# HB0085
# Handbook
# CoreABC v3.7

Microsemi
Power Matters.

**Microsemi Corporate Headquarters**
One Enterprise, Aliso Viejo,
CA 92656 USA
Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Fax: +1 (949) 215-4996
Email: sales.support@microsemi.com
www.microsemi.com





## About Microsemi

Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for aerospace & defense, communications, data center and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; enterprise storage and communication solutions, security technologies and scalable anti-tamper products; Ethernet solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, California, and has approximately 4,800 employees globally. Learn more at www.microsemi.com.

50200085. 11.0 12/16

# Contents

# Figures

# Tables

# 1 Revision History

The revision history describes the changes that were implemented in the document. The changes are listed by revision, starting with the most current publication.

## 1.1 Revision 11.0

Updated changes related to CoreABC v3.7 release.

## 1.2 Revision 10.0

Updated changes related to CoreABC v3.6 release.

## 1.3 Revision 9.0

Updated changes related to CoreABC v3.5 release.

## 1.4 Revision 8.0

Updated changes related to CoreABC v3.4 release.

## 1.5 Revision 7.0

The following is a summary of the changes in revision 7.0 of this document.

- Added "Constant Expressions" and "Conditional Code" section
- Added Figure 5

## 1.6 Revision 6.0

Updated changes related to CoreABC v3.3 release.

## 1.7 Revision 5.0

Updated changes related to CoreABC v3.1 release.

## 1.8 Revision 4.0

Updated changes related to CoreABC v3.0 release.

## 1.9 Revision 3.0

The following is a summary of the changes in revision 3.0 of this document.

- The "Supported Device Families" section was added.
- A note regarding frequency of IGLOO devices was added to Table 1 and Table 2.
- The “Fusion, ProASIC3/E, ProASIC3L, Axcelerator, and RTAX-S Families” section was updated to include ProASIC3L.

## 1.10 Revision 2.0

The following is a summary of the changes in revision 2.0 of this document.

- Supported core version updated in "Core Version" section.
- Supported version of Libero IDE updated in “Supported Tool Flows” section.
- The LOADLOOP register was renamed Z Register. The LOADZ condition flag was renamed ZZERO.
- Table 1 replaced and Table 2 created.
- "Utilization and Performance" section updated.
- Figure 2 updated.
- EQ 1-2 and EQ 1-4 updated.
- Figure 8 updated.
- The "Simulation Flows" section was updated.
- Table 3 updated, with numerous parameter changes, additions, and deletions.
- THe "EN_DATAM Parameter" section was added.
- "Internal Data RAM Address Space (optional)" and "I/O Address Space" sections updated.
- The “Instruction Set” section was replaced.
- CoreABC Instruction Encoding updated variously.
- "Simulation Logging" section updated.
- Figure 6 and Figure 8 were updated.
- "Number of I/O Inputs" section added and "Number of I/O Flag Inputs" section modified.
- "ALU Operation from Memory", "APB Indirect Addressing", and "Supported Data Sources" sections added.
- Figure 10 and Figure 11 were updated.
- "Verification Tests" section updated.
- “Example Instruction Sequence” section modified.
- Many instructions were added or changed in the "Instruction Summary" section.
- The “Core Version” and "Supported Interfaces" sections are new.
- Values in the Configuration column were updated in Table 1.
- The last paragraph was changed in the "ACM Lookup Table for Use with CoreAI" section.
- The "Automatically Created Memory Image Files" section is new.
- The "Updating the Program and Flash Memory Contents" section is new.
- The "Instruction Summary" section is new.

## 1.11 Revision 1.0

Revision 1.0 was the first publication of this document.

# 2 Introduction

## 2.1 CoreABC Overview

CoreABC (ABC = APB bus controller) is a simple, configurable, low gate count, programmable state machine/controller primarily targeted toward the implementation of Advanced Microcontroller Bus Architecture (AMBA®) Advanced Peripheral Bus (APB) based designs. It is particularly suitable in the following situations:

- A programmable controller is required but a full featured CPU such as Core8051s or ARM® Cortex®-M1 is not needed or cannot be justified due to cost or resource/size constraints.
- A full featured CPU based system requires a CoreABC based programmable offload engine/coprocessor subsystem for performance reasons.
- A Fusion AFS system using CoreAI or CorePWM, for example, requires programmable control either as a standalone design or as a Fusion AFS analog offload engine/coprocessor for a larger CPU based system.

CoreABC supports a comprehensive assembler based configurable instruction set architecture and extensive and flexible configuration of size and feature options, allowing it to be tuned to meet the resource constraints and processing power requirements of a wide variety of applications.

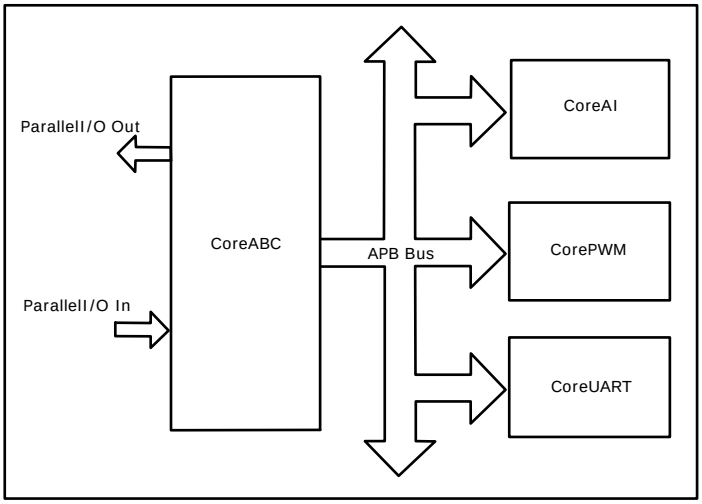
CoreABC supports three program storage modes:

- **Hard mode:** Program image is stored in an internal ROM implemented in FPGA fabric tiles
- **Soft mode:** Program image is stored in Microsemi FPGA RAM blocks which are initialized at runtime from the binary image stored in Fusion AFS NVM or an external flash memory
- **NVM mode** (Fusion AFS only): Program image stored in and executed directly from Fusion AFS NVM

CoreABC is available through the Libero® Integrated Design Environment (IDE) IP Catalog, through which it can be downloaded from a remote web-based repository and installed into the user's local vault, ready for use. It operates natively within the SmartDesign design entry environment, allowing it to be easily instantiated, configured, and connected to other IP core instances and generated ready for simulation, synthesis, etc. CoreABC is an AMBA3 APB master which can connect to and manage any APB slave peripherals via an AMBA3 APB bus fabric component such as CoreAPB3.

Figure 1 shows a CoreABC based system that can monitor analog inputs, adjust output levels, and report status via an RS-232 link using CoreUART.

***Figure 1 •*** **Typical CoreABC System**



## 2.2 Supported Device Families

PolarFire

RTG4™

IGLOO®2

IGLOO

IGLOOe

IGLOO PLUS

ProASIC3

ProASIC3E

ProASIC®3L

SmartFusion®2

SmartFusion™

Fusion

ProASIC PLUS®

Axcelerator®

RTAX-S

## 2.3 Core Version

This handbook supports CoreABC v3.7.

## 2.4 Supported Interfaces

CoreABC has an AMBA3 APB master interface, which is described in the section Interface, page 10. When configured in NVM mode, an additional AMBA3 APB slave interface is available for accessing the NVM block used to store instructions within CoreABC. APB access to the instruction NVM block may be used, for example, to maintain a nonvolatile log of data values in cases where only one NVM block is available for CoreABC's use.

When configured in soft mode, an initialization and configuration (InitCfg) interface is used for initializing the RAM blocks used for CoreABC's instruction memory.

## 2.5 Supported Tool Flows

CoreABC requires Libero IDE v8.6 SP1 or later. Additionally, Verilog users MUST use Synplicity® v8.6.1 or later, which is downloadable from www.synplicity.com.

## 2.6 Utilization and Performance

CoreABC utilization varies depending on how it is configured Table 1 below and Table 2 provide typical utilization data for a range of devices and data widths. The other configuration options for the core are collectively grouped to give three different CoreABC configurations named small, medium, and large; these configurations are listed in Table 3. CoreABC can be implemented in several Microsemi FPGA devices.

***Table 1 •*** **CoreABC Utilization Data (Hard Mode—instructions held in tiles)**

| Family | Data Width | Config. | Comb. | Seq. | RAM | Total | Device | Utilization | Frequency MHz* |
|---|---|---|---|---|---|---|---|---|---|
| Fusion ProASIC®3/E IGLOO™/e | 8 | Small | 179 | 46 | 0 | 225 | AFS600 A3P600 AGL600 | 1.6% | 92 |
| ProASICPLUS | 8 | Small | 195 | 51 | 0 | 246 | APA450 | 2.0% | 81 |
| Axcelerator® RTAX-S | 8 | Small | 96 | 45 | 0 | 141 | AX250 RTAX250 | 3.3% | 123 |
| Fusion ProASIC3/E IGLOO/e | 16 | Small | 238 | 59 | 0 | 297 | AFS600 A3P600 AGL600 | 2.1% | 79 |
| ProASICPLUS | 16 | Small | 269 | 63 | 0 | 332 | APA450 | 2.7% | 79 |
| Axcelerator RTAX-S | 16 | Small | 127 | 57 | 0 | 184 | AX250 RTAX250 | 4.4% | 98 |
| Fusion ProASIC3/E IGLOO/e | 32 | Small | 319 | 74 | 0 | 393 | AFS600 A3P600 AGL600 | 2.8% | 58 |
| ProASICPLUS | 32 | Small | 381 | 84 | 0 | 465 | APA450 | 3.9% | 60 |
| Axcelerator RTAX-S | 32 | Small | 192 | 78 | 0 | 270 | AX250 RTAX250 | 6.4% | 97 |
| PolarFire | 8 | Small | 61 | 40 | 0 | 101 | MPF300TS | 0.03% | 200 |
| PolarFire | 16 | Small | 83 | 48 | 0 | 131 | MPF300TS | 0.05% | 200 |
| PolarFire | 32 | Small | 120 | 65 | 0 | 185 | MPF300TS | 0.06% | 200 |
| RTG4 | 8 | Small | 71 | 38 | 0 | 101 | RT4G150 | 0.08% | 100 |
| RTG4 | 16 | Small | 93 | 46 | 0 | 139 | RT4G150 | 0.09% | 100 |
| RTG4 | 32 | Small | 130 | 63 | 0 | 193 | RT4G150 | 0.13% | 100 |
| IGLOO2 /SmartFusion2 | 8 | Small | 61 | 40 | 0 | 101 | M2S150TS | 0.07% | 150 |
| IGLOO2/ SmartFusion2 | 16 | Small | 83 | 48 | 0 | 131 | M2S150TS | 0.09% | 150 |
| IGLOO2/ SmartFusion2 | 32 | Small | 120 | 65 | 0 | 185 | M2S150TS | 0.12% | 150 |
| Fusion ProASIC3/E IGLOO/e | 8 | Medium | 363 | 76 | 1 | 439 | AFS600 A3P600 AGL600 | 3.2% | 55 |
| ProASICPLUS | 8 | Medium | 439 | 88 | 1 | 527 | APA450 | 4.3% | 41 |
| Axcelerator RTAX-S | 8 | Medium | 229 | 76 | 1 | 305 | AX250 RTAX250 | 7.2% | 86 |
| Fusion ProASIC3/E IGLOO/e | 16 | Medium | 558 | 88 | 1 | 646 | AFS600 A3P600 AGL600 | 4.7% | 41 |
| ProASICPLUS | 16 | Medium | 630 | 95 | 2 | 725 | APA450 | 5.9% | 32 |
| Axcelerator RTAX-S | 16 | Medium | 307 | 92 | 1 | 399 | AX250 RTAX250 | 9.4% | 73 |
| Fusion ProASIC3/E IGLOO/e | 32 | Medium | 896 | 104 | 2 | 1,000 | AFS600 A3P600 AGL600 | 7.2% | 37 |
| ProASICPLUS | 32 | Medium | 947 | 112 | 4 | 1,059 | APA450 | 8.6% | 28 |
| Axcelerator RTAX-S | 32 | Medium | 442 | 108 | 2 | 550 | AX250 RTAX250 | 13.0% | 64 |
| PolarFire | 8 | Medium | 578 | 148 | 2 | 726 | MPF300TS | 0.24% | 130 |

***Table 1 •*** **CoreABC Utilization Data (Hard Mode—instructions held in tiles)**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| PolarFire | 16 | Medium | 770 | 234 | 4 | 1,004 | MPF300TS | 0.34% | 130 |
| PolarFire | 32 | Medium | 1,208 | 394 | 8 | 1,602 | MPF300TS | 0.53% | 120 |
| IGLOO2/ SmartFusion2 | 8 | Medium | 471 | 146 | 2 | 617 | M2S150TS | 0.42% | 125 |
| IGLOO2/ SmartFusion2 | 16 | Medium | 685 | 232 | 4 | 917 | M2S150TS | 0.63% | 125 |
| IGLOO2/ SmartFusion2 | 32 | Medium | 1,429 | 396 | 8 | 1825 | M2S150TS | 1.25% | 100 |
| RTG4 | 8 | Medium | 492 | 144 | 2 | 636 | RT4G150 | 0.41% | 88 |
| RTG4 | 16 | Medium | 652 | 227 | 4 | 879 | RT4G150 | 0.58% | 84 |
| RTG4 | 32 | Medium | 785 | 171 | 2 | 956 | RT4G150 | 0.63% | 74 |
| Fusion ProASIC3/E IGLOO/e | 8 | Large | 474 | 82 | 1 | 556 | AFS600 A3P600 AGL600 | 4.0% | 42 |
| ProASICPLUS | 8 | Large | 565 | 94 | 1 | 659 | APA450 | 5.4% | 38 |
| Axcelerator RTAX-S | 8 | Large | 291 | 86 | 1 | 377 | AX250 RTAX250 | 8.9% | 71 |
| Fusion ProASIC3/E IGLOO/e | 16 | Large | 648 | 94 | 1 | 742 | AFS600 A3P600 AGL600 | 5.4% | 27 |
| ProASICPLUS | 16 | Large | 763 | 105 | 2 | 868 | APA450 | 7.1% | 24 |
| Axcelerator RTAX-S | 16 | Large | 399 | 98 | 1 | 497 | AX250 RTAX250 | 11.8% | 69 |
| Fusion ProASIC3/E IGLOO/e | 32 | Large | 1,014 | 111 | 2 | 1,125 | AFS600 A3P600 AGL600 | 8.1% | 34 |
| ProASICPLUS | 32 | Large | 1,082 | 126 | 4 | 1,208 | APA450 | 9.8% | 18 |
| Axcelerator RTAX-S | 32 | Large | 586 | 119 | 2 | 705 | AX250 RTAX250 | 16.7% | 53 |
| PolarFire | 8 | Large | 692 | 227 | 4 | 919 | MPF300TS | 0.31% | 130 |
| PolarFire | 16 | Large | 710 | 229 | 4 | 939 | MPF300TS | 0.32% | 130 |
| PolarFire | 32 | Large | 1,427 | 397 | 8 | 1824 | MPF300TS | 0.61% | 110 |
| SmartFusion2/ IGLOO2 | 8 | Large | 532 | 220 | 4 | 752 | M2S150TS | 0.51% | 125 |
| SmartFusion2/ IGLOO2 | 16 | Large | 641 | 230 | 4 | 871 | M2S150TS | 0.6% | 125 |
| SmartFusion2 | 32 | Large | 1,400 | 395 | 8 | 1795 | M2S150TS | 1.23% | 105 |
| RTG4 | 8 | Large | 482 | 146 | 2 | 628 | RT4G150 | 0.42% | 88 |
| RTG4 | 16 | Large | 692 | 227 | 4 | 919 | RT4G150 | 0.61% | 81 |
| RTG4 | 32 | Large | 1,138 | 173 | 2 | 1311 | RT4G150 | 0.86% | 64 |

**Note:** The frequency given in the table does not apply to the IGLOO devices. IGLOO family devices will run significantly slower than the speed listed in the table.

***Table 2 •*** **CoreABC Utilization Data (Soft Mode—instructions held in RAM)**

| Family | Data Width | Config. | Comb. | Seq. | RAM | Total | Device | Utilization | Frequency MHz* |
|---|---|---|---|---|---|---|---|---|---|
| Fusion ProASIC3/E IGLOO/e | 8 | Small | 126 | 27 | 3 | 153 | AFS600 A3P600 AGL600 | 1.1% | 68 |
| ProASICPLUS | 8 | Small | 137 | 30 | 6 | 167 | APA450 | 1.4% | 53 |

***Table 2 •*** **CoreABC Utilization Data (Soft Mode—instructions held in RAM)**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Axcelerator RTAX-S | 8 | Small | 61 | 27 | 3 | 88 | AX250 RTAX250 | 2.1% | 95 |
| Fusion ProASIC3/E IGLOO/e | 16 | Small | 179 | 36 | 4 | 215 | AFS600 A3P600 AGL600 | 1.6% | 67 |
| ProASICPLUS | 16 | Small | 213 | 41 | 8 | 254 | APA450 | 2.1% | 50 |
| Axcelerator RTAX-S | 16 | Small | 94 | 35 | 4 | 129 | AX250 RTAX250 | 3.1% | 84 |
| Fusion ProASIC3/E IGLOO/e | 32 | Small | 353 | 53 | 5 | 406 | AFS600 A3P600 AGL600 | 2.9% | 46 |
| ProASICPLUS | 32 | Small | 359 | 58 | 10 | 417 | APA450 | 3.4% | 42 |
| Axcelerator RTAX-S | 32 | Small | 155 | 52 | 5 | 207 | AX250 RTAX250 | 4.9% | 65 |
| Fusion ProASIC3/E IGLOO/e | 8 | Medium | 326 | 56 | 4 | 382 | AFS600 A3P600 AGL600 | 2.8% | 46 |
| ProASICPLUS | 8 | Medium | 409 | 59 | 7 | 468 | APA450 | 3.8% | 34 |
| Axcelerator RTAX-S | 8 | Medium | 210 | 55 | 4 | 265 | AX250 RTAX250 | 6.3% | 59 |
| Fusion ProASIC3/E IGLOO/e | 16 | Medium | 548 | 64 | 5 | 612 | AFS600 A3P600 AGL600 | 4.4% | 40 |
| ProASICPLUS | 16 | Medium | 659 | 73 | 10 | 732 | APA450 | 6.0% | 28 |
| Axcelerator RTAX-S | 16 | Medium | 271 | 64 | 5 | 335 | AX250 RTAX250 | 7.9% | 58 |
| Fusion ProASIC3/E IGLOO/e | 32 | Medium | 851 | 80 | 8 | 931 | AFS600 A3P600 AGL600 | 6.7% | 32 |
| ProASICPLUS | 32 | Medium | 892 | 96 | 16 | 988 | APA450 | 8.0% | 26 |
| Axcelerator RTAX-S | 32 | Medium | 399 | 80 | 8 | 479 | AX250 RTAX250 | 11.3% | 50 |
| Fusion ProASIC3/E IGLOO/e | 8 | Large | 462 | 62 | 5 | 524 | AFS600 A3P600 AGL600 | 3.8% | 40 |
| ProASICPLUS | 8 | Large | 534 | 67 | 9 | 601 | APA450 | 4.9% | 31 |
| Axcelerator RTAX-S | 8 | Large | 282 | 61 | 5 | 343 | AX250 RTAX250 | 8.1% | 63 |
| Fusion ProASIC3/E IGLOO/e | 16 | Large | 626 | 71 | 6 | 697 | AFS600 A3P600 AGL600 | 5.0% | 25 |
| ProASICPLUS | 16 | Large | 732 | 83 | 12 | 815 | APA450 | 6.6% | 21 |
| Axcelerator RTAX-S | 16 | Large | 380 | 70 | 6 | 450 | AX250 RTAX250 | 10.7% | 56 |

***Table 2 •*** **CoreABC Utilization Data (Soft Mode—instructions held in RAM)**

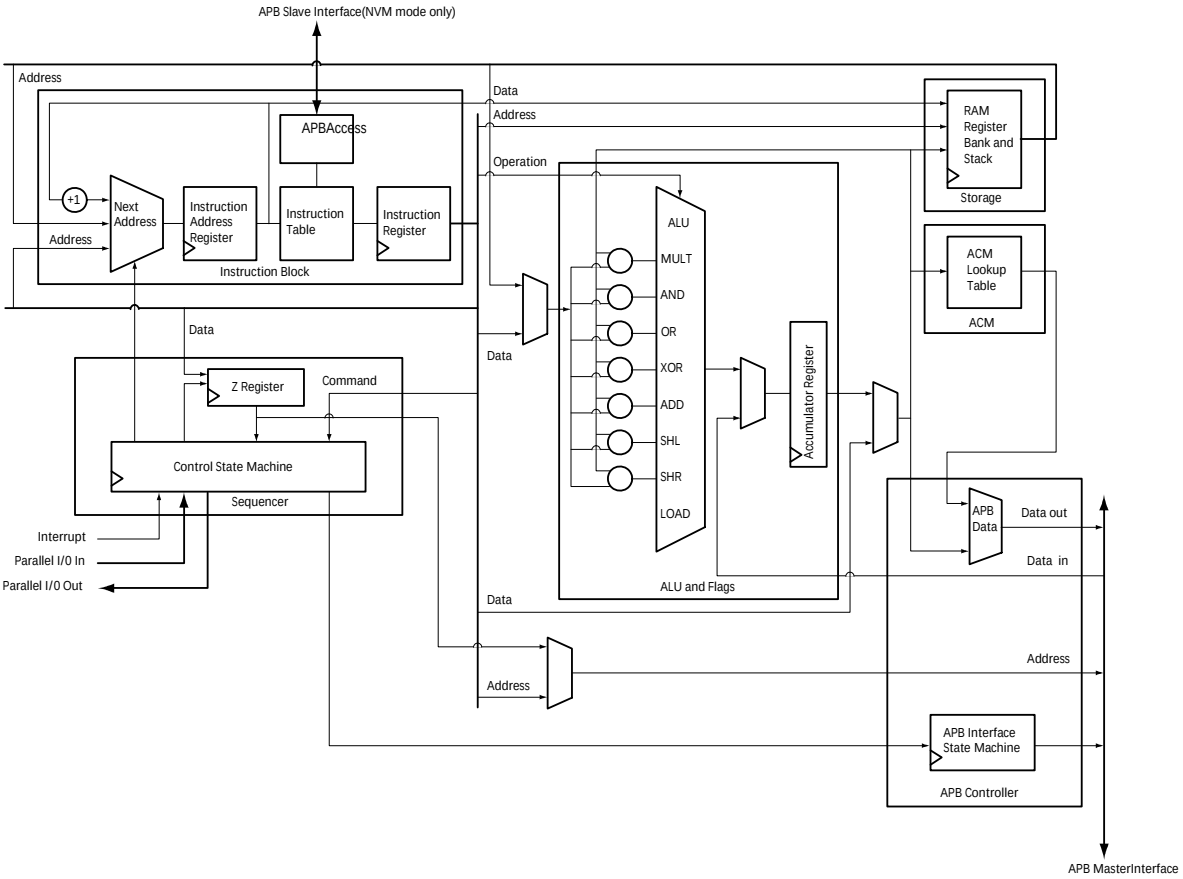
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Fusion ProASIC3/E IGLOO/e | 32 | Large | 1,053 | 86 | 8 | 1,139 | AFS600 A3P600 AGL600 | 8.2% | 34 |
| ProASICPLUS | 32 | Large | 1,228 | 106 | 16 | 1,334 | APA450 | 10.9% | 27 |
| Axcelerator RTAX-S | 32 | Large | 579 | 85 | 8 | 664 | AX250 RTAX250 | 15.7% | 46 |
| **Note:** *The frequency given in the table does not apply to the IGLOO devices. IGLOO family devices will run significantly slower than the speed listed in the table. | | | | | | | | | |

# 3 Functional Description

CoreABC internal architecture is shown in Figure 2, page 9. The core consists of six main blocks:

- Instruction block
- Sequencer
- ALU and Flags
- Storage
- Analog configuration MUX (ACM)
- APB controller

***Figure 2 •*** **CoreABC Block Diagram**




The Instruction block contains the instruction counter and the instruction table that contains the instructions to be executed. In soft mode, these instructions are fetched from RAM internal to CoreABC.

The ALU and Flags block implements the main ALU block. Each of the supported operations can be disabled to obtain a minimal-gate-count solution. The Storage block provides local storage for data values and implements the stack required by the call instruction.

The ACM block implements a small lookup table that can be initialized with the configuration data required by CoreAI. This allows the analog functions within a Fusion AFS FPGA to be easily configured.

# 4 Interface

## 4.1 Overview of Interfaces

CoreABC has an AMBA3 APB master interface which typically will be connected to CoreAPB3. When in NVM mode (INSMODE parameter set to 2), an additional AMBA3 APB slave interface is available for data type access to the (NVM) instruction store.

**Note:** When CoreABC is mastering CoreAPB3, the APB Slot Size configuration option settings should match for both of these cores.

In soft mode (INSMODE parameter set to 1), an initialization and configuration (InitCfg) interface is available for initializing the RAM blocks used as CoreABC's instruction store. This interface is intended to be used to connect to a Flash Memory System Builder (FMSB) RAM Initialization client. The use of FMSB clients is supported only on Fusion AFS devices. On other device families, a different means must be employed to initialize the instruction RAM blocks through the InitCfg interface. This could involve implementing some logic to allow another processor in the system to communicate with the InitCfg interface.

In addition to the interfaces already mentioned, CoreABC has clock, reset, and interrupt related signals as well as general purpose parallel input and output buses. The widths of these input and output buses are configurable.

## 4.2 Parameters

The parameters described are those directly in the RTL. When working with CoreABC in the SmartDesign tool, a configuration GUI is available for setting these parameters. The recommended configuration flow is to use the configuration GUI in SmartDesign, which will then set these parameters correctly. Importantly, when using the configuration GUI, the parameter settings will be cross checked with the CoreABC program (which is entered in another tab of the configuration GUI). The configuration GUI will indicate any inconsistencies between the program and the parameter settings. For more information about configuring GUI, refer section "CoreABC Configuration, page 24".

***Table 3 •*** **CoreABC Parameters**

| Parameter | Values | Description | Value | | |
|---|---|---|---|---|---|
| | | | Small | Medium | Large |
| APB_AWIDTH | 8 to 16 | Sets the width of the APB address bus. | 8 | 8 | 8 |
| APB_DWIDTH | 8, 16, or 32 | Sets the width of the APB data bus. | 8, 16, 32 | 8, 16, 32 | 8, 16, 32 |
| APB_SDEPTH | 1 to 16 | Sets the number of supported APB devices. | 1 | 4 | 16 |
| ICWIDTH | 1 to 16 | Sets the maximum number of supported instructions. Number of allowed instructions is $2^{ICWIDTH}$. ICWIDTH must be ≤ APB_AWIDTH. | 5 | 8 | 8 |
| ZRWIDTH | 0 to 16 | Sets the width of the Z register. A setting of 8 would allow for a maximum value of $2^8$ (i.e., 256). Zero will disable and remove the Z register. | 0 | 8 | 8 |
| IIWIDTH | 1 to 32 | Sets the width of the IO_IN input. IIWIDTH must be ≤ APB_DWIDTH. | 1 | 4 | 4 |
| IFWIDTH | 1 to 28 | Sets how many of the IO_IN bits can be used with the conditional instructions. IFWIDTH must be ≤ APB_DWIDTH – 4. | | | |

| Parameter | Values | Description | Value | | |
|---|---|---|---|---|---|
| | | | Small | Medium | Large |
| IOWIDTH | 1 to 32 | Sets the width of the IO_OUT output. IOWIDTH must be ≤ APB_DWIDTH. | 1 | 8 | 8 |
| STWIDTH | 1 to 8 | Sets the size of the internal stack counter used to support the call instruction and interrupt function. The depth of the stack is $2^{STWIDTH}$. | 1 | 4 | 4 |
| EN_RAM | 0 or 1 | When 1, a RAM block is used in the core to provide 256 storage locations. This RAM is also used to store return addresses for the call and interrupt functions. | 0 | 1 | 1 |
| EN_AND | 0 or 1 | When 1, the ALU supports the AND function. | 1 | 1 | 1 |
| EN_XOR | 0 or 1 | When 1, the ALU supports the XOR function. | 1 | 1 | 1 |
| EN_OR | 0 or 1 | When 1, the ALU supports the OR function. | 0 | 1 | 1 |
| EN_ADD | 0 or 1 | When 1, the ALU supports the ADD function. | 0 | 1 | 1 |
| EN_INC | 0 or 1 | When 1, the ALU supports the INC function. | 0 | 1 | 1 |
| EN_SHL | 0 or 1 | When 1, the ALU supports the SHL/ROL function. | 0 | 1 | 1 |
| EN_SHR | 0 or 1 | When 1, the ALU supports the SHR/ROR function. | 0 | 1 | 1 |
| EN_CALL | 0 or 1 | When 1, the core supports the call and return operations. | 0 | 1 | 1 |
| EN_PUSH | 0 or 1 | When 1, the core supports the push and pop operations. | 0 | 1 | 1 |
| EN_ACM | 0 or 1 | When 1, enables the ACM initialization table. | 0 | 1 | 1 |
| EN_DATAM | 0 to 3 | Controls internal multiplexing; see EN_DATAM Parameter, page 13 | 1 | 1 | 1 |
| EN_INT | 0 to 2 | Enables the external interrupt function. When 0, interrupts are disabled. When 1, INTREQ is active high. When 2, INTREQ is active low. | 0 | 1 | 1 |
| EN_MULT | 0 to 3 | Enables the hardware multiplier; four options exist (example for 16-bit core):<br>0: No hardware multiplier<br>1: Half multiplier, P(15:0) <= A(7:0) * B(7:0)<br>2: Full multiplier returning lower half, P(15:0) <= A(15:0) * B(15:0)<br>3: Full multiplier returning upper half, P(31:16) <= A(15:0) * B(15:0) | 0 | 0 | 0 |
| EN_IOREAD | 0 or 1 | When 1, the IOREAD instruction is enabled. | 0 | 1 | 1 |
| EN_IOWRT | 0 or 1 | When 1, the IOWRT instruction is enabled. | 1 | 1 | 1 |
| EN_ALURAM | 0 or 1 | When 1, the Boolean and Arithmetic instructions can operate on memory contents. | 0 | 1 | 1 |
| EN_INDIRECT | 0 or 1 | When 1, the Z register can be used to generate the APB address, and the APBWRTZ and APBREADZ instructions are enabled. | 0 | 0 | 1 |
| ISRADDR | 0 to 65,535 | The address CoreABC should jump to when responding to an interrupt request. | 0 | 220 | 220 |

| Parameter | Values | Description | Value | | |
|---|---|---|---|---|---|
| | | | Small | Medium | Large |
| INSMODE | 0 to 2 | When 0, the instructions are contained in internal logic gates, implementing a ROM function. When 1, internal RAM blocks are used to hold the instruction sequence. When 2, internal NVM is used to hold the instruction sequence. INSMODE = 2 is supported only on Fusion AFS devices. | 0 | 0 | 1 |
| ACT_CALIBRATIONDATA | 0 or 1 | When 1, the NVM block containing the calibration data for the device is selected if INSMODE = 2. When 0, any available NVM block may be used. This option is only applicable when INSMODE = 2, which implies that a Fusion AFS device is being used. | N/A | N/A | N/A |
| IMEM_APB_ACCESS | 0 to 2 | When 0, APB access to instruction memory is not supported. When 1, read only APB access to instruction memory is possible. When 2, read and write APB access to instruction memory is supported. | N/A | N/A | N/A |
| INITWIDTH | 1 to 16 | Specifies the width of the INITADDR input used to initialize the instruction RAM blocks when INSMODE = 1. The actual width depends on several generic values. Utilities used to support soft operation calculate this value. | 0 | 0 | 16 |
| DEBUG | 0 or 1 | When 1 during simulation, a detailed log will be generated of the internal operation. | N/A | N/A | N/A |
| TESTMODE | 0 to 16 | Selects a predefined set of instructions used for core verification. This should be set to 0 unless the verification test sequences are being used. | N/A | N/A | N/A |
| UNIQ_STRING_LENGTH | 0 to 256 | This parameter gives the length (number of characters) of the unique string which is derived from the instance name of a particular CoreABC instance. This parameter forms part of the mechanism which allows multiple instances of CoreABC to be easily used in a single design. | N/A | N/A | N/A |
| MAX_NVMDWIDTH | 16 or 32 | Indicates the maximum bit width supported on the data buses connecting to any NVM macro within CoreABC. This parameter is only applicable when CoreABC is configured to operate in NVM mode which is only possible for a Fusion AFS device. This parameter is not directly controllable from the configuration GUI but is instead automatically set to match the target device. A setting of 16 is applied when an AFS090 device is targeted. For all other devices the parameter is set to 32. | N/A | N/A | N/A |

## 4.3 EN_DATAM Parameter

This allows various internal multiplexers to be optimized out of the core, lowering tile counts. The settings supported are given in Table 4 through Table 7, and the tables show which instructions are allowed with each setting.

***Table 4 •*** **Accumulator Only (EN_DATAM = 0)**

| | Immediate Data | Accumulator |
|---|---|---|
| APBWRT | No | Yes + ACM |
| RAMWRT | No | Yes |
| PUSH | No | Yes |
| LOADZ | No | Yes |
| IOWRT | No | Yes |

***Table 5 •*** **Immediate Only (EN_DATAM = 1)**

| | Immediate Data | Accumulator |
|---|---|---|
| APBWRT | Yes | No |
| RAMWRT | Yes | No |
| PUSH | Yes | No |
| LOADZ | Yes | No |
| IOWRT | Yes | No |

***Table 6 •*** **Accumulator and Immediate (EN_DATAM = 2)**

| | Immediate Data | Accumulator |
|---|---|---|
| APBWRT | Yes | Yes + ACM |
| RAMWRT | Yes | Yes |
| LOADZ | Yes | Yes |
| PUSH | Yes | Yes |
| IOWRT | Yes | Yes |

***Table 7 •*** **Instruction-Dependent (EN_DATAM = 3)**

| | Immediate Data | Accumulator |
|---|---|---|
| APBWRT | No | Yes + ACM |
| RAMWRT | No | Yes |
| PUSH | No | Yes |
| LOADZ | Yes | No |
| IOWRT | Yes | No |

## 4.4 Ports

All CoreABC inputs are sampled, and outputs clocked, on the rising edge of PCLK.

***Table 8 •*** **CoreABC Port Descriptions**

| Name | Type | Description |
|---|---|---|
| PCLK | In | Clock input |
| NSYSRESET | In | Reset input (asynchronous active low) |
| PRESETN | Out | Reset output; synchronized version of NSYSRESET |
| PENABLE_M | Out | APB master interface enable signal |
| PWRITE_M | Out | APB master interface write signal |
| PSEL_M | Out | APB master interface select signal |
| PADDR_M[19:0] | Out | APB master interface address bus. The width of this address bus is fixed at 20 bits but some of the upper bits may not be significant, depending on the configuration of the core. The number of significant bits is determined by the APB_AWIDTH and the APB_SDEPTH parameters. Number of significant bits = APB_AWIDTH + |
| PWDATA_M[x:0] | Out | APB master interface write data bus. The width is controlled by APB_DWIDTH. |
| PRDATA_M[x:0] | In | APB master interface read data bus. The width is controlled by APB_DWIDTH. |
| PREADY_M | In | APB master interface ready input. |
| PSLVERR_M | In | APB master interface slave error signal. This input currently is not used by |
| IO_IN[x:0] | In | General-purpose inputs. The width is controlled by IIWIDTH. |
| IO_OUT[x:0] | Out | General-purpose outputs. The width is controlled by IOWIDTH. |
| INTREQ | In | Interrupt request input. When this input is asserted, the instruction sequence will jump to the address set by the ISRADDR parameter. |
| INTACT | Out | Indicates that the core has entered the interrupt service routine. |
| INITDATVAL | In | Enable signal (active high) indicating that the INITADDR and INITDATA inputs are valid. When using a SmartGen initialization client, this signal connects to the |
| INITDONE | In | Indicates that initialization has been completed (active high) and the core should start operating. |
| INITADDR[x:0] | In | Connects to the INITADDR output of the INITCFG block used to configure the RAM blocks when INSMODE = 1. When INSMODE = 0, these inputs should be tied to logic 0. The width of this input is controlled by the INITWIDTH generic. |
| INITDATA[8:0] | In | Connects to the INITDATA output of the INITCFG block, used to configure the RAM blocks when INSMODE = 1. When INSMODE = 0, these inputs should be |
| PSEL_S | In | Select signal of APB slave interface used to access instruction memory in NVM |
| PENABLE_S | In | Enable signal of APB slave interface used to access instruction memory in NVM |
| PWRITE_S | In | Write signal of APB slave interface used to access instruction memory in NVM |
| PADDR_S[x:0] | In | Address bus of APB slave interface used to access instruction memory in NVM mode. Width is determined by APB_AWIDTH. |
| PWDATA_S[x:0] | In | Write data bus of APB slave interface used to access instruction memory in NVM mode. Width is determined by APB_DWIDTH. |
| PRDATA_S[x:0] | Out | Read data bus of APB slave interface used to access instruction memory in NVM mode. Width is determined by APB_DWIDTH. |
| PSLVERR_S | Out | Error signal of APB slave interface used to access instruction memory in NVM |
| PREADY_S | Out | Ready signal of APB slave interface used to access instruction memory in NVM mode. |

# 5 CoreABC Programmer's Model

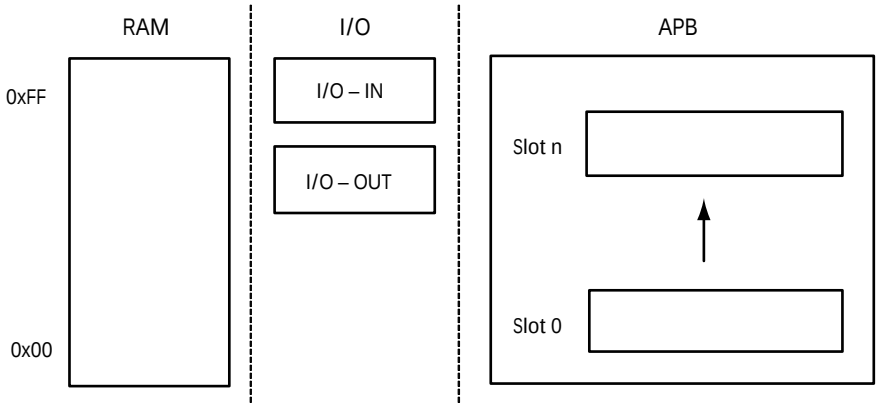
CoreABC is an accumulator based load/store architecture with multiple independent memory spaces. It is effectively a Harvard architecture (independent instruction and data address spaces). Most instructions act only on the accumulator, but there are specific instructions to access the memory spaces described below.

## 5.1 Address Spaces

The instruction address space is linear and is implemented as a hard-coded instruction table (hard mode), or an internal instruction RAM (soft mode), or an internal NVM block (NVM mode). This is implicitly accessed by control transfer instructions such as JUMP and CALL, but it cannot be directly read or written otherwise, except in the case where APB read/write data type access to instruction memory is enabled in NVM mode. In NVM mode, if APB data type read/write access to the instruction memory is enabled, it is possible to modify or overwrite CoreABC's program. Normally you will not want to do this and you must take care to ensure that the CoreABC program does not unintentionally corrupt itself. In practice this usually just means setting the SECTOR, PAGE, and SPARE_PAGE registers in the APB interface to NVM instruction memory to sufficiently high values. That is, read and write data type accesses to the NVM instruction memory should normally be to a region of the NVM above the program which is located from address 0x0000 onwards. For more information, refer to section "APB Access to Instruction Memory, page 41" .

The data address spaces are shown in Figure 3. There are three separate, independent addressable areas. These are accessed by using instructions or instruction modes unique to each one.

***Figure 3 •*** **CoreAPB Data Address Spaces**




### 5.1.1 Internal Data RAM Address Space (optional)

This is an optional internal 256-location RAM storage area. It can be accessed directly using the RAMREAD and RAMWRT instructions, and implicitly using the PUSH and POP instructions (the stack, if one is present, is located at the top of RAM). The ALU instructions can also source the secondary operand from the RAM storage area. The width of each RAM location is equal to the data width of the processor (APB_DWIDTH) or the width of the instruction counter (ICWIDTH), whichever is greater.

### 5.1.2 I/O Address Space

This is a general-purpose input/output area that is accessed by IOREAD (to load the accumulator from the input) or IOWRT (to write the accumulator to the output) and the INPUTn test instructions (to read the inputs—for example, JUMP IF INPUT3).

### 5.1.3 APB Address Space

The APB master interface of CoreABC typically will be connected to CoreAPB3 to provide access to up to 16 peripherals. If CoreABC is connected to CoreAPB3, the settings for the APB Slot Size configuration

options of these cores must match. For example, if CoreABC is configured for a slot size of 256 locations, CoreAPB3 must also have its slot size set to 256 locations. APB peripherals are accessed by APBWRT (to write to an APB peripheral) and APBREAD (to read from an APB peripheral). Both the slot number and the address within the slot must be specified in these instructions.

# 5.2 Registers

## 5.2.1 Accumulator

The accumulator (ACC) holds the result of data operations and is APB_DWIDTH (8, 16, or 32) bits wide.

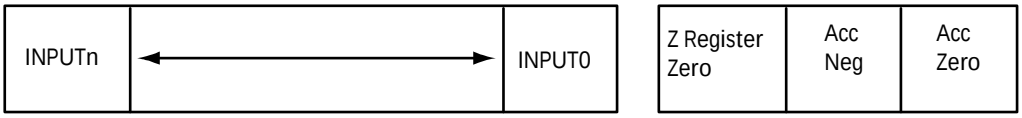
## 5.2.2 Z Register (Optional)

The optional Z register (Z) is a general purpose register which may be used, for example, as a loop counter. The Z register is used to provide the address to the APB space when the APBREADZ and APBWRTZ instructions are executed. When present, the Z register is ZRWIDTH (1 to 16) bits wide.

## 5.2.3 Flags Register—Inputs and Condition Codes

CoreABC maintains a control register that is used in the conditional instructions; for example, JUMP and CALL. This register cannot be read or used directly; instead, each named field can be used to control particular instructions. The Flags register has two sections, as shown in Figure 4.

***Figure 4 •*** **Flags and Inputs Register**




There are three condition code type flags:

- **ZERO:** Accumulator zero
- **NEGATIVE**: Accumulator negative
- **ZZERO:** Register zero

There are *n* INPUTS ($n \leq 28$), INPUT0 … INPUT*n*, which are directly mapped to the general purpose inputs connected to CoreABC's IO_IN[n:0] port. The number of these is configurable up to the lower of 28 or APB_DWIDTH – 4, where APB_DWIDTH is the width specified for the external APB data bus.

From these basic fields, other conditions are constructed and made available in the instruction set.

# 5.3 Instruction Set

Table 9 through Table 18 list the supported instructions. On the right hand side of these tables there are columns entitled Flags and Cycles. The Flags column contains two sub-columns, Acc. Zero and Acc. Neg., and the entries under these columns are either Yes or No. "Yes" indicates that the relevant flag, accumulator zero (Acc. Zero) or accumulator negative (Acc. Neg.), is affected by the instruction named in that row of the table. Similarly, a "No" entry indicates that the flag is not affected by the instruction. The entries in the Cycles column give the number of (PCLK) clock cycles required for each instruction.

***Table 9 •*** **The Boolean and Arithmetic Instruction Group**

| Instruction[1, 2] | Description | Flags | | Cycles |
|---|---|---|---|---|
| | | Acc. Zero | Acc. Neg. | |
| LOAD DAT *Data* | Load accumulator with value. | Yes | Yes | 3 |
| LOAD RAM *Address* | Load accumulator with value. | Yes | Yes | 3 |
| AND DAT *Data* | Bitwise AND accumulator with immediate data. | Yes | Yes | 3 |
| AND RAM *Address* | Bitwise AND accumulator with RAM location. | Yes | Yes | 3 |
| OR DAT *Data* | Bitwise OR accumulator with immediate data. | Yes | Yes | 3 |

***Table 9 •*** **The Boolean and Arithmetic Instruction Group**

| | | | | |
|---|---|---|---|---|
| OR RAM *Address* | Bitwise OR accumulator with RAM location. | Yes | Yes | 3 |
| XOR DAT *Data* | Bitwise XOR accumulator with immediate data. | Yes | Yes | 3 |
| XOR RAM *Address* | Bitwise XOR accumulator with RAM location. | Yes | Yes | 3 |
| INC | Increment accumulator. | Yes | Yes | 3 |
| DEC | Decrement accumulator. | Yes | Yes | 3 |
| ADD DAT *Data* | Add immediate data to accumulator. | Yes | Yes | 3 |
| ADD RAM *Address* | Add RAM location to accumulator. | Yes | Yes | 3 |
| SUB DAT *Data* | Subtract immediate data from accumulator. SUB RAM is not supported. | Yes | Yes | 3 |
| MULT DAT *Data* | Multiply accumulator by immediate data.<br>Core parameters determine multiplier return value. | Yes | Yes | 3 |
| MULT RAM *Address* | Multiply accumulator by RAM location.<br>Core parameters determine multiplier return value. | Yes | Yes | 3 |
| CMP DAT *Data* | Compare accumulator to immediate data.<br>ZERO set if equal; NEGATIVE set if MSBs differ. | Yes | Yes | 3 |
| CMP RAM *Address* | Compare accumulator to RAM location.<br>ZERO set if equal; NEGATIVE set if MSBs differ. | Yes | Yes | 3 |
| CMPLEQ DAT *Data* | Compare accumulator to immediate data.<br>ZERO set if equal; NEGATIVE set if ACC < Data.<br>CMPLEQ RAM is not supported. | Yes | Yes | 3 |
| SHL0 | Shift accumulator left and infill with 0. | Yes | Yes | 3 |

**Note:**

1. *For most instructions, when using the configuration GUI, the DAT keyword can be omitted.*
2. *DAT may be replaced with DAT8 or DAT16 when only lower 8 or 16 data bits contain valid data. Using DAT8/DAT16 will reduce tile counts when instructions are held in logic tiles (that is, when the core is configured to operate in hard mode).*

***Table 10 •*** **The Boolean and Arithmetic Instruction Group (continued)**

| | | **Flags** | | |
|---|---|---|---|---|
| **Instruction1, 2** | **Description** | **Acc. Zero** | **Acc. Neg.** | **Cycles** |
| SHR0 | Shift accumulator right and infill with 0. | Yes | Yes | 3 |
| SHL1 | Shift accumulator left and infill with 1. | Yes | Yes | 3 |
| SHR1 | Shift accumulator right and infill with 1. | Yes | Yes | 3 |
| SHLE | Shift accumulator left and infill with LSB. | Yes | Yes | 3 |
| SHRE | Shift accumulator right and infill with MSB. | Yes | Yes | 3 |
| ROL | Rotate accumulator left. | Yes | Yes | 3 |
| ROR | Rotate accumulator right. | Yes | Yes | 3 |
| BITCLR *Data* | Clear one bit in accumulator specified by argument (AND). In this case, the data value specifies the bit | Yes | Yes | 3 |

***Table 10 •*** **The Boolean and Arithmetic Instruction Group (continued)**

| | | | | |
|---|---|---|---|---|
| BITSET *Data* | Set one bit in accumulator specified by argument (OR). In this case, the data value specifies the bit position. | Yes | Yes | 3 |
| BITTST *Data* | Test one bit in accumulator. ZERO set if all requested bits are clear. In this case, the data value specifies the bit position. | Yes | Yes | 3 |

**Note:**

1. *For most instructions, when using the configuration GUI, the DAT keyword can be omitted.*
2. *DAT may be replaced with DAT8 or DAT16 when only lower 8 or 16 data bits contain valid data. Using DAT8/DAT16 will reduce tile counts when instructions are held in logic tiles (that is, when the core is configured to operate in hard mode).*

***Table 11 •*** **The Memory Instruction Group**

| | | Flags | | |
|---|---|---|---|---|
| **Instruction** | **Description** | **Acc. Zero** | **Acc. Neg.** | **Cycles** |
| PUSH | Push the accumulator onto the stack. | No | No | 3 |
| PUSH ACC | Push the accumulator onto the stack. | No | No | 3 |
| PUSH DAT *Data* | Push immediate data onto stack. | No | No | 3 |
| POP | Pop data from the stack to the accumulator and update the flags. | Yes | Yes | 3 |
| RAMWRT *Address* ACC | Write accumulator to RAM address. | No | No | 3 |
| RAMWRT *Address* DAT *Data* | Write immediate data to RAM address. | No | No | 3 |
| RAMREAD *Address* | Read data from RAM address to the accumulator and update the flags. | Yes | Yes | 3 |

***Table 12 •*** **The Z Register* Instruction Group**

| | | Flags | | |
|---|---|---|---|---|
| **Instruction** | **Description** | **Acc. Zero** | **Acc. Neg.** | **Cycles** |
| LOADZ ACC | Load Z with accumulator. | No | No | 3 |
| LOADZ DAT *Data* | Load Z with immediate value. | No | No | 3 |

**Note:** **The Z register is intended to be used as loop counter or APB address register.*

***Table 13 •*** **The Z Register* Instruction Group**

| | | Flags | | |
|---|---|---|---|---|
| **Instruction** | **Description** | **Acc. Zero** | **Acc. Neg.** | **Cycles** |
| ADDZ ACC | Add accumulator to Z and store in Z. Only ZZERO flag is affected. | No | No | 3 |
| ADDZ DAT *Data* | Add immediate data to Z and store in Z. Only ZZERO flag is affected. | No | No | 3 |

***Table 13 •*** **The Z Register* Instruction Group**

| | | | | |
|---|---|---|---|---|
| SUBZ DAT *Data* | Subtract immediate data from Z and store in Z. Only ZZERO flag is affected<br>SUBZ ACC is not supported. | No | No | 3 |
| INCZ | Increment Z. Only ZZERO flag is affected. | No | No | 3 |
| DECZ | Decrement Z. Only ZZERO flag is affected. | No | No | 3 |

**Note:** **The Z register is intended to be used as loop counter or APB address register.*

***Table 14 •*** **The APB Instruction Group**

| Instruction | Description | Flags | | Cycles |
|---|---|---|---|---|
| | | Acc. Zero | Acc. Neg. | |
| APBREAD Slot Address | Read from APB. | No | No | 5 |
| APBWRT ACC Slot Address | Write accumulator to APB at chosen | No | No | 5 |
| APBWRT ACM Slot Address | Write value of ACM table, at location given by accumulator, to APB at chosen address. | No | No | 5 |
| APBWRT DAT Slot Address Data | Write data to chosen address. | No | No | 5 |
| APBREADZ Slot | Read from APB. The Z register specifies the APB address. | No | No | 5 |
| APBWRTZ ACC Slot | Write accumulator to APB. The Z register specifies the APB address. | No | No | 5 |
| APBWRTZ ACM Slot | Write value of ACM table, at location given by accumulator. The Z register specifies the APB address. | No | No | 5 |
| APBWRTZ DAT Slot Data | Write data; the Z register specifies the APB address. | No | No | 5 |

***Table 15 •*** **The I/O Instruction Group**

| Instruction | Description | Flags | | Cycles |
|---|---|---|---|---|
| | | Acc. Zero | Acc. Neg. | |
| IOWRT ACC | Write accumulator to I/O register. | No | No | 3 |
| IOWRT DAT *Data* | Write data value to I/O register. | No | No | 3 |
| IOREAD | Load the accumulator with the I/O input value. | No | No | 3 |

***Table 16 •*** **The Flow Control Instruction Group**

| Instruction | Description | Flags | | Cycles |
|---|---|---|---|---|
| | | Acc. Zero | Acc. Neg. | |
| JUMP Condition $Label | Jump to label. | No | No | 3 |

***Table 16 •*** **The Flow Control Instruction Group**

| | | | | |
|---|---|---|---|---|
| JUMP IF/IFNOT Condition $Label | Jump on condition to label. | No | No | 3 |
| WAIT UNTIL/WHILE Condition | Stop at this instruction until condition is TRUE. | No | No | 3 |
| CALL $Label | As JUMP, but puts return address on | No | No | 3 |
| CALL IF/IFNOT Condition $Label | As JUMP, but puts return address on | No | No | 3 |
| RETURN | Return from a CALL. | No | No | 3 |
| RETURN IF/IFNOT Condition | Return from a CALL on condition. | No | No | 3 |
| RETISR Condition | Return from an interrupt. | No | No | 3 |
| RETISR IF/IFNOT Condition | Return from an interrupt on condition. | No | No | 3 |
| HALT | Stop at this instruction. Interrupts will still be processed. HALT is a synonym for WAIT, and generally used without a | No | No | Indefinite |

***Table 17 •*** **Conditions for Flow Control Instruction Group**

| Condition | Description |
|---|---|
| ALWAYS | Always. You can get the same effect as this by not specifying any condition. |
| ZERO | Accumulator zero |
| NEGATIVE | Accumulator negative |
| ZZERO | Z register zero |
| INPUT0 | Input0 set |
| INPUT1 | Input1 set *and similarly for higher Inputs, if available.* |
| POSITIVE | Equivalent to NOT NEGATIVE |
| LTE_ZERO | Less than or equal to zero; the combination NEGATIVE OR ZERO |
| GT_ZERO | Greater than zero; the combination NOT (NEGATIVE OR ZERO) |

***Table 18 •*** **Other Instructions**

| | | Flags | | |
|---|---|---|---|---|
| **Instruction** | **Description** | **Acc. Zero** | **Acc. Neg.** | **Cycles** |
| NOP | No operation | No | No | 3 |

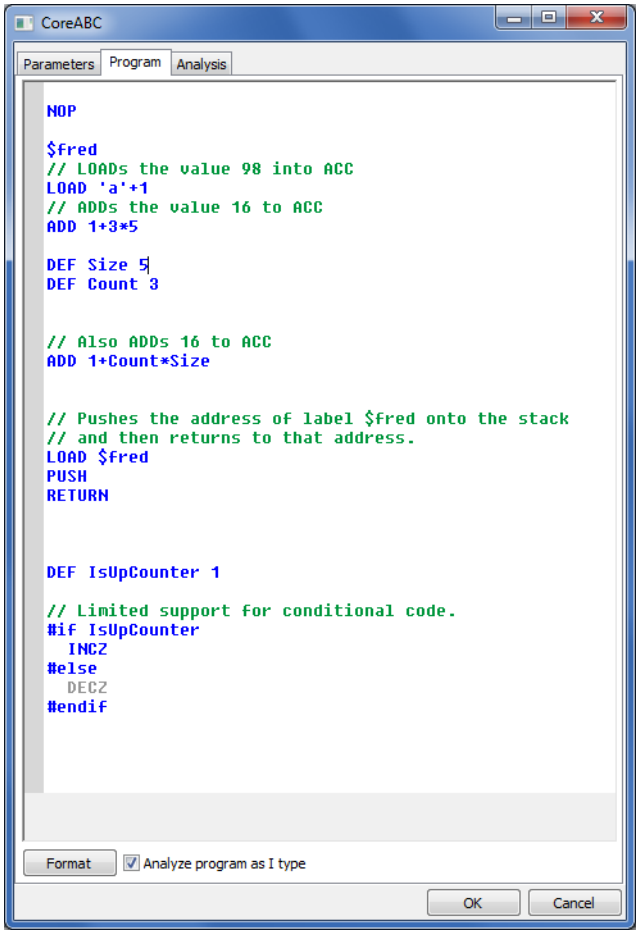
## 5.3.1 Constant Expressions

In most places where a constant appears in a CoreABC instruction a constant expression can be used instead. The expression can use +, -, *, /, &, ^, |, <, ==, >, <=, !=, >=, <<, >> and braces. Constants can use decimal, a prefix of 0x for hexadecimal or 0b for binary. An ASCII character in single '' can also be used; for example, 'a' is equivalent to 97 or 0x61. Labels and named constants created using 'DEF' can also be used. Labels begin with $. Examples of labels are $LABEL_1 and $LABEL_2. In the example program illustrated in Figure 5, the use of constant expressions is shown in the top half of the program.

## 5.3.2 Conditional Code

There is limited support for conditional code. Anything inside a #if, #endif pair will be excluded if the constant expression after the #if is zero or negative, and will be included otherwise. You can also use an #if, #else, #endif construct, where depending on the expression after the #if, either the assembly code between the #if and the #else or the assembly code between the #else and #endif is included. Code that is not being included is shown greyed out. An #if, #else, #endif construct is shown towards the end of the program illustrated in Figure 5. In this example, IsUpCounter has been defined as 1 so the INCZ instruction is included and the DECZ instruction is omitted. DECZ is greyed out to indicate that it is not included.

***Figure 5 •*** **Conditional Code**



# 6 CoreABC Operation

## 6.1 ACM Lookup Table for Use with CoreAI

When generating a SmartDesign design that contains an instance of CoreABC, a check is made to detect if an instance of CoreAI is being mastered by CoreABC's APB master interface. CoreAI may be connected directly to CoreABC if it is the only APB slave being controlled by CoreABC but, more typically, CoreAI will be one of a number of slaves under the control of CoreABC, with all the cores connected together using the CoreAPB3 bus fabric core. In either scenario the presence of CoreAI in CoreABC's APB address space will be detected.

If CoreABC is controlling a CoreAI instance, a lookup table will be implemented within CoreABC. This lookup table will hold data for initializing the CoreAI analog configuration multiplexer (ACM) and the table's contents will be derived from the configuration information entered in CoreAI's configuration GUI. The APBWRT ACM instruction can be used in CoreABC's program to easily load the ACM initialization values for CoreAI. This instruction uses the accumulator value to index into the ACM lookup table to generate the actual data value written. The "Example Instruction Sequence, page 57" shows the initialization of ACM registers (within the instruction loop beginning with the label "$WaitRegProg").

**Note:** CoreAI is a Fusion AFS-specific core, which means that it can only be used on a Fusion AFS device. This implies that the ACM lookup table and the APBWRT ACM instructions described in the preceding paragraphs are only relevant when designing for a Fusion AFS device.

## 6.2 Stack

The upper $2^{STWIDTH}$ memory locations in the 256-location internal storage are used for storing the stack contents. If STWIDTH = 4 (stack is 16 locations deep), the stack will occupy locations 0xF0 to 0xFF. There is no underflow or overflow detection on the stack pointer, so it will simply wrap around from 0xF0 to 0xFF on push operations and 0xFF to 0xF0 on pop operations (assuming STWIDTH = 4).

The RAMREAD and RAMWRT instructions can be used to read and modify the values pushed onto the stack. An indirect jump instruction can be implemented by pushing the required jump address on the stack and executing a return instruction.

## 6.3 Interrupt Operation

When INTREQ is asserted, the core will jump to the interrupt service routine (ISR) on completion of the current instruction. As it does so, it will assert the INTACT (interrupt active) output. The last instruction in the ISR should be a RETISR (return from ISR) instruction. When the RETISR instruction is executed, the INTACT output is cleared. INTACT acts as the interrupt acknowledge, and INTREQ should be deactivated when INTACT becomes active. The core will ignore additional interrupt requests while INTACT is active.

The core will respond to an interrupt request within six clock cycles—five clock cycles for the current instruction to complete, [1]plus one additional clock cycle in the core. The value held in the instruction counter is pushed onto the stack on entering the interrupt service routine. This value is popped from the stack when exiting the routine to provide the return address. The contents of the ZERO and NEGATIVE flags are saved internally (rather than on the stack) on entry to the interrupt service routine and restored on the RETISR instruction. When the ISR is entered, the ZERO and NEGATIVE flags will contain the flag values present when the previous ISR was executed. The accumulator register is not saved on entry to the ISR. The ISR should push and pop the accumulator to preserve its contents.

The INTREQ polarity can be active low or active high. This is set by the EN_INT parameter.

If an interrupt occurs when the HALT or WAIT instructions are being executed, then, after completion, the ISR will return to the HALT or WAIT instruction, unless the ISR does something to remove the reason for

1. *If an APB-related instruction (such as APBREAD or APBWRT) is active when the interrupt occurs, more than five cycles may be required for the instruction to complete if the APB access is extended by pulling the PREADY_M input low for a number of cycles.*

the WAIT or modifies the stack contents; for example, it could POP the return address, modify it, and PUSH it back on the stack.

When the interrupt functionality is being used, CoreABC's program will often be structured such that the first instruction (at instruction address 0) is a JUMP to the main loop of the program and the ISR will be located immediately after this, at instruction address 1. The instructions of the main loop will be located just after the ISR in the program memory.

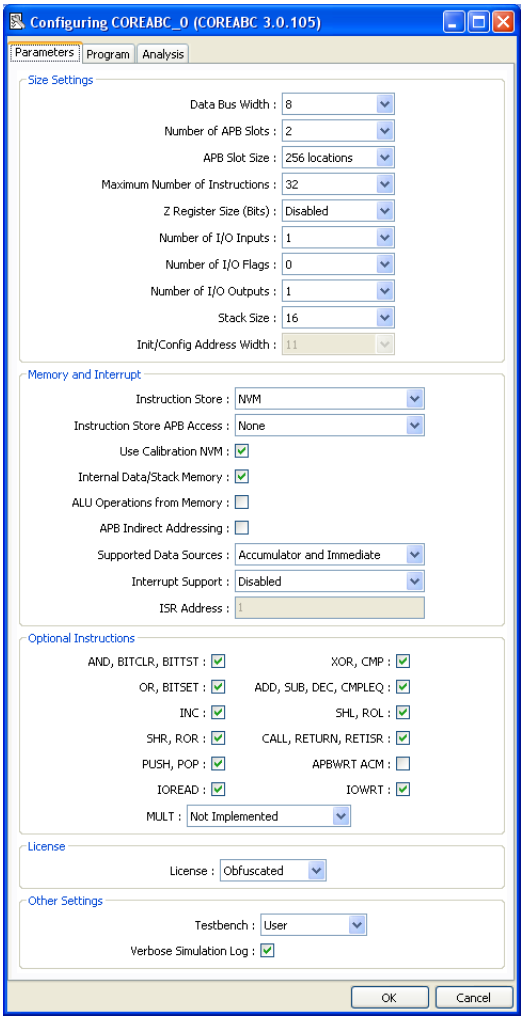
# 7 CoreABC Configuration

The CoreABC configuration GUI is launched when instantiating the core in a SmartDesign design. After instantiation, the configuration GUI can be opened by double-clicking on the CoreABC instance or by right-clicking and selecting Configure Component.

The configuration GUI has three tabs: Parameters, Program, and Analysis.

Select the Parameters tab on the CoreABC configuration GUI to begin configuring the core. When you do this, you will see the screen shown in Figure 6.

***Figure 6 •*** **Configuration Parameters**




## 7.1 Configurable Options

Each of the configurable options presented on the Parameters tab of the configuration GUI is explained in the following sections.

### 7.1.1 Data Bus Width

Selects the width of the data bus within CoreABC and on the APB master interface (and on the APB slave interface, if present). Possible settings are 8, 16, or 32 bits. The accumulator width is equal to the value set for Data Bus Width.

### 7.1.2 Number of APB Slots

This sets the maximum number of APB slots CoreABC can address. Each slot is a location for connecting an APB peripheral, so ensure that you allocate enough slots for your design. It is easy to set this at a later stage in your design if you wish, when you have a clear understanding of the peripherals you are connecting.

### 7.1.3 APB Slot Size

This sets the number of locations in each APB slot. Possible settings range from 256 to 64K locations. This setting should match the corresponding setting (also called APB Slot Size) on any instance of CoreAPB3 mastered by CoreABC.

### 7.1.4 Maximum Number of Instructions

This allocates the instruction space for your program (in a range from 2 to 65,536 instructions). You should not make this larger than necessary, as it is used for configuring multiplexers and will directly impact the size of the core.

### 7.1.5 Z Register Size

This sets the maximum Z register size you intend to use in your program. This is used to set the size on the Z register and associated logic, so the smaller you make it, the smaller your core. There is also a disable setting to remove this feature.

### 7.1.6 Number of I/O Inputs

This sets the number of inputs configured on CoreABC. These can be read using the IOREAD instruction. The range is 1–32.

### 7.1.7 Number of I/O Flag Inputs

This sets the number of inputs connected into the conditional logic. These are accessible for controlling JUMP and similar instructions as INPUT0 – INPUT27 (note that the first input is INPUT0!). The range is 1–28.

### 7.1.8 Number of I/O Outputs

This sets the maximum number of output lines from CoreABC. The range is 1–32. These can be written to using the IOWRT instruction, which allows the accumulator to be written to the output register.

### 7.1.9 Stack Size

CALL and RETURN instructions use a stack to store the return address when subroutines are used. The stack size can be set in this drop-down list. Note that this list will be grayed out (disabled) if Internal Data/Stack Memory is not enabled, because the stack is allocated from that memory.

### 7.1.10 Instruction Store

This is a very important setting that determines whether CoreABC is in hard, soft, or NVM mode. The options are as follows:

- Hard (FPGA tiles) – The program instructions are stored in FPGA tiles which are effectively used to build a hard-coded ROM. No RAM or NVM blocks are instantiated for instruction storage.
- Soft (FPGA RAM) – The program instructions are stored in RAM blocks instantiated inside CoreABC.
- NVM – The program instructions are stored in an NVM block instantiated within CoreABC. This instruction store option is only available on Fusion AFS devices.

### 7.1.11 Init/Config Address Width

This is only applicable when Instruction Store is set to Soft. This option sets the address width of the InitCfg interface for initializing the RAM blocks which provide the instruction store inside a soft mode CoreABC. The easiest way to determine the setting for this option is to look at the Instruction Store Details section under the Analysis tab. The fifth bullet point in this section gives the required width in number of bits.

### 7.1.12 Instruction Store APB Access

This is only applicable when Instruction Store is set to NVM. This option sets the type of access to the instruction store NVM. Possible settings are None, Read Only, or Read/Write.

### 7.1.13 Use Calibration NVM

This check box option is only applicable when Instruction Store is set to NVM and, when selected, causes CoreABC to request use of the NVM block holding the device calibration data for its instruction store. One of the NVM blocks on the device will hold the calibration data in a spare page. Checking this option causes the ACT_CALIBRATIONDATA parameter to be set to 1 on the NVM instance within (an NVM mode) CoreABC instance. It is possible that some other NVM instance not related to CoreABC in the design may also have its ACT_CALIBRATIONDATA parameter set to 1. In this case, CoreABC is not guaranteed to be allocated the NVM block holding the calibration data.

### 7.1.14 Internal Data/Stack Memory

Set this option **ON** if you are going to use the internal scratchpad RAM (with RAMREAD and RAMWRT instructions) or the stack (for CALL and RETURN instructions).

### 7.1.15 ALU Operation from Memory

This allows the ALU data input to accept both immediate data and data from the RAM. It enables ADD RAM and similar instructions.

### 7.1.16 APB Indirect Addressing

This allows the Z register to be used as the source of the APB address for APB instructions (that is, setting this option effectively enables instructions such as APBREADZ and APBWRTZ).

### 7.1.17 Supported Data Sources

This controls the EN_DATAM parameter; refer to “EN_DATAM Parameter, page 13". Setting this to “Accumulator and Immediate” will increase tile counts.

### 7.1.18 Interrupt Support

This allows you to enable or disable interrupt support. If you specify active high or active low interrupt, the interrupt logic is automatically included. When you enable the interrupt logic, you should also set the **ISR Address**.

### 7.1.19 ISR Address

The ISR address should be set when you have enabled the interrupt logic. It is the instruction address from which CoreABC will fetch the next instruction to be executed after an interrupt is detected. At the end of the ISR, you will have a return from interrupt (RETISR) instruction. The default value for ISR address is 1. This setting is suitable for a program which is structured such that the first instruction (at address 0) is a jump to the main part of the program and the ISR is located from address 1 onward (the main part of the program would be located just after the ISR code).

### 7.1.20 Optional Instructions

There is a range of instructions that can be omitted or included in CoreABC to control the size. This empowers you to make size/performance tradeoffs. If you have used omitted instructions in your program, you will receive a validation warning.

### 7.1.21 License

This option is used to generate either obfuscated or plain text RTL code for the core, depending on the type of license you have. An obfuscated license enables you to generate obfuscated RTL code. An RTL license permits generation of either obfuscated or plain text RTL code.

### 7.1.22 Testbench

Set this to User if you want a user testbench generated with your core.

### 7.1.23 Verbose Simulation Log

This enables the feature that allows CoreABC to log the operations being performed during simulation along with the current accumulator values. See the "Testbench, page 45" for more details.

## 7.2 Cross-Validation of Configuration Fields

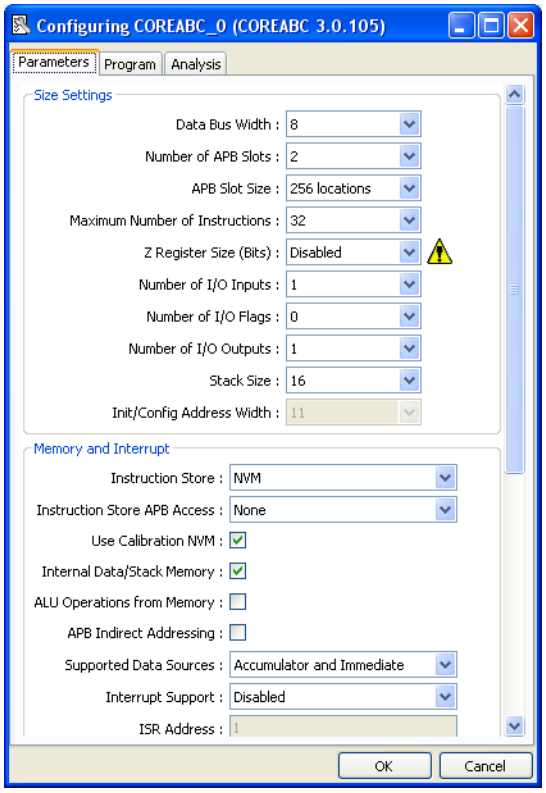
There is extensive cross-validation of settings in the CoreABC configuration screen to ensure that the overall configuration is consistent. This also extends to validation between the program and the configuration. Most possible inconsistencies are covered.

Figure 8 shows the symbols that are displayed to indicate a possible error. When you click the symbol (Figure 7), information is given as to the precise nature of the problem.

***Figure 7 •*** **Error Symbol**

In the example shown in Figure 8, the Maximum Z Register has been set to Disabled, but there is an instruction in the program (LOADZ) which requires that the Z register features are available.

***Figure 8 •*** **CoreABC Configuration Validation**




In general, the validation is more extensive on the Parameters tab than on the Program tab, so it is a good idea to take a look at the Parameters tab when you have completed writing your program.

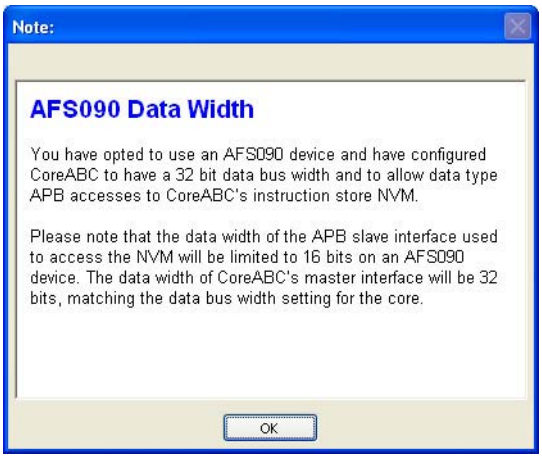
Some cross-validation actually grays out fields that are inappropriate when other settings have not been made.

## 7.3 NVM Data Width on AFS090 Device

On an AFS090 device, the data width when accessing NVM is limited to a maximum of 16 bits. On other Fusion AFS devices, 32 bit access to NVM is supported. This has implications when targeting an NVM mode CoreABC design at an AFS090 device. For such a design, if a data bus width of 32 is selected along

with read only or read/write APB access to the instruction store NVM, the message shown in Figure 9 will be displayed on pressing the OK button of the CoreABC configuration GUI.

***Figure 9 •*** **AFS090 Data Width Message**

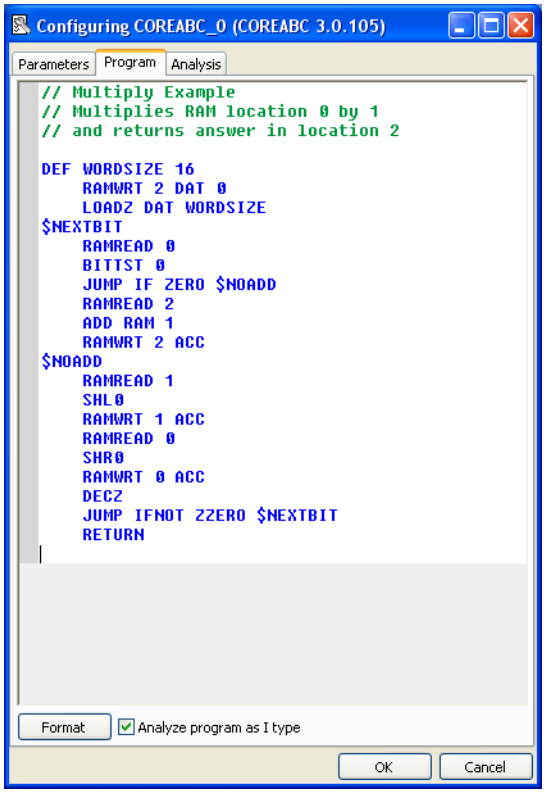



As the message indicates, APB accesses to the NVM instruction store will be limited to 16 bits in this case, even though a data width of 32 has been selected. 32-bit access is supported to any other slaves which may be connected to the APB bus. Click the **OK** button on this message to dismiss the message (and the CoreABC configuration window).

# 8 CoreABC Programming

CoreABC programs are written and assembled under the Program tab of the CoreABC Configuration GUI, as shown in Figure 10. You can view an analysis of your code under the Analysis tab.

***Figure 10 •*** **CoreABC Programming Screen**




## 8.1 Analysis

If the **Analyze program as I type** check box is selected (under the Program tab), your program is continuously analyzed as you write it, to detect any syntax or other errors. These errors are immediately flagged, and information about them is provided. Color coding of the program is used, with comments appearing in green, valid instructions in blue, and errors in red. As the program becomes larger, analysis takes longer with each character typed and this eventually impacts usability. If this is an issue, you can turn off analysis (by clearing the check box) when you enter the program. You can then turn on the analysis again when the program is complete or almost complete.

Under the Analysis tab, you will find useful information and statistics on your program, most of which is self-explanatory. For example, the instructions used in the program are listed and this information may be useful for optimizing your CoreABC instance by omitting support for any unused instructions (under the Optional Instructions section of the Parameters tab). In soft or NVM mode, the Analysis tab will also contain information of use when creating a Flash Memory System Builder Data Storage or Initialization Client.

## 8.2 CoreABC Instruction Modes

The instruction store configuration option (INSMODE parameter) controls how CoreABC's instructions are stored. For all device families, hard mode and soft mode instruction stores are possible. For the Fusion AFS family, an additional NVM mode is also available. Each of these instruction storage modes is described below.

### 8.2.1 Hard Mode

In hard mode, the instructions are stored in FPGA tiles. Essentially, tiles are used to build an instruction ROM. The instructions.v or instructions.vhd RTL file implements the instruction store and this file is automatically created when a CoreABC based design is generated within SmartDesign.

From a design implementation point of view, hard mode is probably the simplest mode. The RTL files completely describe the core and its program and can simply be run through synthesis, compile, layout, etc., along with any other components in the design.

### 8.2.2 Soft Mode

In soft mode, the instructions are stored in RAM blocks on the device. The number of RAM blocks required to hold the program increases with increasing program size and instruction width. The instruction width increases with increasing address width (APB_AWDITH), data width (APB_DWIDTH), and number of locations per APB slot (APB_SDEPTH). Details on the instruction width and the number of instruction store RAM blocks required can be found under the Analysis tab of the CoreABC configuration GUI.

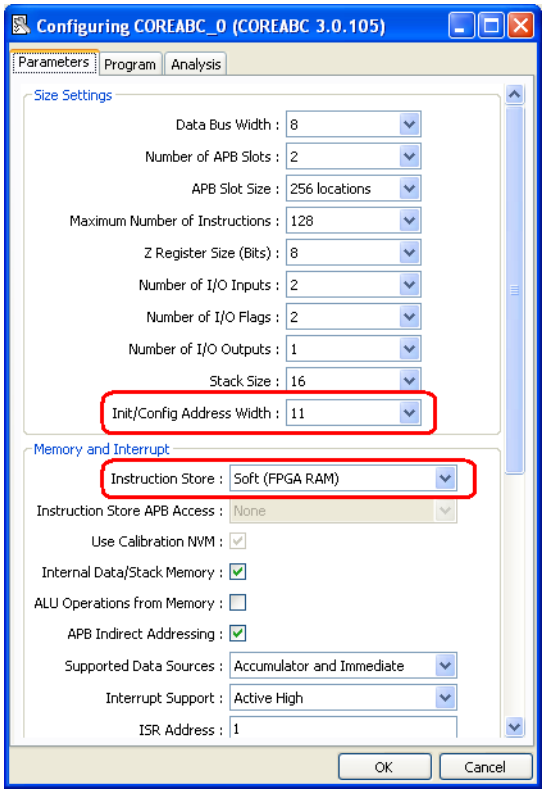
In soft mode, the instructram.v or instructram.vhd RTL file instantiates the required number of RAM blocks within CoreABC. When a design containing a soft mode CoreABC instance is generated in SmartDesign, memory files are created for initializing the instruction RAM blocks during simulation. These files (one per RAM block) are automatically placed in the project's simulation folder to facilitate easy simulation. In addition to these files, a single, consolidated memory file is created. This file is intended for use in initializing the instruction RAM when the design is implemented on a device. When a Fusion AFS device is being used, this consolidated memory file typically will be used to create a RAM Initialization client using the Flash Memory System Builder (FMSB) utility. For a non-Fusion AFS device, you must manually implement some other means of initializing the RAM blocks.

#### 8.2.2.1 Soft Mode Flow on a Fusion AFS Device

The following sequence of steps describes how to implement a soft mode CoreABC instance on a Fusion AFS device. The steps describe the use of an FMSB RAM Initialization client to initialize CoreABC's instruction RAM.

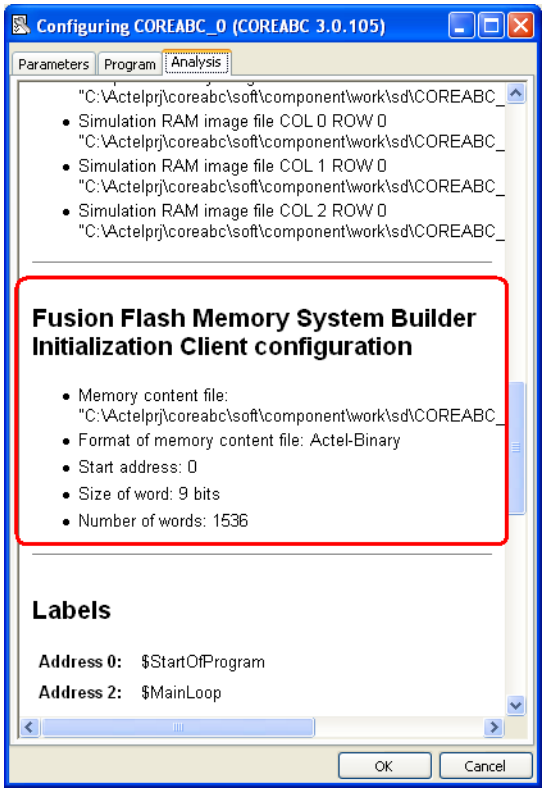
1. Set the Instruction Store option to **Soft (FPGA RAM)** as shown in Figure 11. If there are any validation warnings, ensure that the Init/Config Address Width is configured appropriately.

***Figure 11 •*** **Init/Config Address Width**



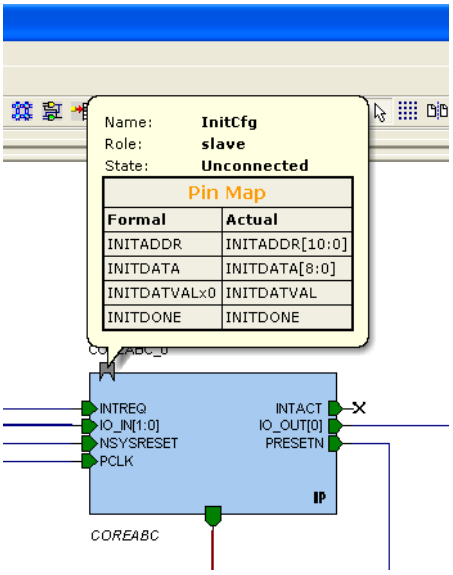

2. In the CoreABC configurator Analysis view, as shown in Figure 12, note the configuration details which will be needed when configuring a Fusion AFS Flash Memory System Builder [RAM] Initialization Client.

***Figure 12 •*** **Initialization Client Configuration**




3. These FMSB Initialization Client configuration details are also written to the CoreABC.log file, which appears in the **Design Explorer** > **Files view** under **Components** > [**SmartDesign-name**]> **Report Files**.
4. Save the CoreABC configuration. Note in SmartDesign that the InitCfg bus interface now appears on the CoreABC instance Figure 13.

***Figure 13 •*** **CoreABC Instance**



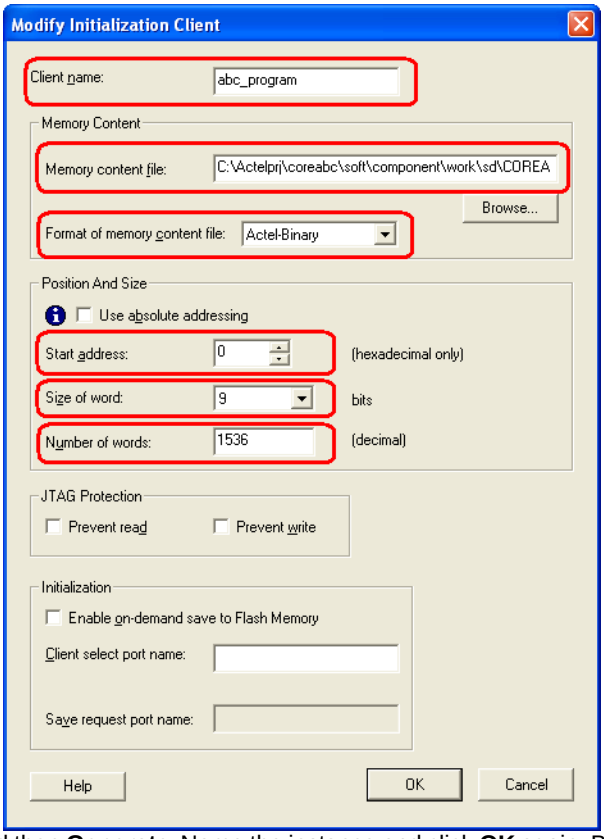

5. The next task is to instantiate, configure, stitch, and generate a Fusion AFS Flash Memory System Builder initialization client into the design to store the soft mode program image in an NVM block and to initialize the CoreABC soft mode program storage RAM blocks at startup time. However, we do not yet have the required soft mode program image so cannot do this yet. For this reason we must generate the currently incomplete design first.

Choose **SmartDesign** > **Generate Design**.

You will get a warning about the CoreABC InitCfg bus interface not being connected, but you can ignore this for now (or temporarily mark this bus interface unused when generating here).

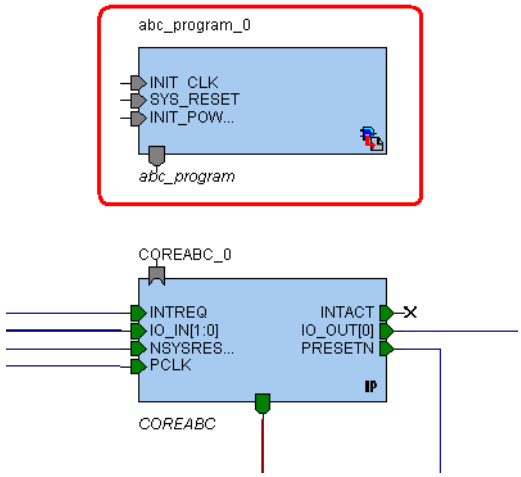
6. Open the CoreABC.log file mentioned in step 2 so that you can view the configuration details required for the soft mode CoreABC's FMSB initialization client. Select and copy the name of the binary format RAM memory image file and keep CoreABC.log visible while configuring the FMSB initialization client.
7. Go to the Libero IDE Catalog, expand the treeview under Fusion AFS Peripherals and double-click on the **Flash Memory System Builder**. Choose an initialization client, click **Add to System,** and then configure the client to match the details given in CoreABC.log (Figure 14).

***Figure 14 •*** **Modify Initialization Client**



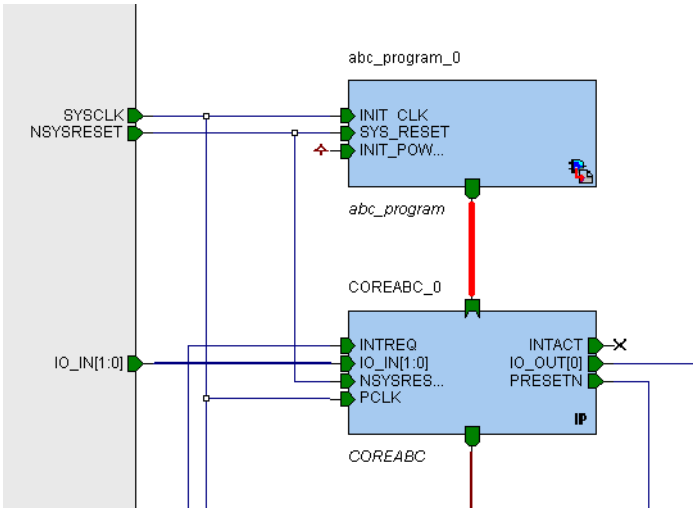

8. Click **OK** and then **Generate.** Name the instance and click **OK** again. Back in SmartDesign, the Flash Memory System Builder initialization client instance should now appear as shown in Figure 15.

***Figure 15 •*** **Flash Memory System Builder Initialization Client**



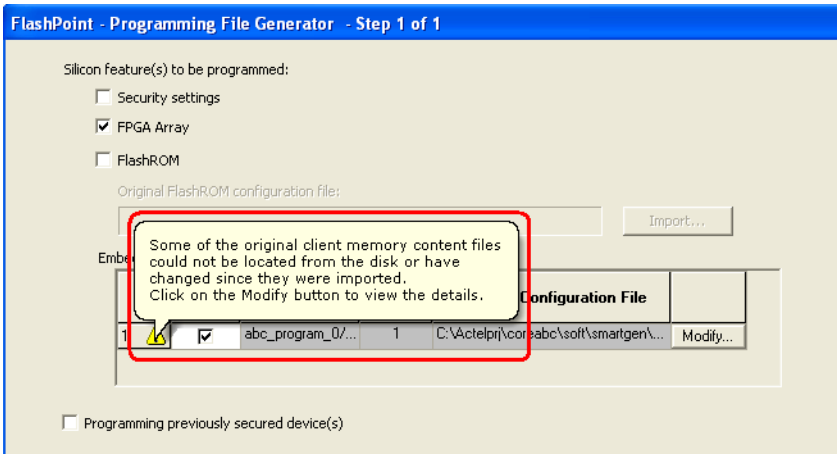

9. Select the **Initialization client** instance and choose **SmartDesign** > **Auto Connect Selected Instance(s)** and SmartDesign will connect the CoreABC's slave InitCfg slave interface to the initialization client's master interface. Manually connect the remaining initialization client's signals, as shown in Figure 16.

***Figure 16 •*** **Connect Initialization Client's Signal**




10. The design is now complete and can be generated using **SmartDesign** > **Generate Design**.
11. Go to the Libero IDE Project Flow view and click on Synplify® to run synthesis.
12. When synthesis has completed, exit Synplify and then click on Place & Route to run Compile, Layout, and Programming File generation.
13. When you click on **Programming File** in Designer to run FlashPoint, to generate the programming file (a PDB file, for example), you will get the warning shown in Figure 17 if the SmartDesign design was recently regenerated. This is because the FMSB initialization client's input binary soft mode program image file is more recent than the generated EFC file, so you need to reimport the updated input file.

***Figure 17 •*** **Import Updated Input File**




14. Click on **Modify** > **Import Content** and reimport the soft mode binary memory image file. The Import dialog should open on the correct folder containing the file (that is, <Libero-project-root>\component\work\<SmartDesign-name>\<CoreABC-instance-name>). Click **OK** and then **Finish** to generate the programming file (PDB file). Click **Generate** and, if warned about overwriting a previously generated programming file, accept/confirm this. Once the Programming File button in Designer turns green, exit Designer and return to the Libero IDE Program Flow view.
15. You can now program the device. The program image will be programmed into an NVM block and, at startup time, this image will be used to initialize the soft mode CoreABC instruction RAM blocks. **Note:**If you change your CoreABC configuration or program, you must ensure that the Initialization client configuration matches the details presented in the CoreABC configurator's Analysis view. If you forget to do this, it could result in an incorrectly formatted or incomplete program image being stored or initialized to CoreABC RAM blocks.

## 8.2.3 NVM Mode

With a Fusion AFS device it is possible to set the Instruction Store option to NVM. When this setting is selected, the CoreABC program is stored in an NVM block and the instructions are read directly from NVM during operation.

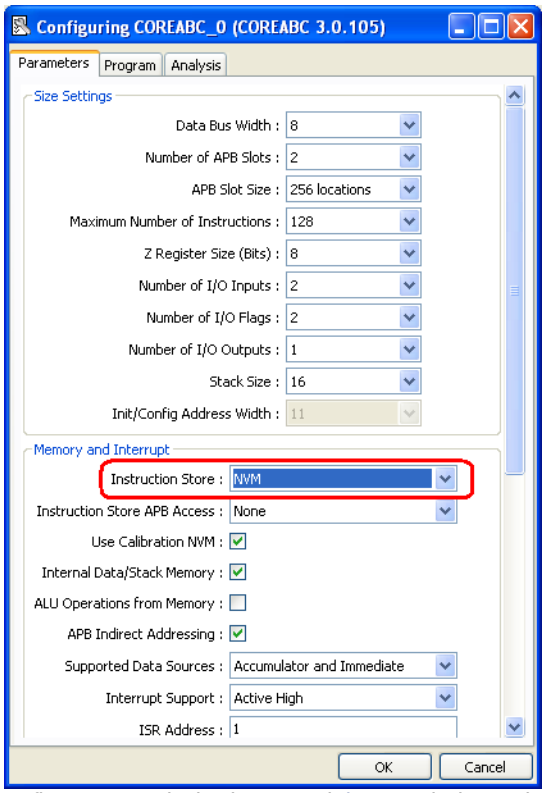
In NVM mode, the instructnvm.v or instructnvm.vhd RTL file instantiates an NVM block within CoreABC. When a design containing an NVM mode CoreABC instance is generated in SmartDesign, memory files are created for initializing the NVM block during simulation and to enable the NVM block to be programmed with the program image during programming of the Fusion AFS device. The simulation-related memory file is automatically placed in the project's simulation folder to facilitate easy simulation. The file which is related to programming of the NVM block has a *.hex suffix and contains information in the Intel Hex format. This file is intended to be used to create a Data Storage client using the Flash Memory System Builder (FMSB) utility.

### 8.2.3.1 NVM Mode Flow on a Fusion AFS Device

The following sequence of steps describes how to implement an NVM mode CoreABC instance on a Fusion AFS device. The steps describe the creation of an FMSB Data Storage client to produce an embedded flash configuration (EFC) file which contributes to the overall programming file for the device. The CoreABC program is effectively contained in this EFC file.

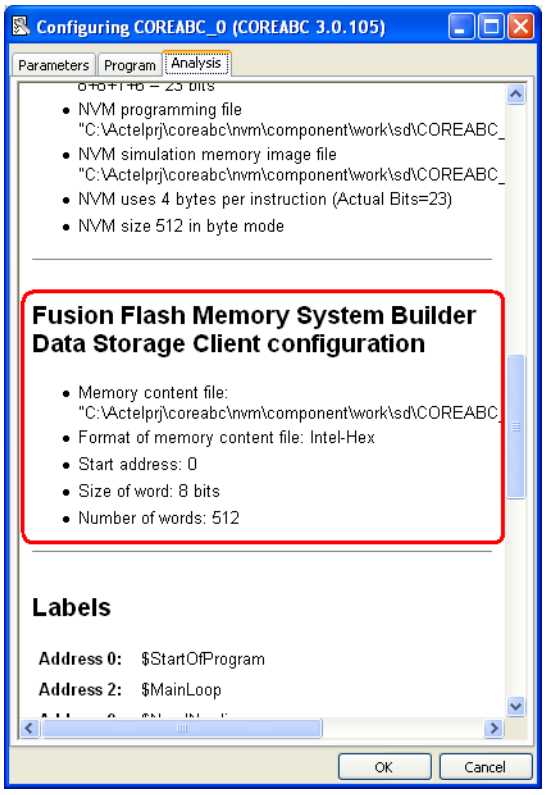
1. Set the **Instruction Store** option to **NVM,** as shown in Figure 18.

***Figure 18 •*** **Instruction Store Option**




2. The CoreABC configurator Analysis view note (Figure 19) shows the configuration details which will be needed when configuring a Fusion AFS Flash Memory System Builder Data Storage Client.

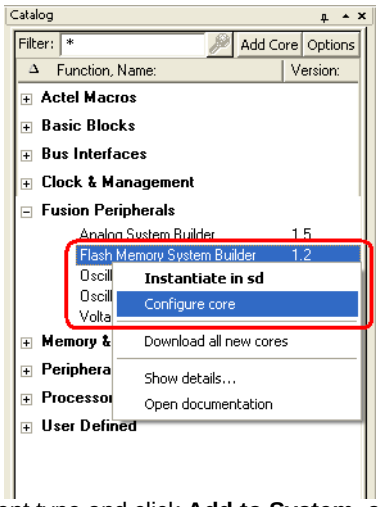
***Figure 19 •*** **Analysis View**




The CoreABC generator also emits a text version of the Analysis view content into a log file (*<Libero-project-root>\component\work\<SmartDesign-name>\<CoreABC-instance- name>\CoreABC.log).* It will appear in the Design Explorer > Files view under Components > [SmartDesign-name] > Report Files > CoreABC.log. This will be used in the following steps when configuring the FMSB Data Storage Client.

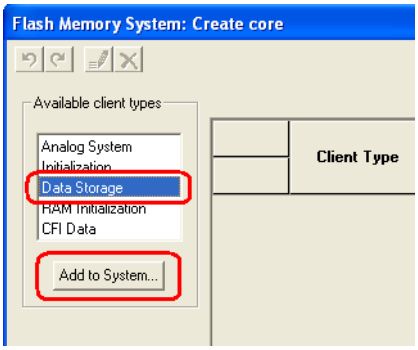
3. **Choose** SmartDesign > Generate Design.
4. Go to **Design Explorer** > **Files** > **Components** > **[SmartDesign-name]** > **Report Files** and open CoreABC.log, which contains the same details as the CoreABC configurator Analysis view. In particular it includes the details required for configuration of the Fusion AFS Flash Memory System Builder Data Storage Client required for the NVM mode CoreABC instance. Scroll down to the Fusion AFS Flash Memory System Builder Data Storage Client configuration section. Select and copy the name of the NVM mode Intel-Hex memory image file. You will paste this into the FMSB Data Storage Client configuration in a subsequent step. Keep the CoreABC.log file open so that it is visible and you can see the other FMSB Data Storage Client configuration details during the next steps.
5. In the Libero IDE Catalog, right-click the **Fusion AFS Peripherals > Flash System Memory Builder** core and choose **Configure core** (Figure 20). It is not necessary to create an FMSB instance (by double-clicking or choosing Instantiate in <SmartDesign-name>), although creating one will not cause a problem.

***Figure 20 •*** **Configure Core**



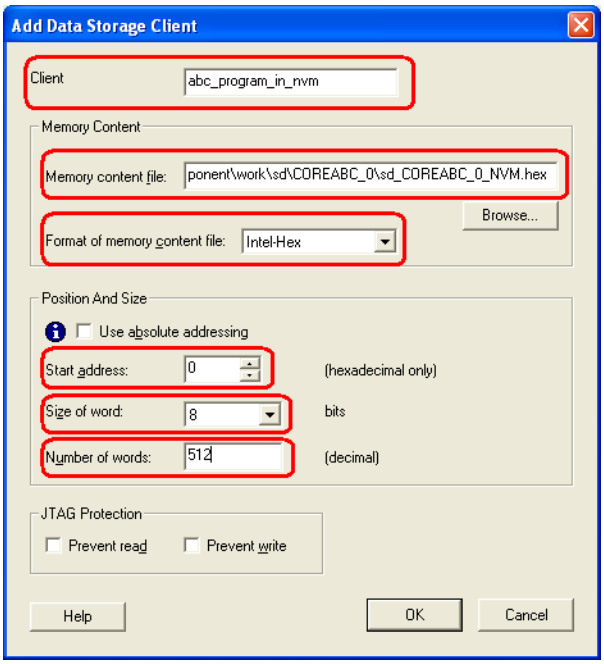

6. Select Data Storage client type and click **Add to System,** as shown in Figure 21.

***Figure 21 •*** **Add Data Storage**



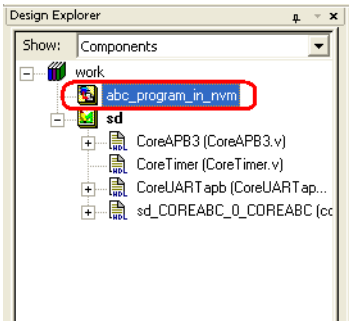

7. Configure the Data Storage Client according to the details displayed in the CoreABC.log file. In particular, paste the NVM memory image file name copied earlier into the Memory content file field and enter a Client name. Configure the **Start address**, **Size of word,** and **Number of words** options (Figure 22).

***Figure 22 •*** **Configure Data Storage Client**



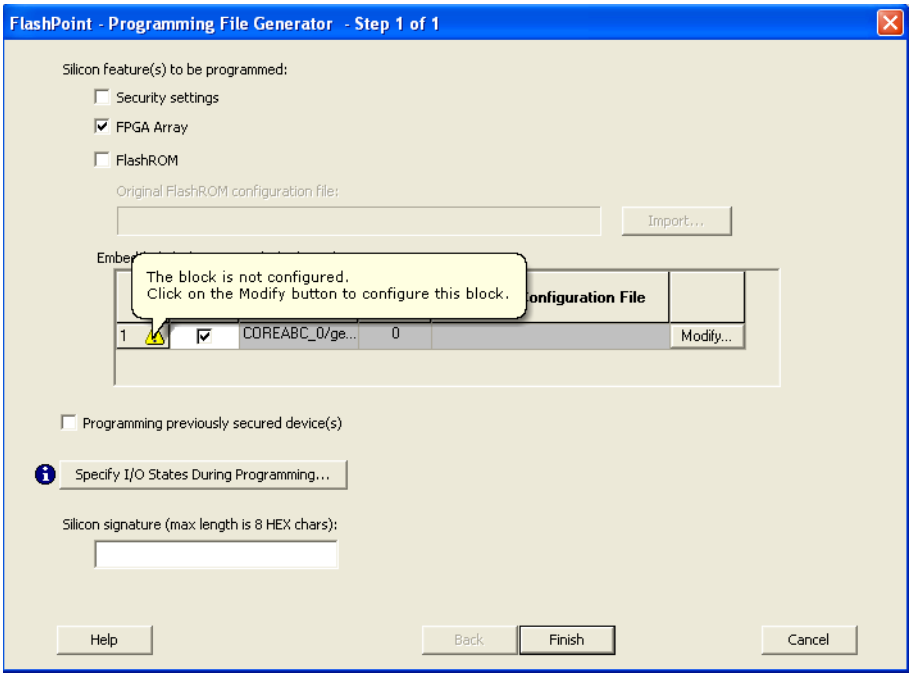

8. Click **OK** and then **Generate.** Name the core when prompted. The configured Fusion AFS Flash Memory System Builder Data Storage Client component should now appear under the Hierarchy tab in your Design Explorer, as shown in Figure 23.

***Figure 23 •*** **Hierarchy Tab in Design Explorer**



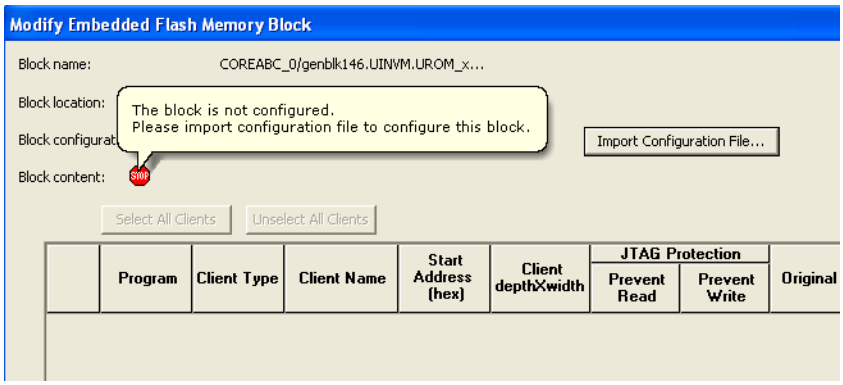

9. Go to the Libero IDE Project Flow view and click **Synplify** to run synthesis.
10. When synthesis has completed, exit Synplify and then click **Place & Route** to run Compile, Layout, and Programming File generation. When you click **Programming File** in Designer to run FlashPoint to generate the programming file (PDB file), you will receive the warning shown in Figure 24.

***Figure 24 •*** **Block Not Configured Warning**




11. In this case it is necessary to update the configuration. Click **Modify** to get the dialog shown in Figure 25.

***Figure 25 •*** **Modify Block Dialog**

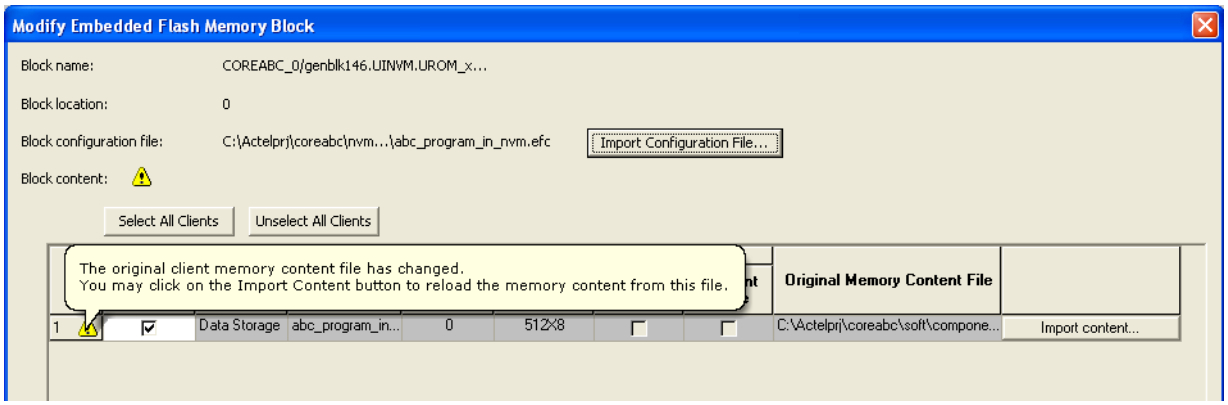



12. Click **Import Configuration File**. Browse to and select the relevant EFC file for the CoreABC NVM mode program image. The EFC file should be in a subfolder of the *<Libero-project- root>\smartgen* folder.
    In this example, the file has the following location:

    <Libero-project-root-folder>\smartgen\abc_program_in_nvm\abc_program_in_nvm.efc.

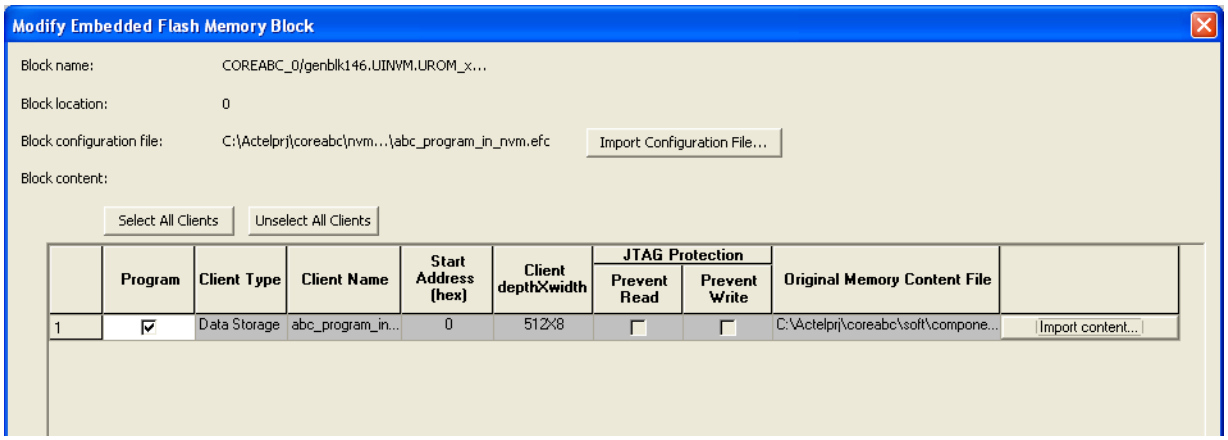
    If the SmartDesign design was regenerated more recently than the FSMB Data Storage Client (which is quite likely), you will receive the warning shown in Figure 26 because the input Intel Hex file is more recent than the generated EFC file.

***Figure 26 •*** **Client Content File Has Changed Warning**




13. Click **Import Content** to import the NVM mode program image Intel Hex file (the Import dialog should open on the correct folder containing this file).
    Once you have done this, the configuration should be up to date, as shown in Figure 27.

***Figure 27 •*** **Configuration Up to Date**




Click **OK**, **Finish**, and then **Generate**. You may be asked to confirm the overwriting of a previously generated programming (PDB) file, in which case confirm/accept this.

14. Once the programming file has been generated, exit Designer and return to the Libero IDE Program Flow view. You can now program the board with your CoreABC NVM mode design.
15. IMPORTANT: Bear in mind that if you change the CoreABC program such that it becomes longer than the size (number of words) previously configured in the Fusion AFS Flash Memory System Builder Data Storage Client component (see step 7), you will need to reconfigure and regenerate the file. For this reason you should always double check the CoreABC Analysis view NVM program details against the currently configured FMSB Data Storage Client configuration to ensure consistency.

### 8.2.3.2 APB Access to Instruction Memory

In NVM mode, it is possible to access the internal NVM block that stores CoreABC's instructions through the APB slave interface. This functionality allows CoreABC to log and retrieve information to and from NVM, for example, while simultaneously running from NVM in cases where only one NVM block is available for use by the CoreABC subsystem. The Instruction Store APB Access configuration option is used to select the type of APB access (if any) to the instruction memory in NVM mode. Possible options are: None, Read Only, or Read/Write.

**Note:** Where read only or read/write access to the instruction memory is required, the APB slave interface which provides access to the instruction memory should ONLY be mastered by CoreABC's APB master interface, typically via CoreAPB3. A separate, independent APB master should not be used to communicate with CoreABC's slave APB interface because this is likely to lead to erroneous behavior. Arbitration between instruction fetches and data type read/write from/to (NVM) instruction memory is deliberately kept as simple as possible to minimize the size of CoreABC.

The APB slave interface provides a register interface for accessing the NVM block. PAGE, SECTOR, and SPARE_PAGE registers together are used to select a 128-byte page to be held in the NVM's page buffer. The page in the buffer can be read and written directly at offset 0x00 to 0x7F in the APB slave interface address space. If writes have been used to modify the contents of the page buffer and the new data is required to be saved in the nonvolatile array of the NVM, the PROGRAM_ENABLE and PROGRAM registers must be written (in that order, using any data) to cause the new page to be programmed to the array. The process of programming the array takes around 8 milliseconds to complete, during which time CoreABC will stall.

It is possible for a CoreABC program to overwrite or corrupt itself when APB read/write access to the instruction memory is enabled in NVM mode. You must take care to avoid this. In practice this usually just means setting the SECTOR, PAGE, and SPARE_PAGE registers in the APB interface to NVM instruction memory to sufficiently high values. That is, read and write data type accesses to the NVM instruction memory should normally be to a region of the NVM above the program which is located from address 0x0000 onwards. Table 19 describes the register interface used to access the internal NVM block using the APB slave interface.

***Table 19 •*** **Address Map of APB Slave Interface, NVM Mode Only**

| Offset | Register Name | R/W | Width | Reset Value | Description |
|---|---|---|---|---|---|
| 0x00 to 0x7F | (This is a range of offsets; see description column for more information.) | R/W | APB_DWIDTH (8, 16, or 32) | – | Any access within this range of offsets accesses offset[6:0] in the page held in the NVM page buffer addressed by {SPARE_PAGE_REG + SECTOR_REG + PAGE_REG}.<br>If APB_DWIDTH = 8, consecutive bytes are at offsets 0x00, 0x01, 0x02, etc.<br>If APB_DWIDTH = 16, consecutive halfwords are at offsets 0x00, 0x02, 0x04, and so on.<br>If APB_DWIDTH = 32, consecutive words are at offsets 0x00, 0x04, 0x08, and so on.<br>The address to the NVM is always a byte address and the lower one or two bits of the address are ignored when the data size is 16 or 32 bits. This means that misaligned addresses are automatically aligned.<br>On an AFS090 device, the data width is restricted to 16 bits when accessing NVM. When APB_DWIDTH is set to 32 in a design targeted at an AFS090 device, APB accesses to the NVM instruction memory will be consistent with the behavior for APB_DWIDTH = 16. That is, only the lowest bit of the (byte) address to the NVM is ignored and only the lower 16 bits of the read and write data buses carry valid data. |
| 0x80 | PAGE_REG | W | 5 | 0x0 | Page of NVM being accessed during APB accesses (to an offset in the range 0x00 to 0x7F). Bits [11:7] of ADDRESS input to NVM block. |
| 0x84 | SECTOR_REG | W | 6 | 0x0 | Sector of NVM being accessed during APB accesses (to an offset in the range 0x00 to 0x7F). |
| 0x88 | SPARE_PAGE_REG | W | 1 | 0x0 | Drives SPAREPAGE input to NVM during APB accesses (to an offset in the range 0x00 to |
| 0x8C | Reserved | - | - | - | - |
| 0x90 | Reserved | | | | - |

***Table 19 •*** **Address Map of APB Slave Interface, NVM Mode Only**

| | | | | | |
|---|---|---|---|---|---|
| 0x94 | PROGRAM_REG | W | 1 | 0x0 | Any write to this register (regardless of the value written) will cause the contents of the page buffer to be programmed to the NVM array, provided the PROGRAM_ENABLE bit is set (see PROGRAM_ENABLE_REG at offset 0x9C). |
| 0x98 | Reserved | - | - | - | - |
| 0x9C | PROGRAM_ENABLE_REG | W | 1 | 0x0 | Any write to this register (regardless of the value written) causes a PROGRAM_ENABLE control bit to be set.<br>This register is cleared by any access (read or write) to any other APB address. This means that the register will be cleared by writing to the PROGRAM_REG. |

# 9 Tool Flows

## 9.1 Licensing

CoreABC is licensed in two ways: Obfuscated and RTL. Tool flow functionality may be limited, depending on your license.

### 9.1.1 Obfuscated

Complete RTL code is provided for the core, enabling the core to be instantiated, configured, and generated within SmartDesign. Simulation, Synthesis, and Layout can be performed with Libero Integrated Design Environment (IDE). The RTL code for the core is obfuscated.

### 9.1.2 RTL

Complete RTL source code is provided for the core.

## 9.2 SmartDesign

CoreABC is available for download to the SmartDesign IP Catalog via the Libero IDE web repository. For information on using SmartDesign to instantiate, configure, connect, and generate cores, refer to the Libero IDE online help.

The APB master interface of CoreABC will typically be connected to the mirrored master interface of CoreAPB3, with various APB slaves connected to the slave interfaces of CoreAPB3.

The core can be configured using the configuration GUI within SmartDesign. See the CoreABC Configuration, page 24 for more details on configuring CoreABC.

## 9.3 Simulation Flows

SmartDesign and Libero IDE facilitate running both a user (or unit) testbench for CoreABC and a basic system testbench for the complete SmartDesign design. You may wish to expand on these simulation capabilities to suit the particular needs of your project. For example, you could make a copy of the system testbench, add additional code to monitor or interact with the design and then use this new testbench as stimulus in a simulation.

To run the CoreABC unit testbench, set the Testbench configuration option to User in the CoreABC configuration GUI before generating the design. After generation, set the design root to be the CoreABC instance and click the Simulation (ModelSim) button. ModelSim will launch and run the unit test.

To run the system testbench for the SmartDesign design, set the design root to be the (SmartDesign) design after generation and again click **Simulation**. ModelSim will launch and run the system simulation.

See Testbench, page 45 for more details on simulation.

## 9.4 Synthesis in Libero IDE

To run synthesis with the configuration selected in the configuration GUI, set the design root appropriately and click the **Synthesis** icon in Libero IDE to launch the Synplicity® synthesis tool. Click the **Run** button in the synthesis window to run synthesis.

## 9.5 Place-and-Route in Libero IDE

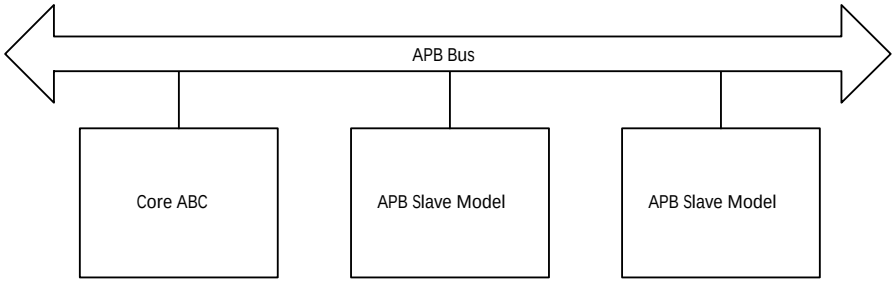
Having set the design route appropriately and run Synthesis, click the **Place & Route** icon in Libero IDE to invoke Designer. CoreABC requires no special place-and-route settings.

# 10 Testbench

## 10.1 Unit Testbench

A unit (or user) testbench is packaged with CoreABC. A block diagram of the testbench is shown in Figure 28. Identical testbenches are supplied for both the VHDL and Verilog versions of the core.

***Figure 28 •*** **CoreABC Verification Testbench**




The CoreABC unit testbench runs a canned program to exercise the core. APB slave models which effectively implement some memory are included in the testbench to allow verification of write and read back operations on the APB interface.

To run the unit testbench, simply set the design root to the CoreABC instance (using right-click, **Set As Root** on the instance name in the Hierarchy tab of the Design Explorer) and click on the **Simulation** (ModelSim®) button in the Project Flow. The unit testbench should automatically launch and run. A "Tests Complete ... OKAY" type message will appear in the simulator transcript window if the simulation is successful.

## 10.2 System Simulation

To simulate a CoreABC based design created in SmartDesign, generate the design and then ensure that the design root is set to the SmartDesign design. During generation of the design, a basic system testbench is created which instantiates the design and provides clock and reset signals to the design. Clicking on the **Simulation** (ModelSim) button will run this testbench. When running the system testbench, CoreABC will execute the program entered in the Program tab of its configuration GUI, rather than a canned program, as is the case when running the CoreABC unit testbench.

By default, the system testbench will run and the clock and reset signals will be displayed in ModelSim's waveform viewer. Often you will want to browse into the design and select other signals to display in the waveform viewer before restarting and rerunning the simulation from within the simulator.

## 10.3 Simulation Logging

CoreABC includes debug code that logs the operations being performed during simulation, along with the current accumulator values. A typical log is shown below.

```
# INS:141: XOR 00 <= 0A XOR 0A Flags:ZERO
# INS:142: JUMP (Not Taken) NOT ZERO
# INS:143: NOP
# INS:144: LOAD 00 <= 00Flags:ZERO
# INS:145: LOADZ <= 5h
```

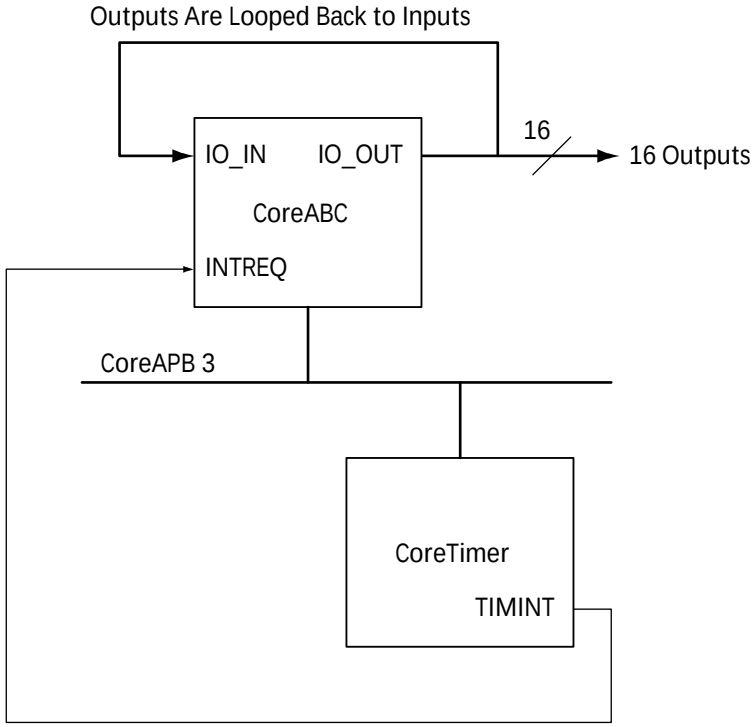
This log starts at instruction 141 and shows the accumulator being XORed with 0x0A, a jump testing the ZERO flag, a NOP instruction, and the accumulator being loaded with 00. Finally, the internal Z register is loaded.

This feature is only available when pre-synthesis simulation is carried out. During synthesis, the debug code is removed from the core. To enable this feature, select the **Verbose Simulation Log** option on the CoreABC configuration GUI.

# 11 Example Design Using CoreABC

This section describes the creation of a simple CoreABC based design. The design uses the general purpose outputs of CoreABC to control eight outputs which may, for example, be used to drive LEDs on a PCB. A "rotating 1"pattern is produced on the outputs and CoreTimer is used to create a delay between pattern changes. CoreAPB3 provides the bus fabric that connects the processor and timer peripheral together. The design is illustrated in Figure 29. In this example, a hard mode CoreABC will be used and the design will be targeted at a Fusion AFS device. Follow the instructions beginning in the "Create a New Project, page 46" to create the example design.

***Figure 29 •*** **Example CoreABC Design**

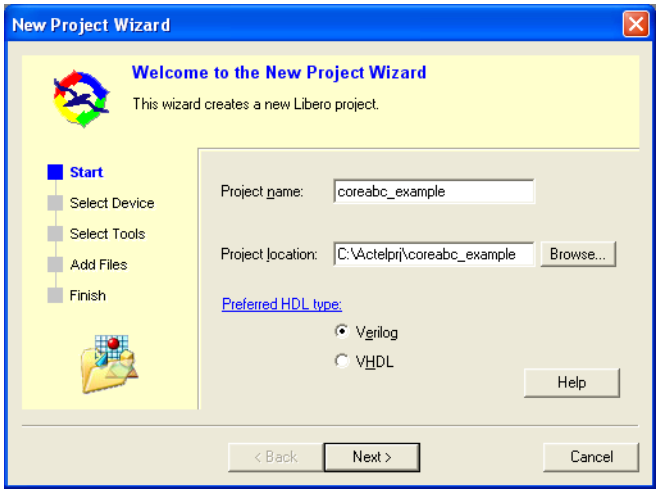



## 11.1 Create a New Project

The first task is to create a new project using the Libero IDE Project Manager. Use the following steps to create the project:

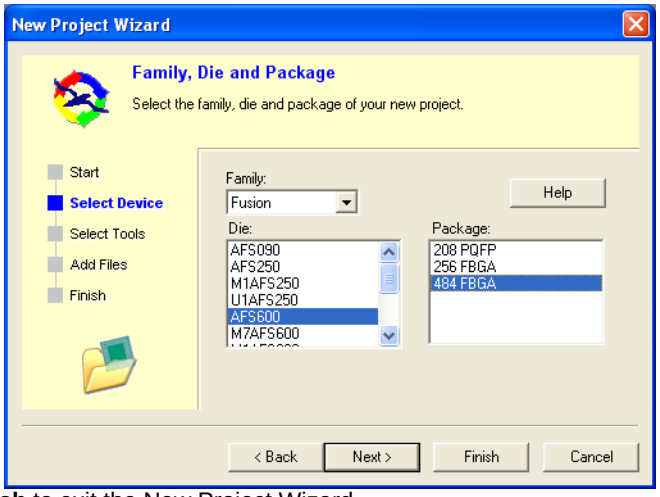
1. Start Project Manager and select **Project** > **New Project.** The New Project Wizard will appear. Enter **coreabc_example** as the project name and select **Verilog** as the preferred HDL type, as shown in Figure 30.

***Figure 30 •*** **New Project Wizard**




2. Click **Next** and on the next screen choose **Fusion AFS** for the Family and select the **AFS600** die and the **484 FBGA** package, as shown in Figure 31.

***Figure 31 •*** **Select Family, Die, and Package**

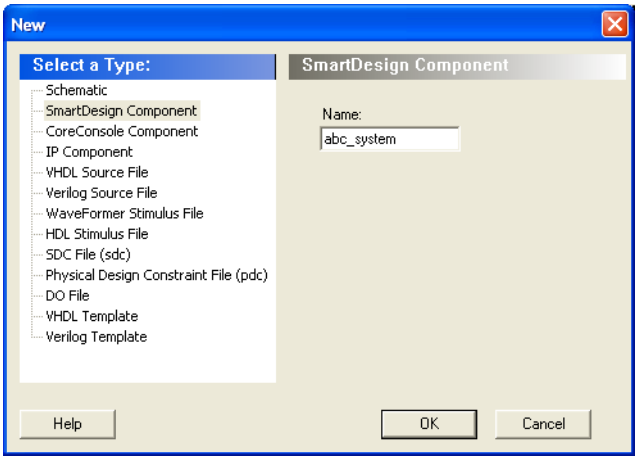



3. Click **Finish** to exit the New Project Wizard.

## 11.2 Create a SmartDesign Design

Click **SmartDesign** in the Project Flow window and enter abc_system as the name of the SmartDesign component to be created, as shown in Figure 32.

***Figure 32 •*** **Name the SmartDesign Component**




4. Click the **OK** button and the SmartDesign canvas for the abc_system will open.

## 11.3 Instantiate, Configure, and Connect the Components

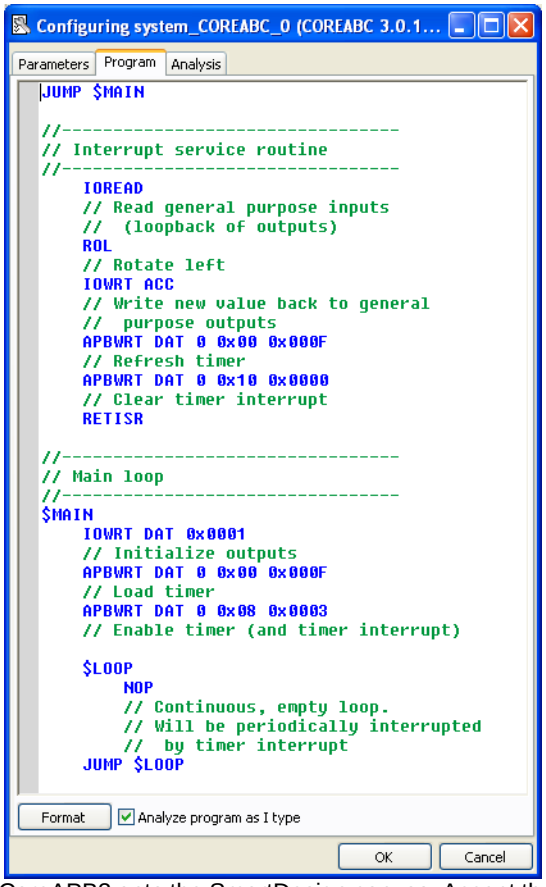
Components can be instantiated on the SmartDesign canvas by dragging and dropping from the Catalog pane on the right hand side of the Project Manager. When a component is dropped onto the canvas, a configuration window will open for that instance of the component. You may need to expand some of the categories in the catalog to see the cores you need. Follow the steps below to instantiate, configure, and connect the components in the design:

1. Drag and drop a CoreABC instance onto the canvas. On the Parameters tab of the CoreABC configuration window, most of the settings can be left at their default values apart from these changes:

- Set Data Bus Width to 16
- Set Number of I/O Inputs to 16
- Set Number of I/O Outputs to 16
- Set Interrupt Support to Active High.

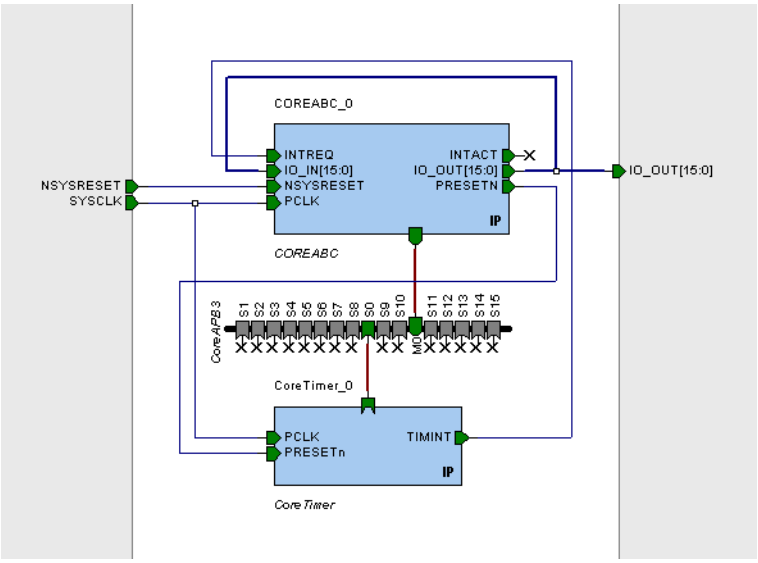
2. On the Program tab of the configuration window, enter the program shown in the screen shot in Figure 33 and then click the **OK** button to dismiss the CoreABC configuration window.

***Figure 33 •*** **Program Tab**




3. Drag and drop CoreAPB3 onto the SmartDesign canvas. Accept the default configuration by clicking **OK** on the CoreAPB3 configuration window. Note that the APB Slot Size settings should always match for CoreABC and CoreAPB3. This setting has a default value of 256 locations on both cores.
4. Drag and drop CoreTimer onto the SmartDesign canvas. In the CoreTimer configuration window, set the Width option to 16 bit and leave the Interrupt active level as High and click **OK**.
5. Choose **SmartDesign** > **Auto Connec**t (or right-click on a blank area of the canvas and select Auto Connect). A window entitled Modify Memory Map will appear, which provides the opportunity to move the timer peripheral to a different slot on the APB3 bus. Accept the default (slot 0) location by clicking the **OK** button. Auto connect will connect the clock, reset, and bus connections.
   Some manual connections must be made as follows. Click on the **TIMINT pin** of CoreTimer and, while holding the CTRL key down on the keyboard, click on the **INTREQ pin** of CoreABC. Right-click on either of these highlighted pins and select **Connect** to connect the two pins together. Right-click on the **IO_OUT[15:0] pin** of CoreABC and select **Promote to Top Level** to connect the outputs to the top level. Next click again on the **IO_OUT[15:0] pin** of CoreABC and, while holding down the CTRL key, also click on the **IO_IN[15:0] pin** of CoreABC. Then right-click on either of these high-lighted pins and select **Connect** to loop the general purpose outputs back to the general purpose inputs. Finally, right-click on each of the unconnected ports and select **Mark Unused** (the uncon-nected ports are INTACT on CoreABC and ports S1 to S15 on CoreAPB3). An X will appear at the end of the open wire connected to each port marked as unused. The design should resemble the one shown in Figure 34.

***Figure 34 •*** **CoreABC Design**




6. Choose **SmartDesign** > **Generate Design** (or right-click on a blank area of the canvas and select Generate Design) to generate the design. If you have omitted marking unconnected ports as unused, an information window mentioning warnings will pop up. If there are any warnings, choose **SmartDesign** > **Check Design Rules** and review the warnings.

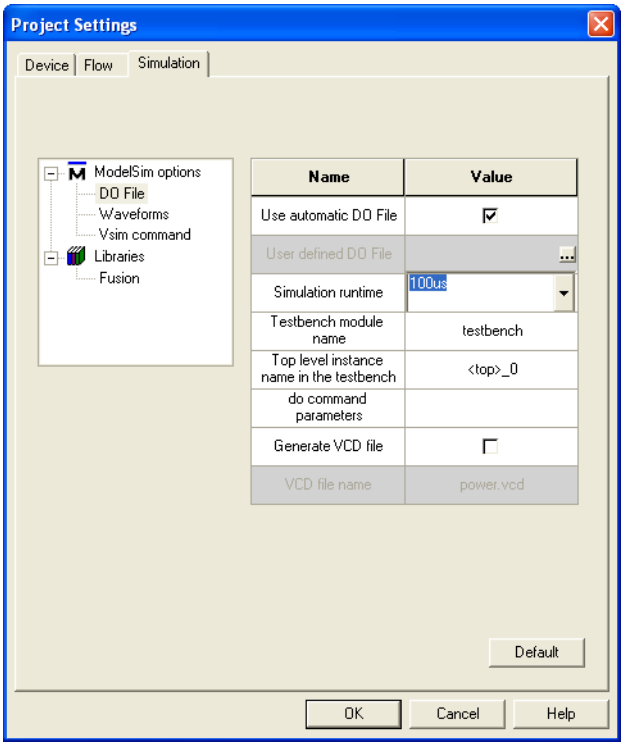
## 11.4 System Simulation

Before running a simulation of the system, we will adjust some of the simulation options.

1. In the Project Flow window, right-click on the Simulation (ModelSim®) button and select **Options**.

A Project Settings window will appear, with the Simulation tab selected. In the left pane visible in the Simulation tab, click on **DO File** under ModelSim options. In the right pane set the Simulation runtime to **100 µs**, shown in Figure 35.

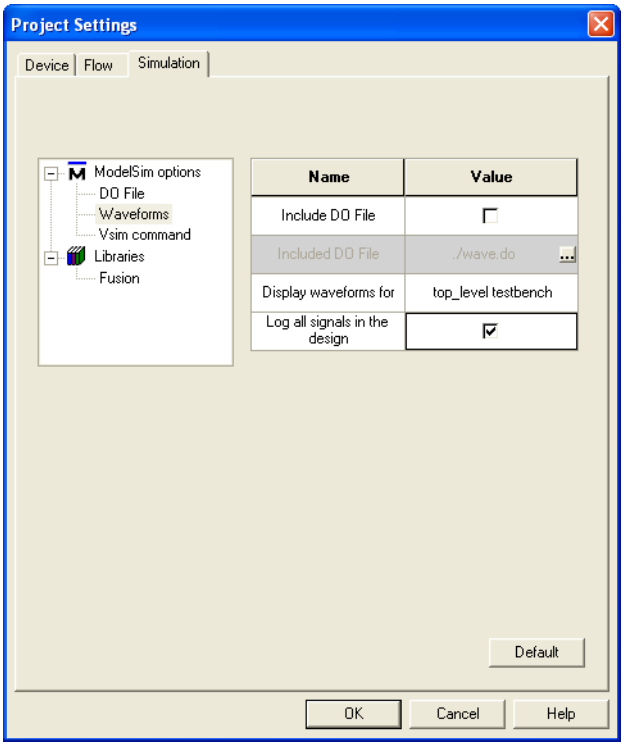
***Figure 35 •*** **Project Settings – Simulation Time**




2. In the left pane, click on Waveforms under ModelSim options and in the right pane click the check box to select Log all signals in the design, as shown in Figure 36.

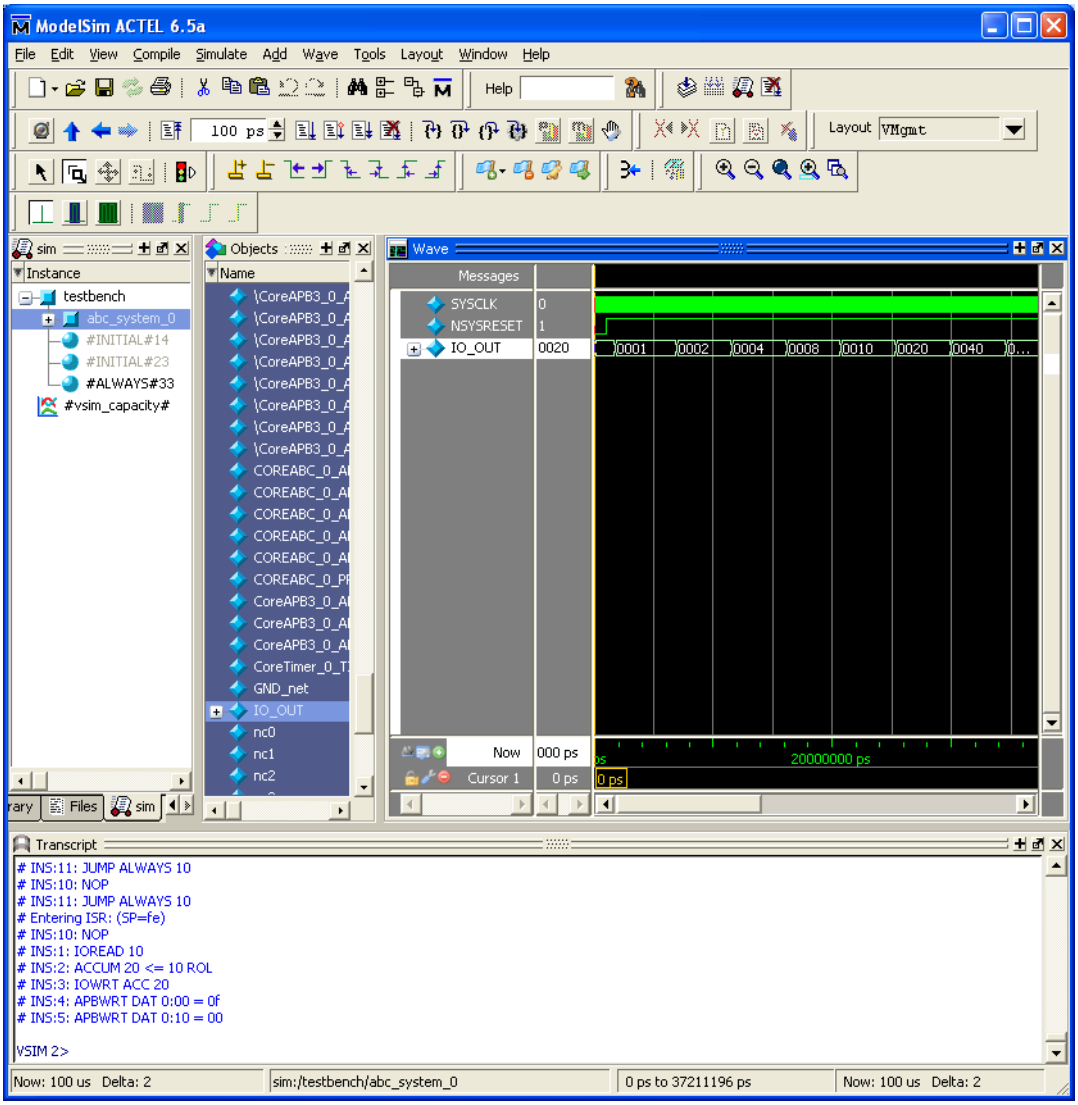
Logging all signals allows signals to be added to the waveform viewer in the simulator after the simulation has completed. For a large design and/or a long simulation run time, it is probably better first to run a short simulation and then add the signals of interest to the waveform viewer. The waveform format would then be saved to a DO file (typically named wave.do) and, in the Waveforms options window, you would click the **Include DO File** option and enter the appropriate filename for the Included DO File value. The **Log all signals in the design** option would be deselected.

***Figure 36 •*** **Simulation Settings**




3. Click **OK** to dismiss the window.
4. Back in the Project Flow window, click **Simulation** (ModelSim) to launch the simulation. The simulator will launch and run and, by default, all testbench signals will be displayed in the waveform viewer.
   The testbench automatically created for this design, when the design was generated in SmartDesign, contains only clock and reset signals and these are displayed in the ModelSim Wave window (waveform viewer) with their exact names of SYSCLK and NSYSRESET. The IO_OUT output from CoreABC is also of interest in this design. We should be able to observe the moving 1 pattern on this port.
5. To view IO_OUT in the Wave window, click on the abc_system instance name (which should be abc_system_0 by default) in the simulation window. After doing this, the Objects window will list all of the signals present in abc_system. Scroll to the IO_OUT signal in the Objects window and drag and drop this onto the **Wave** window. It should be possible to observe the moving 1 pattern on the IO_OUT trace. It may be easier to see the pattern by viewing IO_OUT in hexadecimal form. To do this, right-click the **IO_OUT** signal in the Wave window and select **Radix** > **Hexadecimal**. Figure 37 illustrates what should be observed when IO_OUT is displayed as a hexadecimal signal.

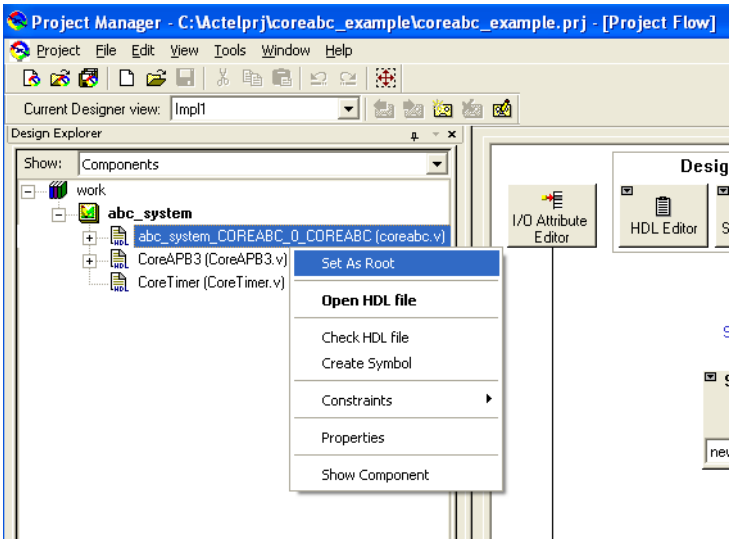
***Figure 37 •*** **ModelSim Simulation Showing IO_OUT Waveform**




## 11.5 Simulation of CoreABC Only (unit test)

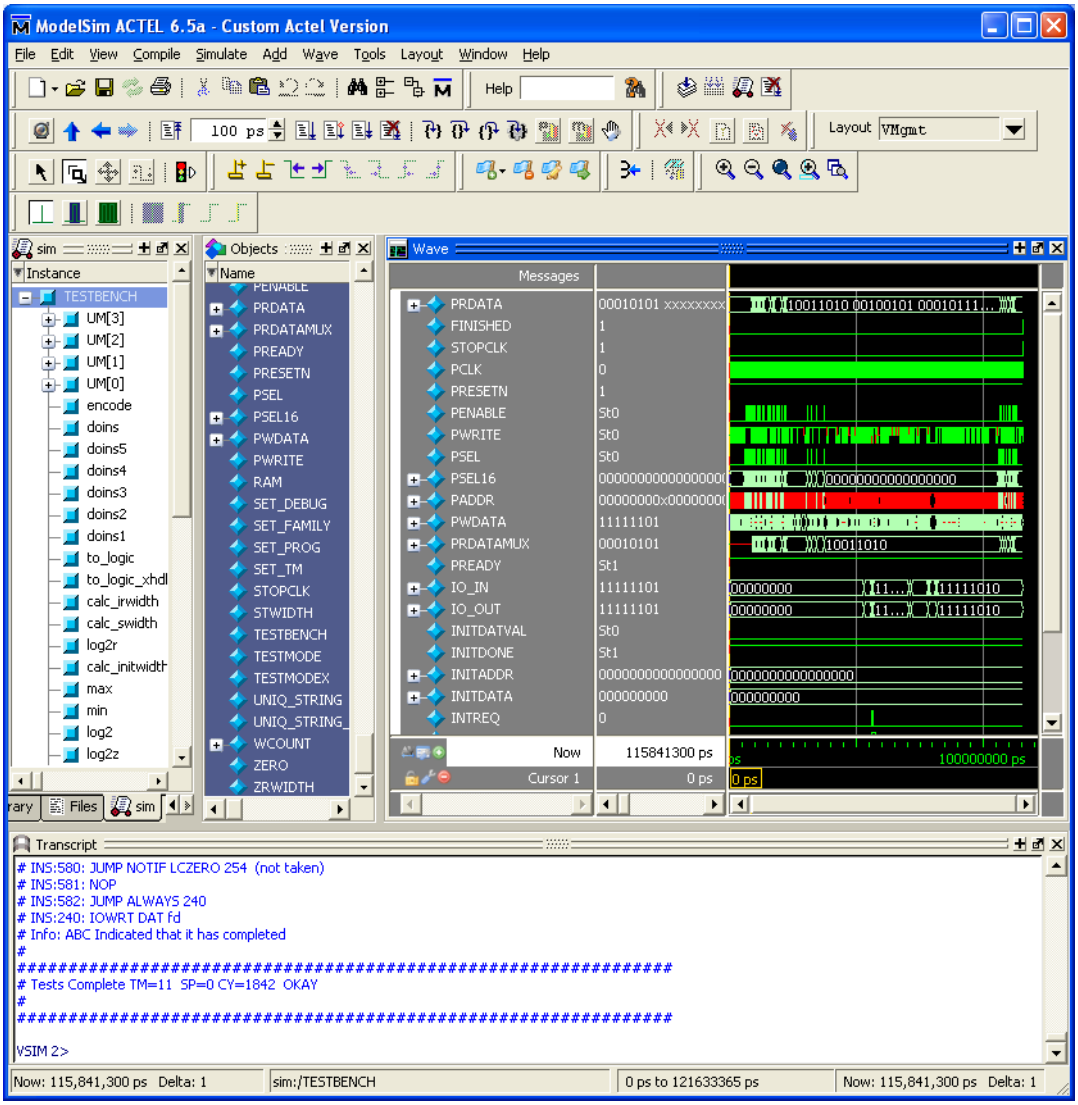
As well as running a system simulation, it is also possible to run a unit test on CoreABC only. To do this, ensure that the Testbench configuration option for CoreABC is set to User (which is the default setting) before generating the design in SmartDesign. In the Hierarchy tab of the Design Explorer window of Project Manager, browse to the CoreABC instance. Right-click on the instance and select **Set As Root**, as illustrated in Figure 38.

***Figure 38 •*** **Set As Root**




With the CoreABC instance set as the design root, click the **Simulation** button. ModelSim will launch and automatically run the CoreABC unit testbench. A "Tests Complete OKAY" type message will be displayed in the ModelSim transcript window on successful completion of the testbench, as shown in Figure 39.

***Figure 39 •*** **ModelSim Simulation Window**




## 11.6 Synthesis

To synthesize the design, first ensure that the design root is set to the top level of the design, which is abc_system. The design root may have changed if, for example, you ran a CoreABC unit test as described in the Simulation of CoreABC Only (unit test), page 53. Click the **Synthesis** button in the Project Flow window to launch the Synplify synthesis tool. Click **Run** to run synthesis.

## 11.7 Place-and-Route

To run place-and-route, click the **Place&Route** button in the Project Flow window to launch the Designer tool. Some dialog windows will be displayed as Designer starts. Enter appropriate information in these windows—normally the default entries can be accepted by clicking the **OK** button on each window. In Designer, click the **Compile** button to run the compile stage. If you intend to implement the design on a real board, you will need to make some pin assignments to suit the target board. One way of doing this is to use the **I/O Attribute Editor** (by clicking on the button of the same name) after compile has completed. After compiling and making any necessary pin assignments, click the Layout button to run the layout stage. After layout has completed, a programming file can be created by clicking the **Programming File** button and clicking **OK** to the subsequent windows which pop up after making any necessary edits to the information presented in these windows.

# 12 CoreABC v2.3 Migration Guide

Migrating an existing design which uses CoreABC v2.3 to one which uses CoreABC v3.0 or later involves a number of steps. CoreABC v2.3 required the CoreConsole tool to either create a complete CoreABC based design or to create a CoreABC component (essentially a wrapped CoreABC instance) which would typically be instantiated in a SmartDesign design. CoreABC v3.0 or later can be instantiated natively in a SmartDesign design and does not require the CoreConsole tool at all.

A key difference to be aware of between CoreABC v2.3 and CoreABC v3.0 or later is that the CoreABC v2.3 is designed for use with CoreAPB whereas CoreABC v3.0 or later must be used with CoreAPB3.

Follow these steps to migrate a design using CoreABC v2.3 to one using CoreABC v3.0 or later:

1. Open the original CoreABC v2.3 based design in CoreConsole.
2. Note/record the CoreABC configuration settings and make a copy of the program code.
3. Delete the CoreABC instance from the design.
4. Save and generate the design minus the CoreABC instance. It may be necessary to make some stitching/connection changes at this point to allow the design to be generated without the CoreABC instance in place. For example, you may need to tie off some inputs to other cores which were previously driven by outputs from CoreABC.
5. Import the generated design into Libero IDE / SmartDesign and, when prompted, allow the tool to convert the design from a CoreConsole design to a SmartDesign design.
6. Open the SmartDesign design.
7. If in the original design CoreABC v2.3 was used to master CoreAPB, replace CoreAPB with CoreAPB3.
8. Instantiate CoreABC v3.0 or later and apply the original configurations and program code from Step 2.
9. Connect and generate the design.

# 13 Example Instruction Sequence

The following shows an example instruction sequence that uses CoreABC to control CoreAI, to detect whether a voltage source is within a range.

```
// Sample code that reads an analog input and sets an output depending on a
threshold DEF
    ACM_SIZE 90
    DEF ADC_STAT_HI_ADDR 0x11
    DEF ACM_CTRLSTAT 0x0
    DEF ACM_DATA_ADDR 0x04
    DEF ACM_ADDR_ADDR 0x02

DEF ADC_CTRL2_HI_ADDR 0x09

// Set up UART and put out welcome 115200 baud assuming 50 MHz clock
$RESET
    APBWRT DAT8 1 8 27
    APBWRT DAT8 1 12 1

$WelcomeMessage
    WAIT UNTIL INPUT0 APBWRT
    DAT8 1 0 'O' WAIT UNTIL
    INPUT0 APBWRT DAT8 1 0 'K'
    WAIT UNTIL INPUT0 APBWRT
    DAT8 1 0 10 WAIT UNTIL INPUT0
    APBWRT DAT8 1 0 13

// Set up core AI
// Reset ACM
    WAIT WHILE INPUT1
    APBWRT DAT8 0
    ACM_CTRLSTAT 1 WAIT
    WHILE INPUT1

// Wait until calibrated
$WaitCalibrate
    APBREAD 0
    ADC_STAT_HI_ADDR AND
    0x8000
    JUMP IFNOT ZERO
    $WaitCalibrate

// Program AV, AC, AT, AG
    registers LOAD 0
$WaitRegProg
    WAIT WHILE INPUT1
    APBWRT      ACC      0
    ACM_ADDR_ADDR  APBWRT
```

```
    ACM  0  ACM_DATA_ADDR
    ADD 1
    CMP ACM_SIZE
    JUMP IFNOT ZERO $WaitRegProg

// Wait for ADC
    calibrated WAIT
    WHILE INPUT1
    IOWRT 1

// Now get the POT value, which is on AC5 = Ch17 0x11
// Also mask bits
$mainloop
    APBWRT DAT16 0 ADC_CTRL2_HI_ADDR
    0x1100 WAIT WHILE INPUT0
    APBREAD 0 ADC_STAT_HI_ADDR
    AND 0x0FFF
// Got the value in the accumalator, store in RAM in 1 mV
    value SHL0
    SHL0
    RAMWRT 0
// Now generate BCD value
    LOAD 0
    RAMWRT 11
    RAMWRT 12
    RAMWRT 13
// 0 = Value; 11-14 is BCD value
$BCD1
    SUB 1000
    JUMP IF NEGATIVE $BCD2
    PUSH
    RAMREAD 11
    INC
    RAMWRT 11
    POP
    JUMP $BCD1
$BCD2
    ADD 1000
$BCD3
    SUB 100
    JUMP IF NEGATIVE $BCD4
    PUSH
    RAMREAD 12
    INC
    RAMWRT 12
    POP
    JUMP $BCD3
$BCD4
    ADD 100
$BCD5
```

```
    SUB 10
    JUMP IF NEGATIVE $BCD6
    PUSH
    RAMREAD 13
    INC
    RAMWRT 13
    POP
    JUMP $BCD5
$BCD6
    ADD 10
    RAMWRT 14

// BCD value is now in memory; send to UART
$valueToUart
    WAIT UNTIL INPUT0
    RAMREAD 14
    ADD 0x30
    APBWRT ACC 1 0
    WAIT UNTIL INPUT0
    APBWRT DAT8 1 0
    '.' WAIT UNTIL
    INPUT0 RAMREAD 13
    ADD 0x30
    APBWRT ACC 1 0
    WAIT UNTIL INPUT0
    RAMREAD 12
    ADD 0x30
    APBWRT ACC 1 0
    WAIT UNTIL INPUT0
    RAMREAD 11
    ADD 0x30
    APBWRT ACC 1 0
    WAIT UNTIL INPUT0
    APBWRT DAT8 1 0
    'V' WAIT UNTIL
    INPUT0 APBWRT
    DAT8 0 0 10 WAIT
    UNTIL INPUT0
    APBWRT DAT8 0 0
    13
    JUMP $mainloop
```

This sequence allows CoreABC to initialize CoreAI and then sample an ADC channel, converting the value to BCD (binary coded decimal) and transmitting the value using CoreUART. In this case, the BUSY output from CoreAI is connected to the IO_IN(0) input of CoreABC.

The following shows a simple example instruction sequence that uses CoreABC to write and read to the MSS peripherals in indirect addressing mode without Z registers.

```
JUMP $MAIN
$MAIN
$LOOP
APBWRT DAT 1 0x0
0x20000000
APBWRT DAT 0
0x0008 0xAB
APBREAD 0 0x0008
IOWRT ACC
JUMP $LOOP
```

The following shows a simple example instruction sequence that uses CoreABC to write and read to the MSS peripherals in indirect addressing mode with Z registers.

```
JUMP $MAIN
$MAIN
$LOOP
APBWRT DAT 1 0x0
0x20000000
LOADZ DAT 0x0008
APBWRTZ DAT 0
0xAA
APBREADZ 0
IOWRT ACC
JUMP $LOOP
```

# 14 Instruction Summary

This section details all the CoreABC instructions. The encoding can be found in Table 20.

## 14.1 Instructions

### 14.1.1 NOP

**Operation**

No operation

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.2 LOAD DAT Data

**Operation**

Load accumulator with immediate data value.

**Flags**

ZERO: Set if value is zero. NEGATIVE: Set if value is negative.

**Clock Cycles**

3

### 14.1.3 LOAD RAM Address

**Operation**

Load accumulator with RAM location.

**Flags**

ZERO: Set if value is zero. NEGATIVE: Set if value is negative.

**Clock Cycles**

3

### 14.1.4 INC

**Operation**

Increment the accumulator.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.5 AND DAT Data

**Operation**

AND the accumulator with the immediate data value.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.6 AND RAM Address

**Operation**

AND the accumulator with the RAM location.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.7 OR DAT *Data*

**Operation**

OR the accumulator with the immediate data value.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.8 OR RAM Address

**Operation**

OR the accumulator with the RAM location.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.9 XOR DAT *Data*

**Operation**

XOR the accumulator with the immediate data value.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.10 XOR RAM *Address*

**Operation**

XOR the accumulator with the RAM location.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.11 ADD DAT *Data*

**Operation**

ADD the immediate data value to the accumulator.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.12 ADD RAM *Address*

**Operation**

ADD the RAM location to the accumulator.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.13 SUB DAT *Data*

**Operation**

Subtract the immediate data value from the accumulator.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.14 SHL0

**Operation**

Shift the accumulator left; LSB <= 0.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.15 SHR0

**Operation**

Shift the accumulator right; MSB <= 0.

**Flags**

ZERO: Set if resultant value is zero.

NEGATIVE: Set if resultant value is negative (*not set*).

**Clock Cycles**

3

### 14.1.16 SHL1

**Operation**

Shift the accumulator left; LSB <= 1.

**Flags**

ZERO: Set if resultant value is zero (*not set*). NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.17 SHR1

**Operation**

Shift the accumulator right; MSB <= 1.

**Flags**

ZERO: Set if resultant value is zero (*not set*). NEGATIVE: Set if resultant value is negative (*set*).

**Clock Cycles**

3

### 14.1.18 SHLE

**Operation**

Shift the accumulator left; LSB <= LSB.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.19 SHRE

**Operation**

Shift the accumulator right; MSB <= MSB.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.20 ROL

**Operation**

Rotate the accumulator left; LSB <= MSB.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.21 ROR

**Operation**

Rotate the accumulator right; MSB <= LSB.

**Flags**

ZERO: Set if resultant value is zero. NEGATIVE: Set if resultant value is negative.

**Clock Cycles**

3

### 14.1.22 CMP DAT *Data*

**Operation**

Compare the accumulator with the immediate data value. Uses Boolean AND.

**Flags**

ZERO: Set if values are equal. NEGATIVE: Set if both MSBs are set.

**Clock Cycles**

3

### 14.1.23 CMP RAM *Address*

**Operation**

Compare the accumulator with the RAM location. Uses Boolean AND.

**Flags**

ZERO: Set if values are equal. NEGATIVE: Set if both MSBs are set.

**Clock Cycles**

3

### 14.1.24 CMPLEQ DAT *Data*

**Operation**

Compare the accumulator with the immediate data value. Uses subtract operation.

**Flags**

ZERO: Set if values are equal.

NEGATIVE: Set if accumulator is less than the data value.

**Clock Cycles**

3

### 14.1.25 BITCLR *N*

**Operation**

Clear accumulator bit *N.* Uses Boolean AND.

**Flags**

ZERO: Set if resultant accumulator value is zero. NEGATIVE: Set if resultant accumulator value is negative.

**Clock Cycles**

3

### 14.1.26 BITSET *N*

**Operation**

Set accumulator bit *N.* Uses Boolean OR.

**Flags**

ZERO: Set if resultant accumulator value is zero *(not set*). NEGATIVE: Set if resultant accumulator value is negative.

**Clock Cycles**

3

### 14.1.27 BITTST *N*

**Operation**

Tests accumulator bit *N.* Uses Boolean AND.

**Flags**

ZERO: Set if the bit is zero. NEGATIVE: Undefined

**Clock Cycles**

3

### 14.1.28 APBREAD Slot Address

**Operation**

Reads the APB from the specified slot and address, and stores the value in the accumulator.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.29 APBWRT ACC Slot Address

**Operation**

Writes the accumulator to the APB at the specified slot and address.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.30 APBWRT ACM Slot Address

**Operation**

Writes the value in the ACM table indexed by the accumulator to the APB at the specified slot and address.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.31 APBWRT DAT Slot Address Data

**Operation**

Writes the data value to the APB at the specified slot and address.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.32 APBWRT DAT8 Slot Address Data

**Operation**

Writes only the lowest eight bits of the data value to the APB at the specified slot and address. Specifying DAT8 rather than DAT may reduce tile count when AHB_DWIDTH ≥ 16.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.33 APBWRT DAT16 Slot Address Data

**Operation**

Writes only the lowest 16 bits of the data value to the APB at the specified slot and address. Specifying DAT16 rather than DAT may reduce tile count when AHB_DWIDTH = 32.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.34 APBREADZ *Slot*

**Operation**

Reads the APB from the specified slot and address, and stores the value in the accumulator. The Z register is used as the APB address.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.35 APBWRTZ ACC *Slot*

**Operation**

Writes the accumulator to the APB at the specified slot and address. The Z register is used as the APB address.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.36 APBWRTZ ACM *Slot*

**Operation**

Writes the value in the ACM table indexed by the accumulator to the APB at the specified slot and address. The Z register is used as the APB address.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.37 APBWRTZ DAT *Slot Data*

**Operation**

Writes the data value to the APB at the specified slot and address. The Z register is used as the APB address.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.38 APBWRTZ DAT8 *Slot Data*

**Operation**

Writes only the lowest eight bits of the data value to the APB at the slot and address pointed to by the Z register. Specifying DAT8 rather than DAT may reduce tile count when AHB_DWIDTH ≥ 16. The Z register is used as the APB address.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.39 APBWRTZ DAT16 *Slot Data*

**Operation**

Writes only the lowest 16 bits of the data value to the APB at the specified slot and address. Specifying DAT16 rather than DAT may reduce tile count when AHB_DWIDTH = 32. The Z register is used as the APB address.

**Flags**

Unchanged

**Clock Cycles**

5 plus any additional cycles caused by PREADY

### 14.1.40 LOADZ DAT *Data*

**Operation**

Loads the Z register with immediate data value.

**Flags**

ZZERO: Set if value is zero.

**Clock Cycles**

3

### 14.1.41 DECZ

**Operation**

Decrements the Z register.

**Flags**

ZZERO: Set if the Z register decrements to zero.

**Clock Cycles**

3

### 14.1.42 INCZ

**Operation**

Increments the Z register.

**Flags**

ZZERO: Set if the Z register Increments to zero.

**Clock Cycles**

3

### 14.1.43 ADDZ Data

**Operation**

Adds Data to the Z register.

**Flags**

ZZERO: Set if the resultant Z register value is zero.

**Clock Cycles**

3

### 14.1.44 IOREAD

**Operation**

Load the IO_IN port value into the accumulator.

**Flags**

Updated

**Clock Cycles**

3

### 14.1.45 IOWRT DAT *Data*

**Operation**

Writes the data value to the I/O register that drives the IO_OUT top-level port.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.46 IOWRT *ACC*

**Operation**

Writes the accumulator to the I/O register that drives the IO_OUT top-level port.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.47 RAMREAD *Address*

**Operation**

Loads the accumulator with the value stored at the specified address in the internal memory.

**Flags**

ZERO: Set if read value is zero. NEGATIVE: Set if read value is negative.

**Clock Cycles**

3

### 14.1.48 RAMWRT Address ACC

**Operation**

Writes the accumulator to the specified address in the internal memory.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.49 RAMWRT Address DAT Data

**Operation**

Writes the data value to the specified address in the internal memory.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.50 POP

**Operation**

Decrements the stack pointer and then loads the accumulator with the internal memory location addressed by the stack pointer.

**Flags**

ZERO: Set if read value is zero. NEGATIVE: Set if read value is negative.

**Clock Cycles**

3

### 14.1.51 PUSH DAT *Data*

**Operation**

Writes the immediate data to the internal memory location addressed by the stack pointer and then decrements the stack pointer.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.52 PUSH ACC

**Operation**

Writes the accumulator to the internal memory location addressed by the stack pointer and then decrements the stack pointer.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.53 JUMP Address

**Operation**

Jumps always to specified instruction address.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.54 JUMP IF|IFNOT Condition Address

**Operation**

Jumps on or not on condition to specified instruction address. Conditions are specified in Table 20.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.55 CALL Address

**Operation**

Jumps always to specified instruction address. The following instruction address is pushed onto the stack and the stack pointer decremented.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.56 CALL IF|IFNOT Condition Address

**Operation**

Jumps on or not on condition to specified instruction address. The following instruction address is pushed onto the stack and the stack pointer decremented. Conditions are specified in Table 20.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.57 RETURN

**Operation**

Jumps to the instruction address read from the stack. The stack pointer is incremented.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.58 RETURN IF|IFNOT *Condition*

**Operation**

Jumps on or not on condition to the instruction address read from the stack. The stack pointer is incremented. Conditions are specified in Table 20.

**Flags**

Unchanged

**Clock Cycles**

3

### 14.1.59 RETISR

**Operation**

Jumps to the instruction address read from the stack. The stack pointer is incremented. The INTACT output is deactivated.

**Flags**

Restored to the values preceding the interrupt.

**Clock Cycles**

3

### 14.1.60 RETURN IF|IFNOT *Condition*

**Operation**

Jumps on or not on condition to the instruction address read from the stack. The stack pointer is incremented. The internal INTACT output is deactivated. Conditions are specified below.

**Flags**

Restored to the values preceding the interrupt.

**Clock Cycles**

3

### 14.1.61 WAIT UNTIL|WHILE *Condition*

**Operation**

Wait at the current instruction until or while a condition is true. Conditions are specified below.

**Flags**

Unchanged

**Clock Cycles**

3 to ∞

### 14.1.62 HALT

**Operation**

Halt

**Flags**

Unchanged

**Clock Cycles**

¥

### 14.1.63 Condition Codes

The conditions codes are shown in Table 20.

***Table 20 •*** **Condition Codes**

| Condition | Encoding | Description |
|---|---|---|
| ALWAYS | 0x01 | Always |
| ZERO | 0x02 | Accumulator zero |
| NEGATIVE | 0x04 | Accumulator negative |
| ZZERO | 0x08 | Z register zero |
| INPUT0 | 0x010 | Input0 set |
| INPUT1 | 0x020 | Input1 set and similarly for higher inputs, if available |
| LTE_ZERO | 0x06 | Less than or equal to zero; the combination NEGATIVE OR ZERO |