLIB:UJTAG	U16	☑	--		--	--	--	--	--	--	--	--
9	UJTAG_TRST	ADLIB:UJTAG	U18	☑	--		--	--	--	--	--	--	--	--

Fig. 35: Mapping top level ports to FPGA pins (I/O assignment)

When the I/O assignment was completed, the Layout was started by clicking on the corresponding button. This is another process which could last several minutes, even on powerful computers.

Finally, once the Layout process was successfully completed, the programming file was generated with a click on the `Programming File` button. The Designer tool was then shut down.

2.8 Programming the FPGA

The tool Flash Pro was used to 'program' the FPGA, i.e. to transfer the design (represented by the programming file) onto the FPGA on the PCB. The latter must be connected to the computer via a USB cable to allow this.

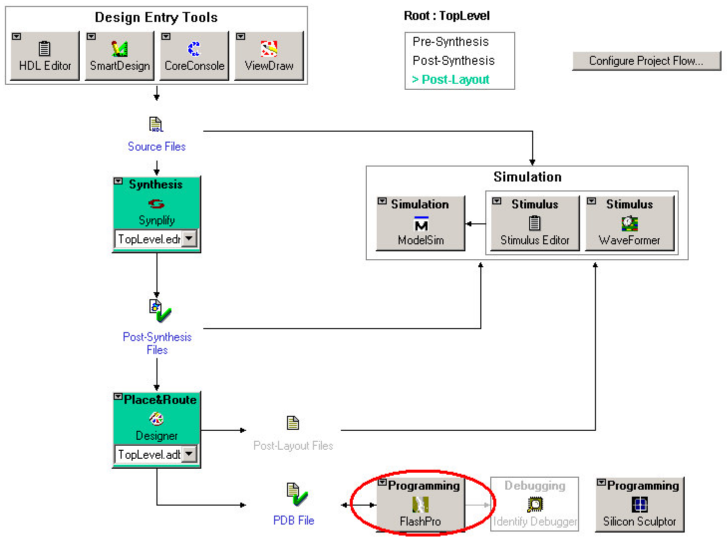
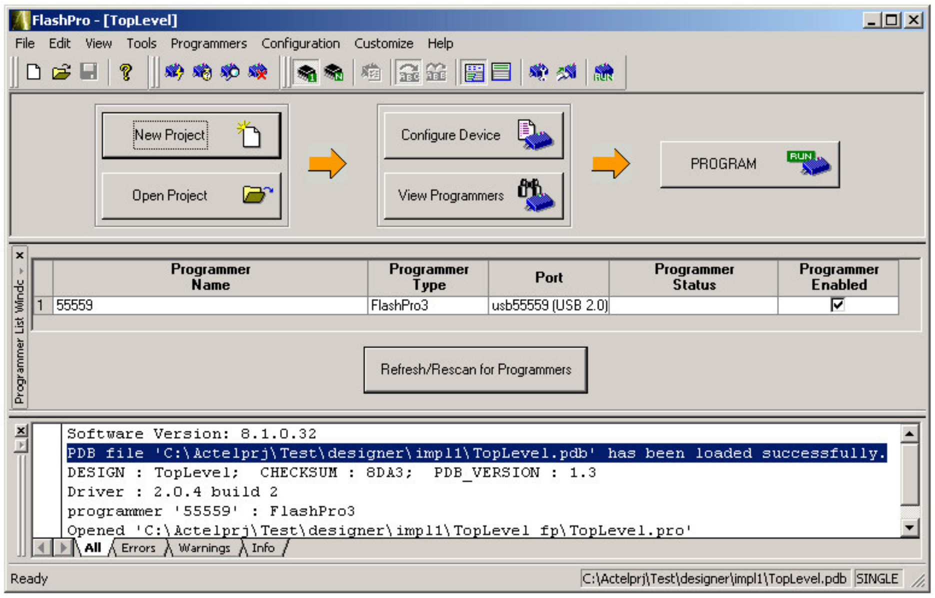


Fig. 36: Invoking the Flash Pro tool

In case no design gets automatically loaded when the tool is started, it is possible to load one manually, by clicking on `Configure Device` and then on `Browse`.



	Programmer Name	Programmer Type	Port	Programmer Status	Programmer Enabled
1	55559	FlashPro3	usb55559 (USB 2.0)		✓

```
Software Version: 8.1.0.32
PDB file 'C:\Actelprj\Test\designer\impl1\TopLevel.pdb' has been loaded successfully.
DESIGN : TopLevel;  CHECKSUM : 8DA3;  PDB_VERSION : 1.3
Driver : 2.0.4 build 2
programmer '55559' : FlashPro3
Opened 'C:\Actelprj\Test\designer\impl1\TopLevel fp\TopLevel.pro'
```

Ready   C:\Actelprj\Test\designer\impl1\TopLevel.pdb   SINGLE

Fig. 37: Flash Pro

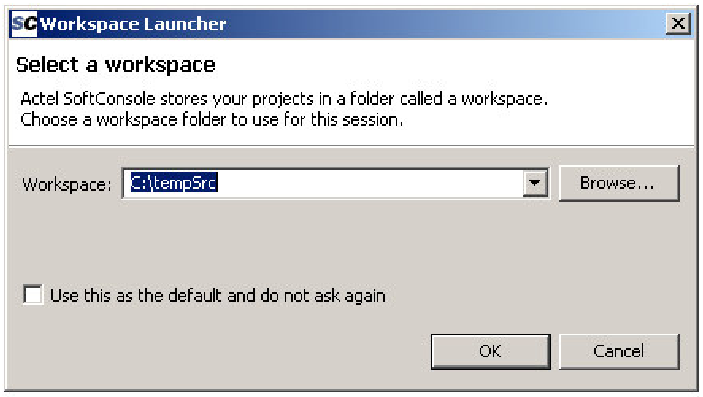
3 ACTEL SOFTCONSOLE

The IDE for programming the synthesized processor on the FPGA is based on Eclipse. Actel has created a plugin for Eclipse, dedicated to the ARM7 and the Cortex-M1 processor cores. This is installed at the same time as the rest of the software.

Fig. 38: Desktop icon of the Actel SoftConsole

3.1 Starting the IDE

When the IDE is started, the user will be asked to select a ‘workspace’, as shown in the following illustration.



Select a workspace

Actel SoftConsole stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\tempSrc

☐ Use this as the default and do not ask again

Fig. 39: Defining a workspace

The folder specified above could be configured to serve as the default folder, so that by subsequent starts of the IDE, it would not be necessary to specify it again.

3.2 Setting up a new C project

To start a new project one should select the menu option `Files / New / Managed Make C Projekt`, as shown in the illustration below.

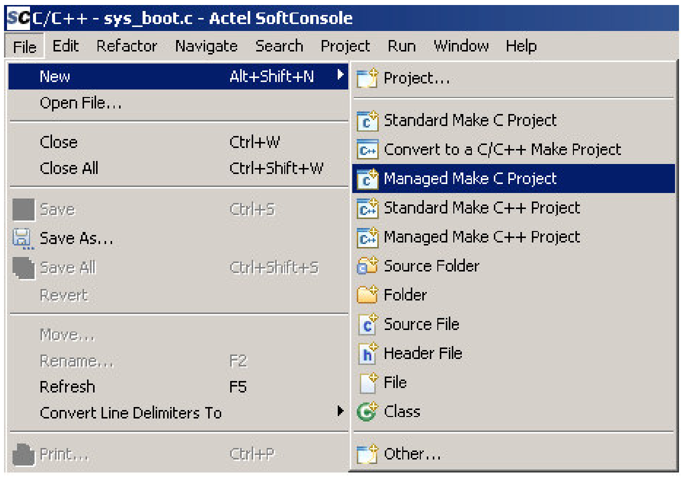
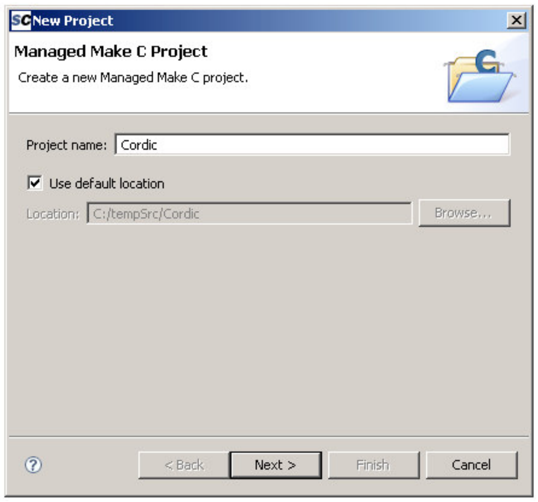


Fig. 40: Starting a new project

In the “New Project” dialogue box the project’s name was entered. The option to change the default folder also exists at this stage.



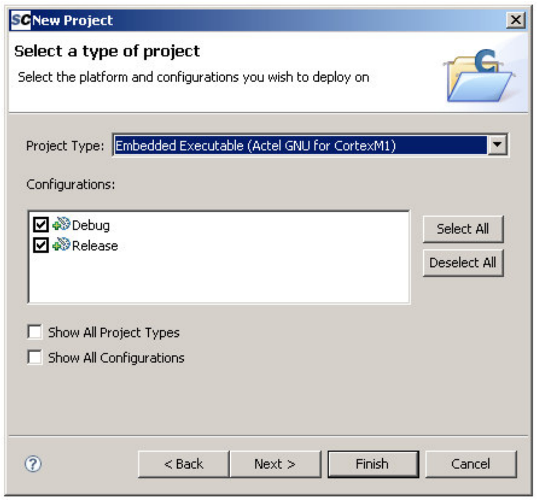
Project name: Cordic

☑ Use default location

Location: C:/tempSrc/Cordic

Fig. 41: Specifying the project’s name

The illustration below shows a fundamental setting for the project: the value of the field `Project Type` must be set to `Embedded Executable (Actel GNU for Cortex-M1)` to allow programming of the Cortex-M1 processor core. Making this choice activates important configuration features of the project. Finally, the dialogue box can be closed by clicking on the `Finish` button.



New Project

Select a type of project

Select the platform and configurations you wish to deploy on

Project Type: Embedded Executable (Actel GNU for CortexM1)

Configurations:

- [✓] Debug
- [✓] Release

- [ ] Show All Project Types
- [ ] Show All Configurations

Fig. 42: Selecting the project type

3.3 Importing project data

Once an empty project was created, it would normally be possible to either import data necessary for the configuration of the processor or to manually define such data. In many IDEs such configuration data is automatically imported once the processor has been specified. This is not the case with this project however, so that the user must perform all the tasks manually.

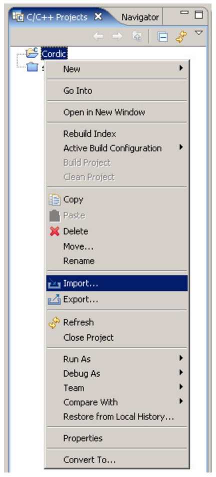
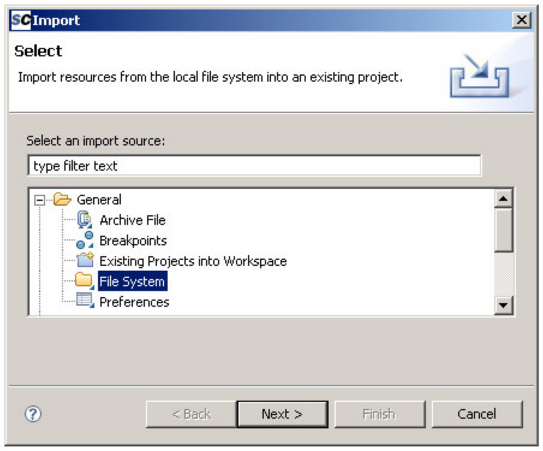


Fig. 43: Importing data

To manually import project data, it was necessary to right-click on the project name (in the “C/C++ Projects” window, as shown in Abb. 43) and, in the context menu that opened up, to select `Import...`; in the “Import” dialogue box (shown below) that appeared as a result, “File System” was selected before continuing by clicking on `Next`.


Import

Select

Import resources from the local file system into an existing project.

Select an import source:

type filter text

Fig. 44: Import dialogue

In the next dialogue, the `Browse...` was clicked and, in the file browser that opened up, one selected the folder containing the data that needed to be imported and confirmed with the `OK` button.

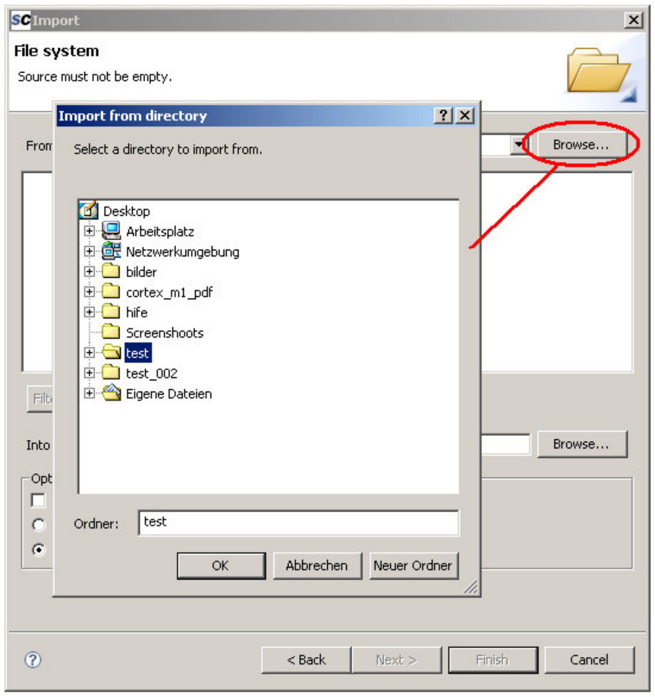
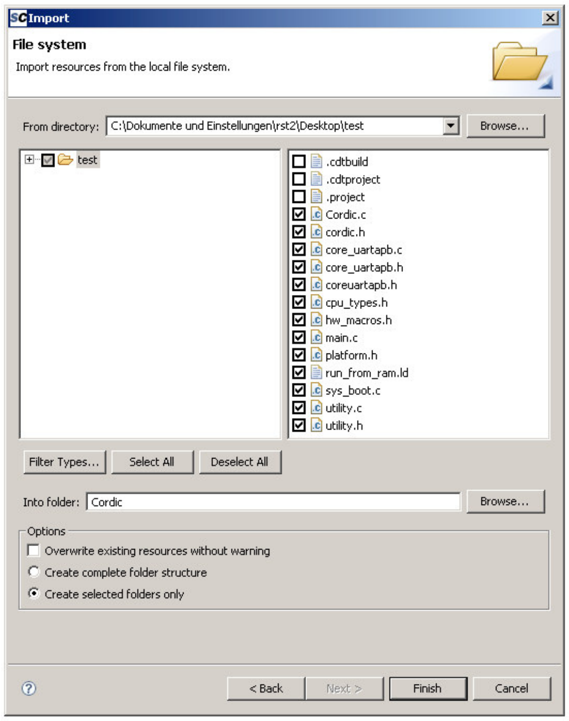


Fig. 45: Choosing the folder containing the data to be imported

All the data contained in the selected folder were now shown in the left pane of the dialogue box. Ticking the box next to the folder's name one selects the data to be imported. As it can be seen in the illustration below, the top three data files, listed in the right pane, are not selected, as they contain configuration data and properties of a project: they were already set up when an empty project was created, a little earlier. Importing those three files would result in the existing setup files being overwritten. Having made a choice, a click on the Finish button performs the data import.


Import

File system

Import resources from the local file system.

From directory: C:\Dokumente und Einstellungen\rst2\Desktop\test

[✓] test

[ ] .cdtbuild
[ ] .cdtproject
[ ] .project
[✓] Cordic.c
[✓] cordic.h
[✓] core_uartapb.c
[✓] core_uartapb.h
[✓] coreuartapb.h
[✓] cpu_types.h
[✓] hw_macros.h
[✓] main.c
[✓] platform.h
[✓] run_from_ram.ld
[✓] sys_boot.c
[✓] utility.c
[✓] utility.h

Into folder: Cordic

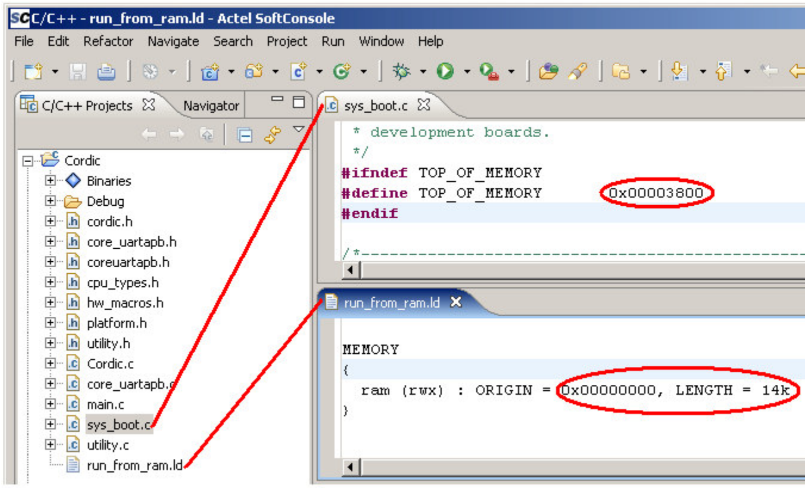
Options
[ ] Overwrite existing resources without warning
( ) Create complete folder structure
(●) Create selected folders only

Fig. 46: Select the data to be imported

3.4 Memory addressing

Once all the necessary data were included in the project, it was necessary to define the memory addressing details.

As the internal SRAM of the FPGA occupied the lowest ‘slot’ in the memory map, it would also be mapped at the memory offset of 0x00000000 in the linker script (run_from_ram.ld), as shown in the following illustration. The size of this memory was configured by CoreConsole to be 14 kByte, a figure which represents the maximum SRAM size possible for this FPGA.



```
 * development boards.
 */
#ifndef TOP_OF_MEMORY
#define TOP_OF_MEMORY       0x00003800
#endif
```

```
MEMORY
{
  ram (rwx) : ORIGIN = 0x00000000, LENGTH = 14k
}
```

Fig. 47: Configuring the memory

The stack pointer was configured to exist at the end of the SRAM address range and as it dynamically grows it moves towards address 0x00000000. This is represented by the “define TOP_OF_MEMORY” statement in the file “sys_boot.c”, where TOP_OF_MEMORY = memory offset + memory size.

3.5 Project properties

The linker script (run_from_ram.ld) must be included in the project’s features while the standard start up file must be deactivated. The project properties can be accessed from the context menu which appears when one right-clicks on the project. (s. Abb. 48).

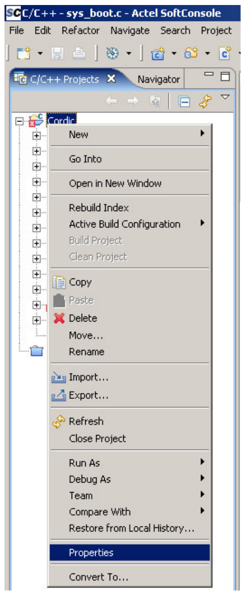
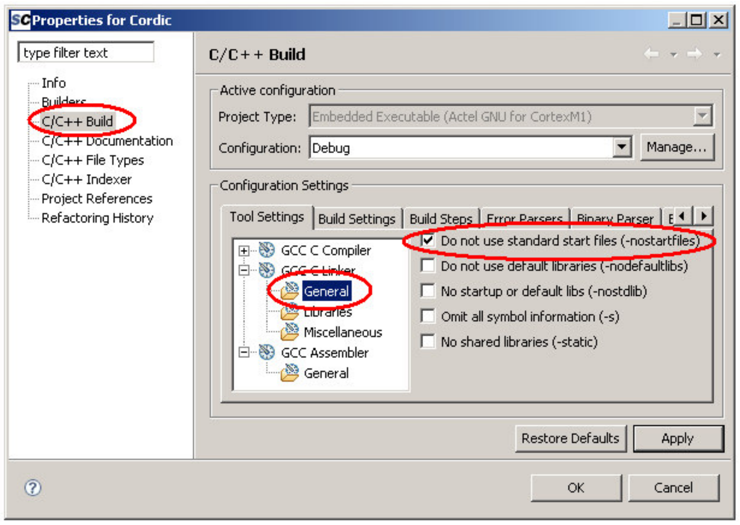


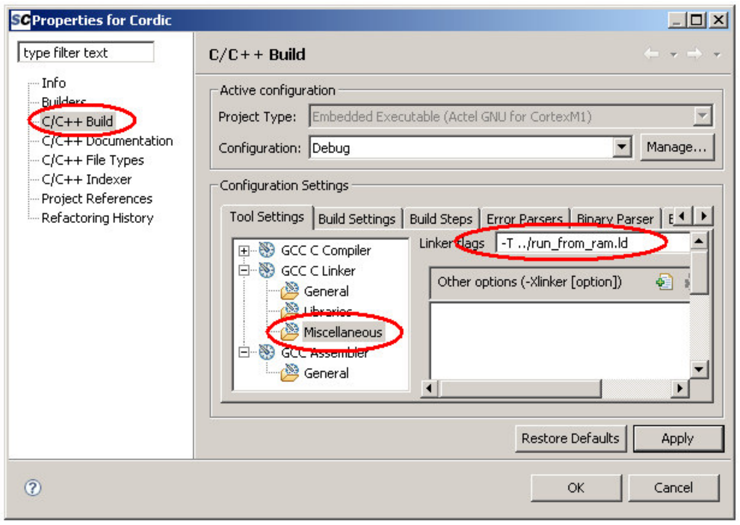
Fig. 48: Setting up the project properties

In the project's properties dialogue box, the following points were configured as shown in the following two illustrations (Abb. 49 and Abb. 50).



[✓] Do not use standard start files (-nostartfiles)
[ ] Do not use default libraries (-nodefaultlibs)
[ ] No startup or default libs (-nostdlib)
[ ] Omit all symbol information (-s)
[ ] No shared libraries (-static)

Fig. 49: Excluding any standard start up files



Linker flags: -T ../run_from_ram.ld

Fig. 50: Including the linker script

3.6 Compiling

Once everything had been configured properly, the project was compiled.. It is important to note, at this point, that any files which might still be open, should have been saved. Any files which have been modified after they were last saved, will be identified with a "*" to the left of their filename (s. Abb. 51)

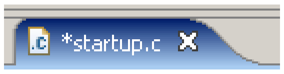
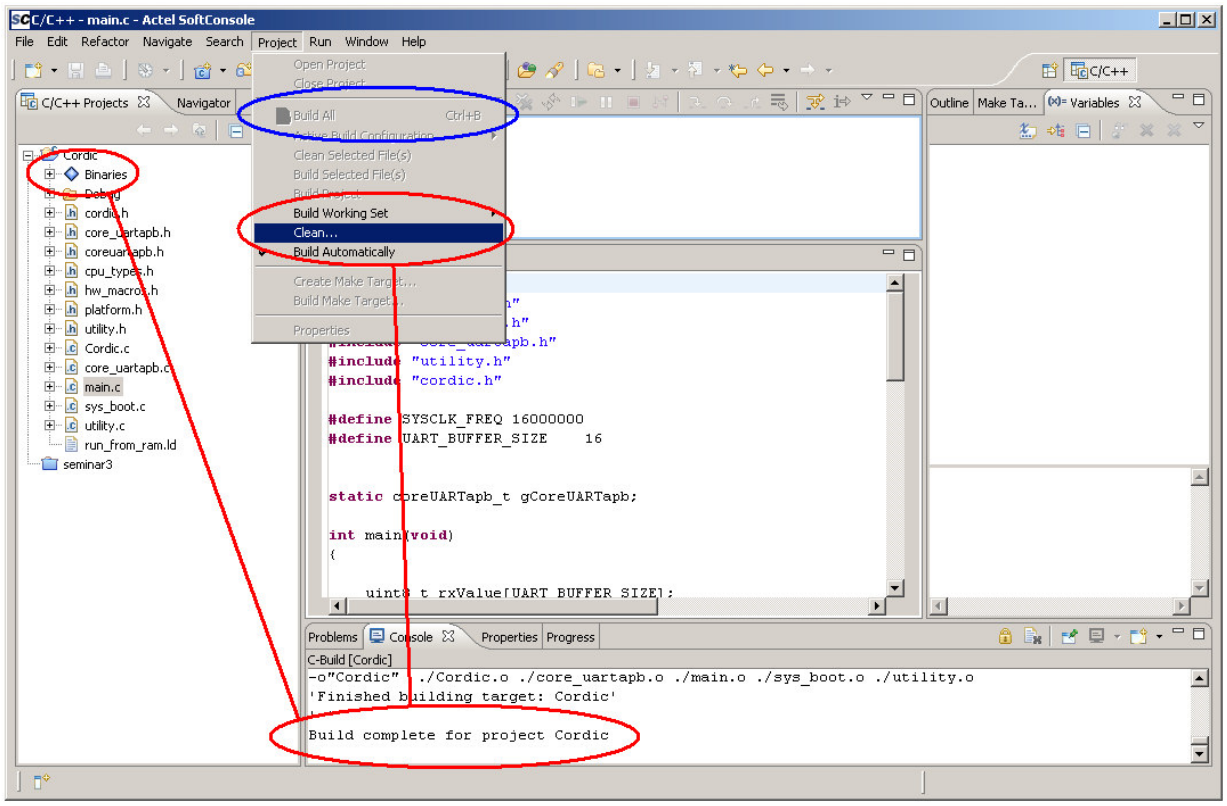


Fig. 51: Identification of a modified and not yet saved file

The compilation can be done either with the `Build All` option, where only modified files will be re-compiled, or with the `Clean...` option, where all object files will be deleted and generated again. The commands to perform the compilation can be found in menu "Project". All messages, generated during the compilation of the project, will appear in the console window.

At the end of a successful compilation, the message `Build complete for project xxx` (where "xxx" is the name of the project) will appear in the console pane while a new file, "Binaries", appears in the project pane. (s. Abb. 52).



```
#include "utility.h"
#include "cordic.h"

#define SYSCLK_FREQ 16000000
#define UART_BUFFER_SIZE    16


static coreUARTapb_t gCoreUARTapb;

int main(void)
{

    uint8_t rxValue[UART_BUFFER_SIZE];
```

```
-o"Cordic"  ./Cordic.o ./core_uartapb.o ./main.o ./sys_boot.o ./utility.o
'Finished building target: Cordic'
' '
Build complete for project Cordic
```

Fig. 52: Compiling the project

3.7 Setting up the programmer and the debugger

The programmer is implemented on the PCB. To configure it, one needs to select the option `External Tool...` in the toolbar (s. Abb. 53).

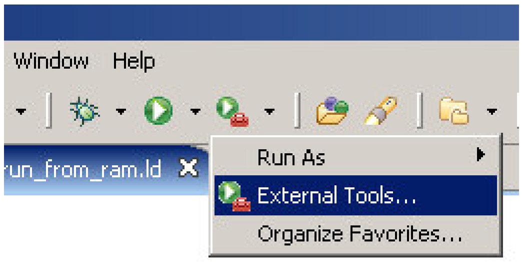
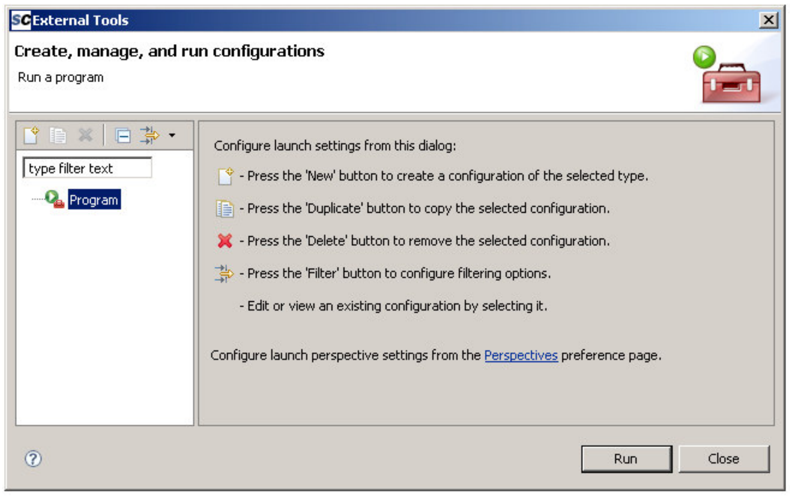


Fig. 53: Configuring the programmer (Step 1)

The following illustration shows how one can access the various configuration fields.



Create, manage, and run configurations

Run a program

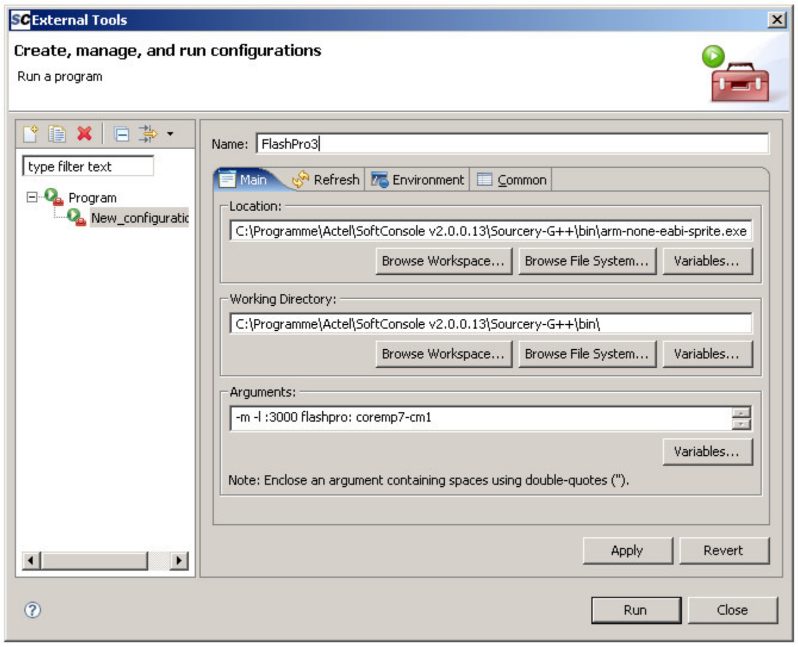
Configure launch settings from this dialog:

- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the Perspectives preference page.

Fig. 54: Configuring the programmer (Step 2)

The configuration settings for the Development Kit “M1A3P-DEV-KIT-SCS” are shown in the dialogue box in the following illustration. When the value of any one field in that dialogue is changed, it can be saved by clicking on the “`Apply`” button.



Name: FlashPro3

Location: C:\Programme\Actel\SoftConsole v2.0.0.13\Sourcery-G++\bin\arm-none-eabi-sprite.exe

Working Directory: C:\Programme\Actel\SoftConsole v2.0.0.13\Sourcery-G++\bin\

Arguments: -m -l :3000 flashpro: coremp7-cm1

Note: Enclose an argument containing spaces using double-quotes (").

Fig. 55: Configuring the programmer (Step 3)

Once the programmer is configured, it must also be included in the `Favorites` so that it can be invoked. The illustration below shows how this can be done.

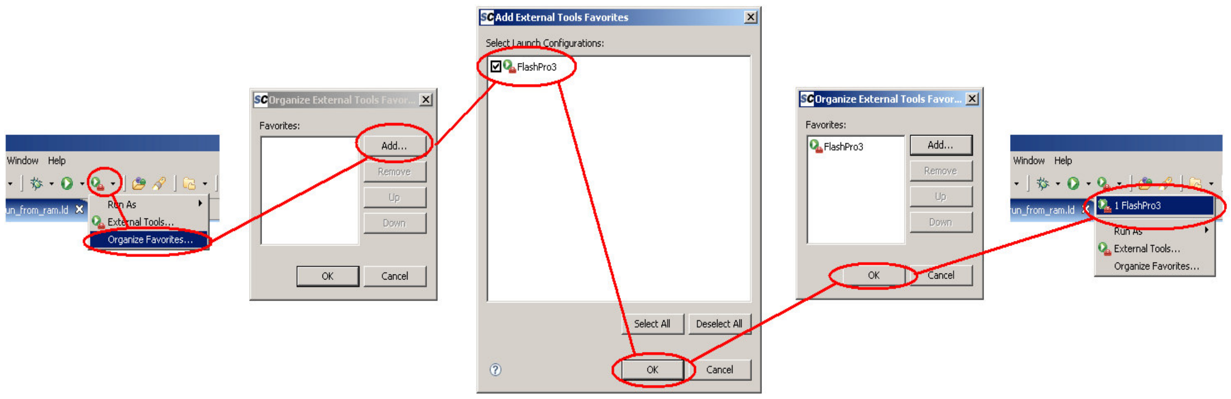


Fig. 56: Configuring the programmer (Step 4)

Configuring the Debugger is very similar to the configuration of the programmer. By selecting the option `Debug...`, in the menu bar, a dialogue opens up, as shown in the illustration below. The first thing that was done was to set up the options for `Embedded debug`. In the tab `Main` one has to select first the created project and then the compiled file “Binaries”.

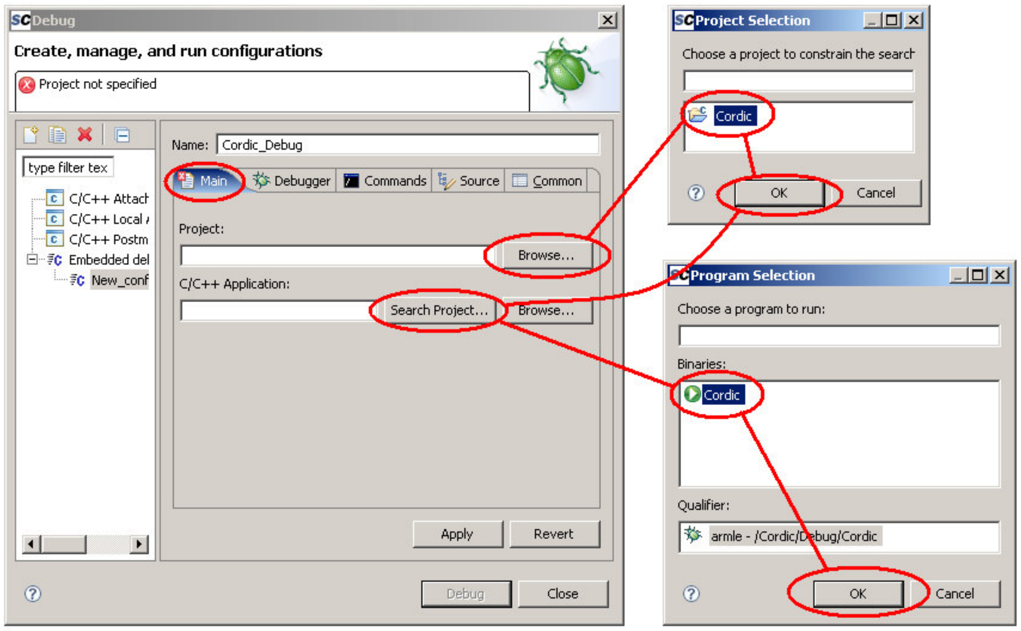
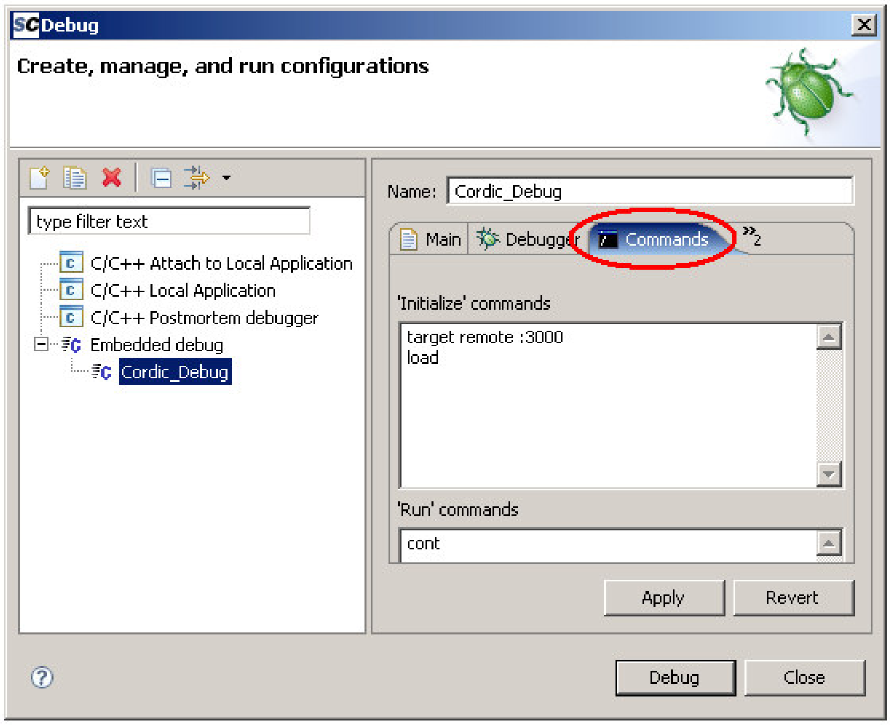


Fig. 57: Configuring the Debugger (Step 1)

Once the required file for the debugging was selected, the invocation parameters for the debugger were also specified. This was done by changing to the `Commands` tab and specifying the parameters as shown in the illustration below. The parameters were saved when the button `Apply` was clicked.



Name: Cordic_Debug

'Initialize' commands

```
target remote :3000
load
```

'Run' commands

```
cont
```

Fig. 58: Configuring the Debugger (Step 2)

These settings were then included in the `Favorites`, just like the programmer was, a little earlier. The method used for this was similar to that used for the programmer, described earlier in this document. The end result is shown in the following illustration.

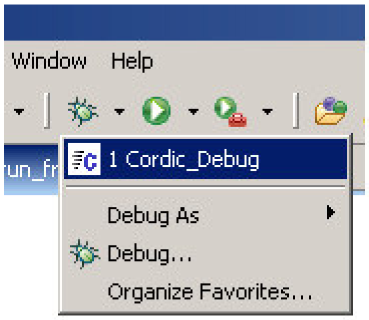


Fig. 59: Configuring the Debugger (Step 3)

3.8 Starting the programmer and the debugger

Once the source code has been successfully compiled and the programmer and debugger have been set up, the program could be downloaded and debugged.

The `Debug` window was opened first, as shown in the following illustration.

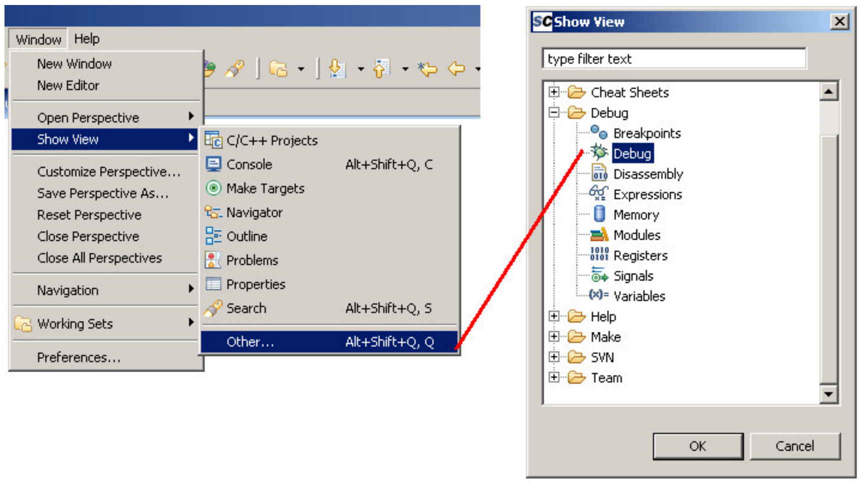
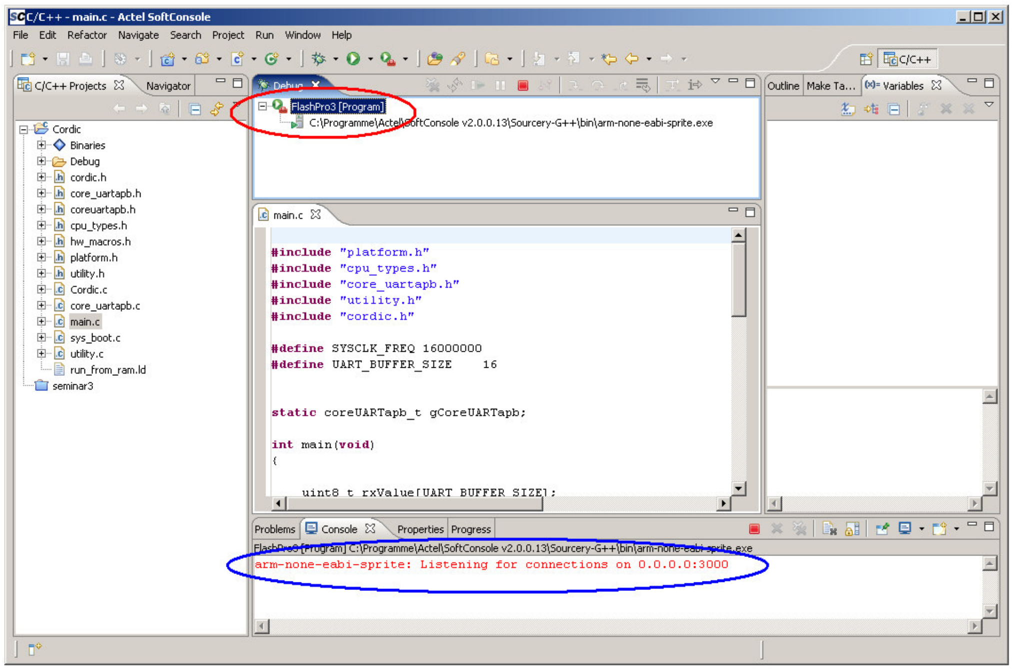


Fig. 60: Opening the Debug window

The connection to the programmer was then built, by clicking on the attached settings for the programmer (s. Abb. 56).



```
FlashPro3 [Program]
    C:\Programme\Actel\SoftConsole v2.0.0.13\Sourcery-G++\bin\arm-none-eabi-sprite.exe
```

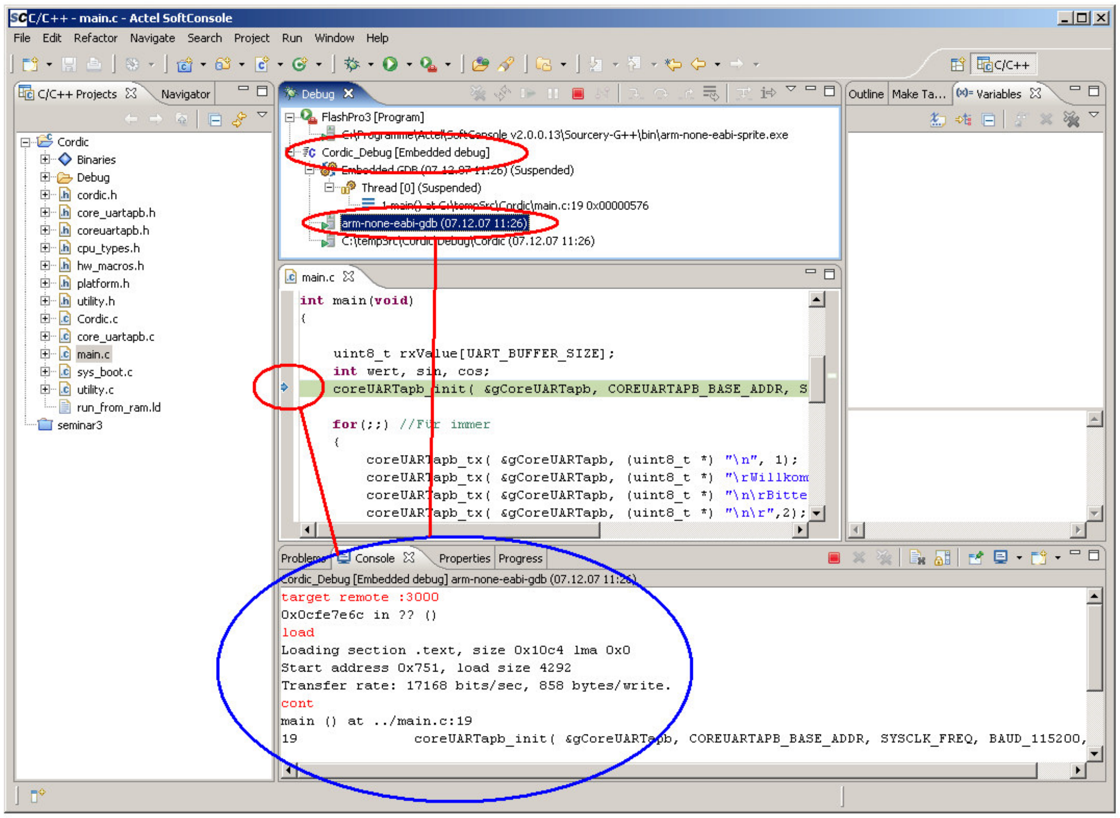
```
#include "platform.h"
#include "cpu_types.h"
#include "core_uartapb.h"
#include "utility.h"
#include "cordic.h"

#define SYSCLK_FREQ 16000000
#define UART_BUFFER_SIZE    16


static coreUARTapb_t gCoreUARTapb;

int main(void)
{

    uint8_t rxValue[UART_BUFFER_SIZE];
```

```
FlashPro3 [Program] C:\Programme\Actel\SoftConsole v2.0.0.13\Sourcery-G++\bin\arm-none-eabi-sprite.exe
arm-none-eabi-sprite: Listening for connections on 0.0.0.0:3000
```

Fig. 61: Starting the programmer

The Debug window shows the connection to the programmer. Once this connection has been built successfully, the display of the Console window would be as shown in the illustration above. The debugging can then be started by clicking on the Debug settings (s. Abb. 59).

When the Debugger was started, this was shown in the Debug window. Clicking on `arm-none-eabi-gdb...` in the Debug window, results in status information, concerning the connection to the Debugger and the downloads, appearing in the Console window. Once the program got loaded successfully, the Debugger 'jumped' at the first statement of the main function (s. Abb. 62).



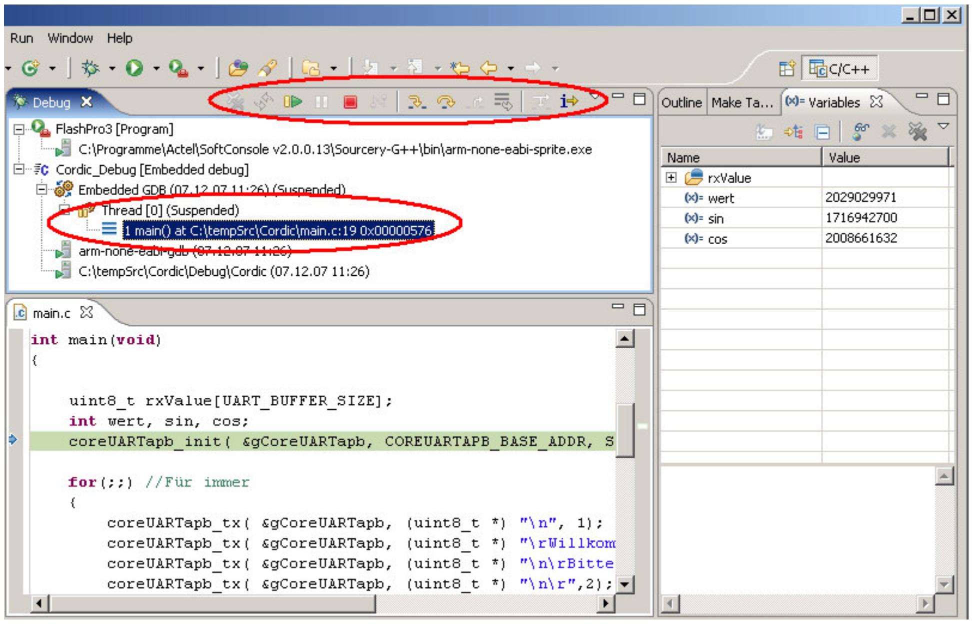
```
FlashPro3 [Program]
    C:\Programme\Actel\SoftConsole v2.0.0.13\Sourcery-G++\bin\arm-none-eabi-sprite.exe
Cordic_Debug [Embedded debug]
    Embedded GDB (07.12.07 11:26) (Suspended)
        Thread [0] (Suspended)
            1 main() at C:\tempSrc\Cordic\main.c:19 0x00000576
    arm-none-eabi-gdb (07.12.07 11:26)
    C:\tempSrc\Cordic\Debug\Cordic (07.12.07 11:26)
```

```
int main(void)
{

    uint8_t rxValue[UART_BUFFER_SIZE];
    int wert, sin, cos;
    coreUARTapb_init( &gCoreUARTapb, COREUARTAPB_BASE_ADDR, S

    for(;;) //Für immer
    {
        coreUARTapb_tx( &gCoreUARTapb, (uint8_t *) "\n", 1);
        coreUARTapb_tx( &gCoreUARTapb, (uint8_t *) "\rWillkom
        coreUARTapb_tx( &gCoreUARTapb, (uint8_t *) "\n\rBitte
        coreUARTapb_tx( &gCoreUARTapb, (uint8_t *) "\n\r",2);
```

```
Cordic_Debug [Embedded debug] arm-none-eabi-gdb (07.12.07 11:26)
target remote :3000
0x0cfe7e6c in ?? ()
load
Loading section .text, size 0x10c4 lma 0x0
Start address 0x751, load size 4292
Transfer rate: 17168 bits/sec, 858 bytes/write.
cont
main () at ../main.c:19
19              coreUARTapb_init( &gCoreUARTapb, COREUARTAPB_BASE_ADDR, SYSCLK_FREQ, BAUD_115200,
```

Fig. 62: The debugger started

When the ‘main’ function was selected in the Debug window, as shown in the illustration below, the buttons for the debugging were activated.



```
FlashPro3 [Program]
    C:\Programme\Actel\SoftConsole v2.0.0.13\Sourcery-G++\bin\arm-none-eabi-sprite.exe
Cordic_Debug [Embedded debug]
    Embedded GDB (07.12.07 11:26) (Suspended)
        Thread [0] (Suspended)
            1 main() at C:\tempSrc\Cordic\main.c:19 0x00000576
    arm-none-eabi-gdb (07.12.07 11:26)
    C:\tempSrc\Cordic\Debug\Cordic (07.12.07 11:26)
```

```
int main(void)
{

    uint8_t rxValue[UART_BUFFER_SIZE];
    int wert, sin, cos;
    coreUARTapb_init( &gCoreUARTapb, COREUARTAPB_BASE_ADDR, S

    for(;;) //Für immer
    {
        coreUARTapb_tx( &gCoreUARTapb, (uint8_t *) "\n", 1);
        coreUARTapb_tx( &gCoreUARTapb, (uint8_t *) "\rWillkom
        coreUARTapb_tx( &gCoreUARTapb, (uint8_t *) "\n\rBitte
        coreUARTapb_tx( &gCoreUARTapb, (uint8_t *) "\n\r",2);
```

Name	Value
rxValue	
wert	2029029971
sin	1716942700
cos	2008661632

Fig. 63: Debugging

4 DESCRIPTION OF THE PROGRAM

In order to test the hardware, a Cordic algorithm was implemented. This algorithm calculates the sine and cosine functions of a given number. Such a number was entered over the UART interface through a Hyper-Terminal or some similar program on a PC.

The first action of the main program was to initialise the UART. Then some text was output over the UART, following which the program would wait for input from the user. Such input would first be checked for any errors, before being formatted as a fixed point number, with 8 bits before and 24 bits after the decimal point, and passed on to the function that implemented the Cordic algorithm. The result of the calculation were transformed back into strings and output over the UART.

BIBLIOGRAPHY

www.actel.com

TABLE OF FIGURES