# CoreABC Handbook

*v3.0*

Actel®

**Actel Corporation, Mountain View, CA 94043**



Printed in the United States of America

Part Number: 50200085-2

Release: April 2007



## Trademarks

# Table of Contents


Introduction . . . 5
CoreABC Overview . . . 5
Core Version . . . 5
Supported Interfaces . . . 5
Supported Tool Flows . . . 5
Utilization and Performance . . . 6

1 Internal Architecture . . . 9
Advanced Peripheral Bus . . . 10
Soft Configuration—RAM-Based Operation . . . 11

2 Tool Flows . . . 13
Licenses . . . 13
CoreConsole . . . 13
Importing into Libero IDE . . . 15
Simulation Flows . . . 15
Synthesis in Libero IDE . . . 15
Place-and-Route in Libero IDE . . . 15

3 Interface Descriptions . . . 17
Interface Descriptions . . . 17
Parameters . . . 17
Ports . . . 21

4 CoreABC Programmer's Model . . . 23
Address Spaces . . . 23
Flags Register—Inputs and Condition Codes . . . 24
Instruction Set . . . 24

5 CoreABC Operation . . . 29
Instruction Encoding . . . 29
ACM Lookup for Use with CoreAI . . . 32
Stack . . . 33
Interrupt Operation . . . 33
User Instructions . . . 34
Simulation Logging . . . 34

6 CoreABC Configuration . . . 35
Cross-Validation of Configuration Fields . . . 38

7 CoreABC Programming . . . 39
Analysis . . . 39
Modifying the RTL Code Directly . . . 40

Soft Operation—Creating the Programming File . . . . . 42

8 Testbench Operation . . . . . 45
Basic Testbench . . . . . 45
Verification Tests . . . . . 46

A Example Instruction Sequence . . . . . 47

B Adding User Instructions . . . . . 51

C Instruction Summary . . . . . 53
Condition Codes . . . . . 67

D List of Document Changes . . . . . 69

E Product Support . . . . . 71
Customer Service . . . . . 71
Actel Customer Technical Support Center . . . . . 71
Actel Technical Support . . . . . 71
Website . . . . . 71
Contacting the Customer Technical Support Center . . . . . 71

Index . . . . . 73

# Introduction

## CoreABC Overview

CoreABC provides a simple, low-gate-count controller for the Advanced Peripheral Bus (APB) devices used within the CoreConsole IP Deployment Platform (IDP). In particular, it is targeted at controlling CoreAI and CorePWM in a Fusion application. 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.

CoreABC provides a controller for the APB that is easily configured to read and write from the APB and do basic processing. CoreABC can be configured as either "hard" or "soft."

The instructions executed are either held in a small internal ROM constructed from logic tiles ("hard" configuration) or stored in RAM blocks internal to CoreABC ("soft" configuration). The RAM blocks can initialized using the embedded flash memory within the Fusion family or another external source (for example, CoreMP7).

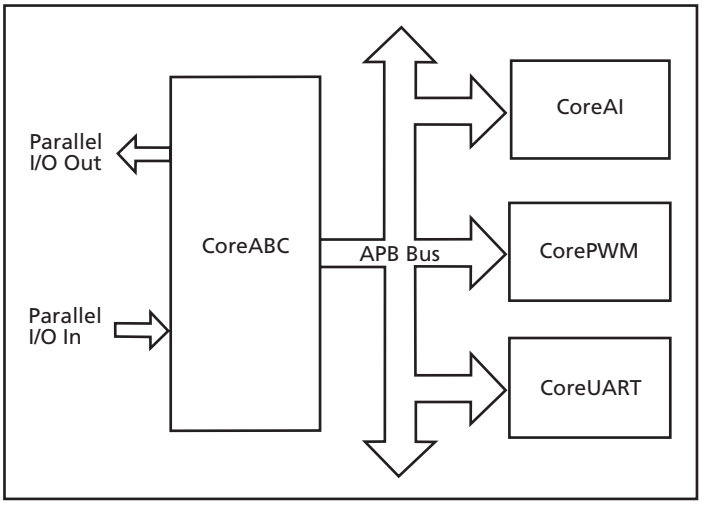


Figure 1 · Typical CoreABC System

## Core Version

This handbook supports CoreABC v2.3.

## Supported Interfaces

CoreABC is available with the APB interface, which is described in "Interface Descriptions" on page 17.

## Supported Tool Flows

CoreABC requires CoreConsole v1.2.1 and Actel Libero® Integrated Design Environment (IDE) v7.3 or later. Additionally, Verilog users MUST use Synplicity® v8.6.1 or later, which is downloadable from www.synplicity.com.

# Utilization and Performance

CoreABC utilization varies depending on how it is configured. Table 1 below and Table 2 on page 7 provide typical utilization data for a range of configurations; these configurations are listed in Table 3-1 on page 17. CoreABC can be implemented in several Actel 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 | 175 | 43 | 0 | 230 | AFS600 A3P600 AGL600 | 1.7% | 94 |
| ProASIC | 8 | Small | 190 | 47 | 0 | 237 | APA450 | 1.9% | 84 |
| Axcelerator® RTAX-S | 8 | Small | 93 | 43 | 0 | 136 | AX250 RTAX250 | 3.2% | 125 |
| Fusion ProASIC3/E IGLOO/e | 16 | Small | 233 | 55 | 0 | 308 | AFS600 A3P600 AGL600 | 2.2% | 82 |
| ProASIC | 16 | Small | 263 | 61 | 0 | 324 | APA450 | 2.6% | 81 |
| Axcelerator RTAX-S | 16 | Small | 123 | 55 | 0 | 178 | AX250 RTAX250 | 4.2% | 103 |
| Fusion ProASIC3/E IGLOO/e | 32 | Small | 311 | 71 | 0 | 418 | AFS600 A3P600 AGL600 | 3.0% | 60 |
| ProASIC | 32 | Small | 375 | 80 | 0 | 455 | APA450 | 3.7% | 62 |
| Axcelerator RTAX-S | 32 | Small | 187 | 75 | 0 | 262 | AX250 RTAX250 | 6.2% | 100 |
| Fusion ProASIC3/E IGLOO/e | 8 | Medium | 356 | 71 | 1 | 427 | AFS600 A3P600 AGL600 | 3.1% | 57 |
| ProASIC | 8 | Medium | 431 | 82 | 1 | 513 | APA450 | 4.2% | 42 |
| Axcelerator RTAX-S | 8 | Medium | 224 | 72 | 1 | 296 | AX250 RTAX250 | 7.0% | 88 |
| Fusion ProASIC3/E IGLOO/e | 16 | Medium | 549 | 83 | 1 | 632 | AFS600 A3P600 AGL600 | 4.6% | 42 |
| ProASIC | 16 | Medium | 622 | 91 | 2 | 713 | APA450 | 5.8% | 32 |
| Axcelerator RTAX-S | 16 | Medium | 299 | 87 | 1 | 386 | AX250 RTAX250 | 9.1% | 74 |
| Fusion ProASIC3/E IGLOO/e | 32 | Medium | 889 | 99 | 2 | 988 | AFS600 A3P600 AGL600 | 7.2% | 38 |
| ProASIC | 32 | Medium | 940 | 108 | 4 | 1,048 | APA450 | 8.5% | 28 |
| Axcelerator RTAX-S | 32 | Medium | 435 | 104 | 2 | 539 | AX250 RTAX250 | 12.8% | 65 |

Table 1 · CoreABC Utilization Data (Hard Mode—instructions held in tiles) (continued)

| Family | Data Width | Config. | Comb. | Seq. | RAM | Total | Device | Utilization | Frequency MHz |
|---|---|---|---|---|---|---|---|---|---|
| Fusion ProASIC3/E IGLOO/e | 8 | Large | 468 | 78 | 1 | 546 | AFS600 A3P600 AGL600 | 4.0% | 42 |
| ProASIC | 8 | Large | 559 | 90 | 1 | 649 | APA450 | 5.3% | 38 |
| Axcelerator RTAX-S | 8 | Large | 285 | 82 | 1 | 367 | AX250 RTAX250 | 8.7% | 73 |
| Fusion ProASIC3/E IGLOO/e | 16 | Large | 640 | 90 | 1 | 730 | AFS600 A3P600 AGL600 | 5.3% | 27 |
| ProASIC | 16 | Large | 752 | 99 | 2 | 851 | APA450 | 6.9% | 24 |
| Axcelerator RTAX-S | 16 | Large | 394 | 94 | 1 | 488 | AX250 RTAX250 | 11.6% | 70 |
| Fusion ProASIC3/E IGLOO/e | 32 | Large | 1,003 | 106 | 2 | 1,109 | AFS600 A3P600 AGL600 | 8.0% | 34 |
| ProASIC | 32 | Large | 1,070 | 121 | 4 | 1,191 | APA450 | 9.7% | 18 |
| Axcelerator RTAX-S | 32 | Large | 578 | 114 | 2 | 692 | AX250 RTAX250 | 16.4% | 54 |

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 | 122 | 25 | 3 | 159 | AFS600 A3P600 AGL600 | 1.2% | 70 |
| ProASIC | 8 | Small | 133 | 28 | 6 | 161 | APA450 | 1.3% | 54 |
| Axcelerator RTAX-S | 8 | Small | 59 | 25 | 3 | 84 | AX250 RTAX250 | 2.0% | 97 |
| Fusion ProASIC3/E IGLOO/e | 16 | Small | 174 | 33 | 4 | 227 | AFS600 A3P600 AGL600 | 1.6% | 69 |
| ProASIC | 16 | Small | 208 | 37 | 8 | 245 | APA450 | 2.0% | 51 |
| Axcelerator RTAX-S | 16 | Small | 90 | 33 | 4 | 123 | AX250 RTAX250 | 2.9% | 85 |
| Fusion ProASIC3/E IGLOO/e | 32 | Small | 347 | 49 | 5 | 396 | AFS600 A3P600 AGL600 | 2.9% | 47 |
| ProASIC | 32 | Small | 352 | 55 | 10 | 407 | APA450 | 3.3% | 42 |
| Axcelerator RTAX-S | 32 | Small | 151 | 49 | 5 | 200 | AX250 RTAX250 | 4.7% | 65 |

Table 2 · CoreABC Utilization Data (Soft Mode—instructions held in RAM) (continued)

| Family | Data Width | Config. | Comb. | Seq. | RAM | Total | Device | Utilization | Frequency MHz |
|---|---|---|---|---|---|---|---|---|---|
| Fusion ProASIC3/E IGLOO/e | 8 | Medium | 320 | 52 | 4 | 372 | AFS600 A3P600 AGL600 | 2.7% | 48 |
| ProASIC | 8 | Medium | 401 | 55 | 7 | 456 | APA450 | 3.7% | 34 |
| Axcelerator RTAX-S | 8 | Medium | 205 | 52 | 4 | 257 | AX250 RTAX250 | 6.1% | 59 |
| Fusion ProASIC3/E IGLOO/e | 16 | Medium | 541 | 60 | 5 | 601 | AFS600 A3P600 AGL600 | 4.4% | 40 |
| ProASIC | 16 | Medium | 652 | 68 | 10 | 720 | APA450 | 5.9% | 28 |
| Axcelerator RTAX-S | 16 | Medium | 265 | 60 | 5 | 325 | AX250 RTAX250 | 7.7% | 58 |
| Fusion ProASIC3/E IGLOO/e | 32 | Medium | 845 | 76 | 8 | 921 | AFS600 A3P600 AGL600 | 6.7% | 32 |
| ProASIC | 32 | Medium | 881 | 90 | 16 | 971 | APA450 | 7.9% | 26 |
| Axcelerator RTAX-S | 32 | Medium | 392 | 76 | 8 | 468 | AX250 RTAX250 | 11.1% | 51 |
| Fusion ProASIC3/E IGLOO/e | 8 | Large | 456 | 58 | 5 | 514 | AFS600 A3P600 AGL600 | 3.7% | 41 |
| ProASIC | 8 | Large | 529 | 64 | 9 | 593 | APA450 | 4.8% | 31 |
| Axcelerator RTAX-S | 8 | Large | 279 | 58 | 5 | 337 | AX250 RTAX250 | 8.0% | 63 |
| Fusion ProASIC3/E IGLOO/e | 16 | Large | 618 | 66 | 6 | 684 | AFS600 A3P600 AGL600 | 5.0% | 25 |
| ProASIC | 16 | Large | 725 | 79 | 12 | 804 | APA450 | 6.5% | 21 |
| Axcelerator RTAX-S | 16 | Large | 373 | 66 | 6 | 439 | AX250 RTAX250 | 10.4% | 57 |
| Fusion ProASIC3/E IGLOO/e | 32 | Large | 1.048 | 82 | 8 | 1,130 | AFS600 A3P600 AGL600 | 8.2% | 35 |
| ProASIC | 32 | Large | 1,219 | 99 | 16 | 1,318 | APA450 | 10.7% | 27 |
| Axcelerator RTAX-S | 32 | Large | 574 | 82 | 8 | 656 | AX250 RTAX250 | 15.5% | 46 |

# 1 Internal Architecture

CoreABC internal architecture is shown in Figure 1-1. The core consists of six main blocks:

- Instruction Block
- Sequencer
- ALU and Flags
- Storage
- ACM (Analog Configuration MUX)
- APB Controller

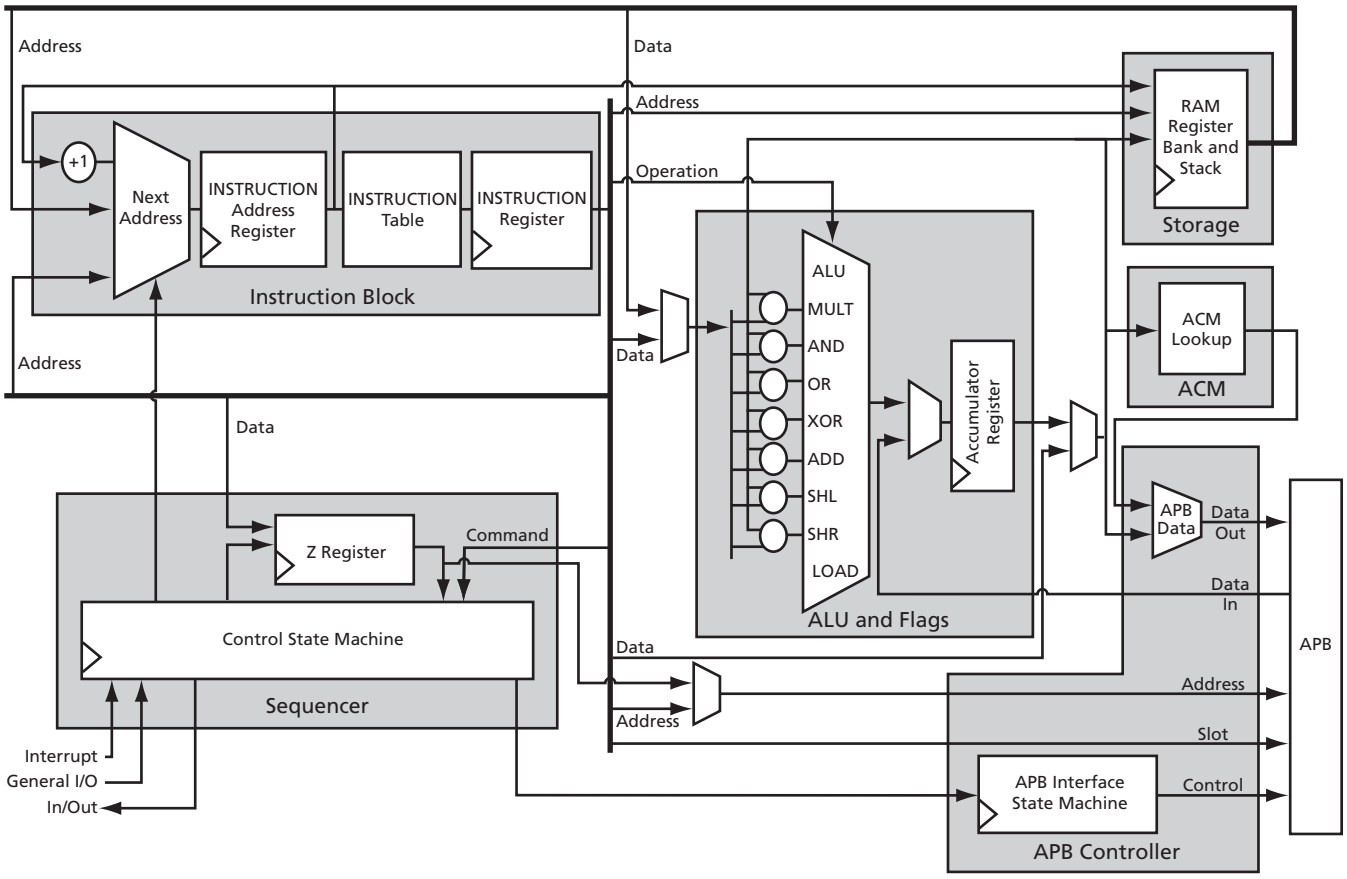


Figure 1-1 · CoreABC Block Diagram

The Instruction Block contains the instruction counter and the instruction table that contains the instructions to be executed. In the soft configuration, these instructions are fetched from RAM internal to CoreABC. See "Soft Configuration—RAM-Based Operation" on page 11.

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 FPGA to be easily configured.

The APB controller implements the APB protocol and the data input MUX, which selects data from one of the sixteen APB devices. Finally, the sequencer controls the operation of the core, decoding the instructions and enabling the other blocks.

To keep tile counts low, all unused functions within CoreABC may be removed at synthesis by setting top-level parameters.

# Advanced Peripheral Bus

CoreABC is designed to act as an APB bus master. It is recommended that the bus interface core, CoreAPB, be used with CoreABC. CoreAPB provides the read data bus multiplexer required when more that one APB slave is connected. When creating a system in CoreConsole, CoreAPB must be included if the system is to be auto-stitched. Figure 1-2 shows a complete system containing CoreABC, CoreAI, and CoreUART. This system can be used to implement a temperature/voltage monitoring system and log information via an RS-232 connection.

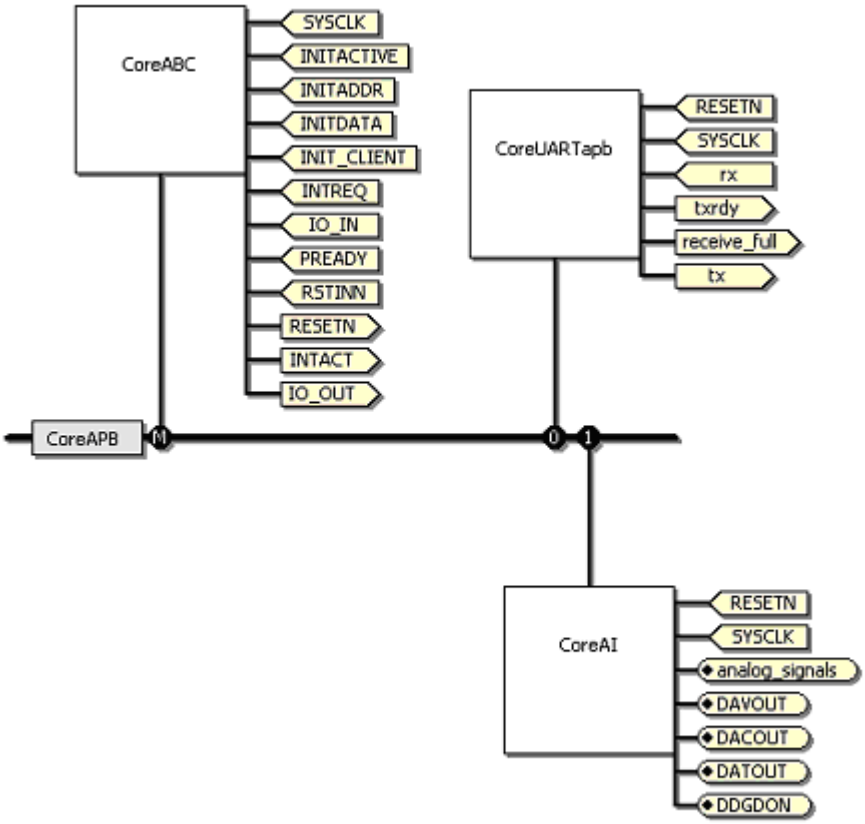


Figure 1-2 · Complete System Containing CoreABC, CoreAI, and CoreUART

# Soft Configuration—RAM-Based Operation

Soft configuration is enabled by selecting **Soft** (RAM) as the choice for Instruction Store on the CoreABC configuration screen in CoreConsole. The choice of APB bus width and other options also affects the generated core. The details of what happens within CoreABC are given in the following paragraphs.

Selecting Soft for Instruction Store sets the INSMODE parameter to 1 in CoreABC. The core implements a set of RAM blocks used to store the instructions. The number of RAM blocks is a function of the APB_AWIDTH, APB_DWIDTH, APB_SWIDTH, and ICWIDTH parameters, as well as the FPGA family being used. EQ 1 through EQ 4 show calculations involving the number of RAM blocks.

## Fusion, ProASIC3/E, Axcelerator, and RTAX-S Families

$$SWIDTH = \log_2(APB\_SDEPTH)$$

*EQ 1*

$$Nrams = (APB\_AWIDTH + APB\_DWIDTH + SWIDTH + 15) / 9 \times 2^{ICWIDTH} / 512$$

*EQ 2*

## ProASIC Family

$$SWIDTH = \log_2(APB\_SDEPTH)$$

*EQ 3*

$$Nrams = 2 \times (APB\_AWIDTH + APB\_DWIDTH + APB\_SWIDTH + 15) / 9 \times 2^{ICWIDTH} / 256$$

*EQ 4*

In soft configuration, the core should be programmed using the CoreConsole Configuration GUI, which will generate the correct memory image files. These are used during simulation to directly initialize the memory blocks, and also to initialize the RAM contents in the actual FPGA through the SmartGen memory installation system. For non-Fusion devices, an alternative scheme is required to initialize the RAM blocks though the initialization interface (using CoreMP7, for example).

The width of the INITADDR bus set by the INITWIDTH generic is given by EQ 5.

$$SWIDTH = \log_2(APB\_SDEPTH)$$

$$Depth = 512 \times (APB\_AWIDTH + APB\_DWIDTH + SWIDTH + 15) / 9 \times (1 + 2^{ICWIDTH} / 512)$$

$$INITWIDTH = rounded\_up(\log_2(Depth))$$

*EQ 5*

# 2
# Tool Flows

## Licenses

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

### Obfuscated

Complete RTL code is provided for the core, allowing the core to be instantiated with CoreConsole. Simulation, Synthesis, and Layout can be performed within Libero IDE. The RTL code for the core is obfuscated, meaning the RTL source files have had formatting and comments removed and all instance and net names have been replaced with random character sequences. Some of the testbench source files are not provided. They are pre-compiled into the compiled simulation library instead.

### RTL

Complete RTL source code is provided for the core and testbenches.

## CoreConsole

CoreABC is pre-installed in the CoreConsole IDP. To use the core, instantiate it in the design by double-clicking (or using the **Add** button) from the IP core list. The CoreConsole project can be exported to Libero IDE at this point, providing access to CoreABC. Other IP blocks can be connected, allowing a complete system to be exported from CoreConsole to Libero IDE.

After configuring the core, Actel recommends that you use the top-level Auto Stitch function to connect all the core interface signals to the top level of the CoreConsole project. Once the core is configured, invoke the **Generate** function in CoreConsole. This will export all the required files to the project directory in the *LiberoExport* directory. This is in the CoreConsole installation directory by default.

The core can be configured using the configuration GUI within CoreConsole, as shown in Figure 2-1.

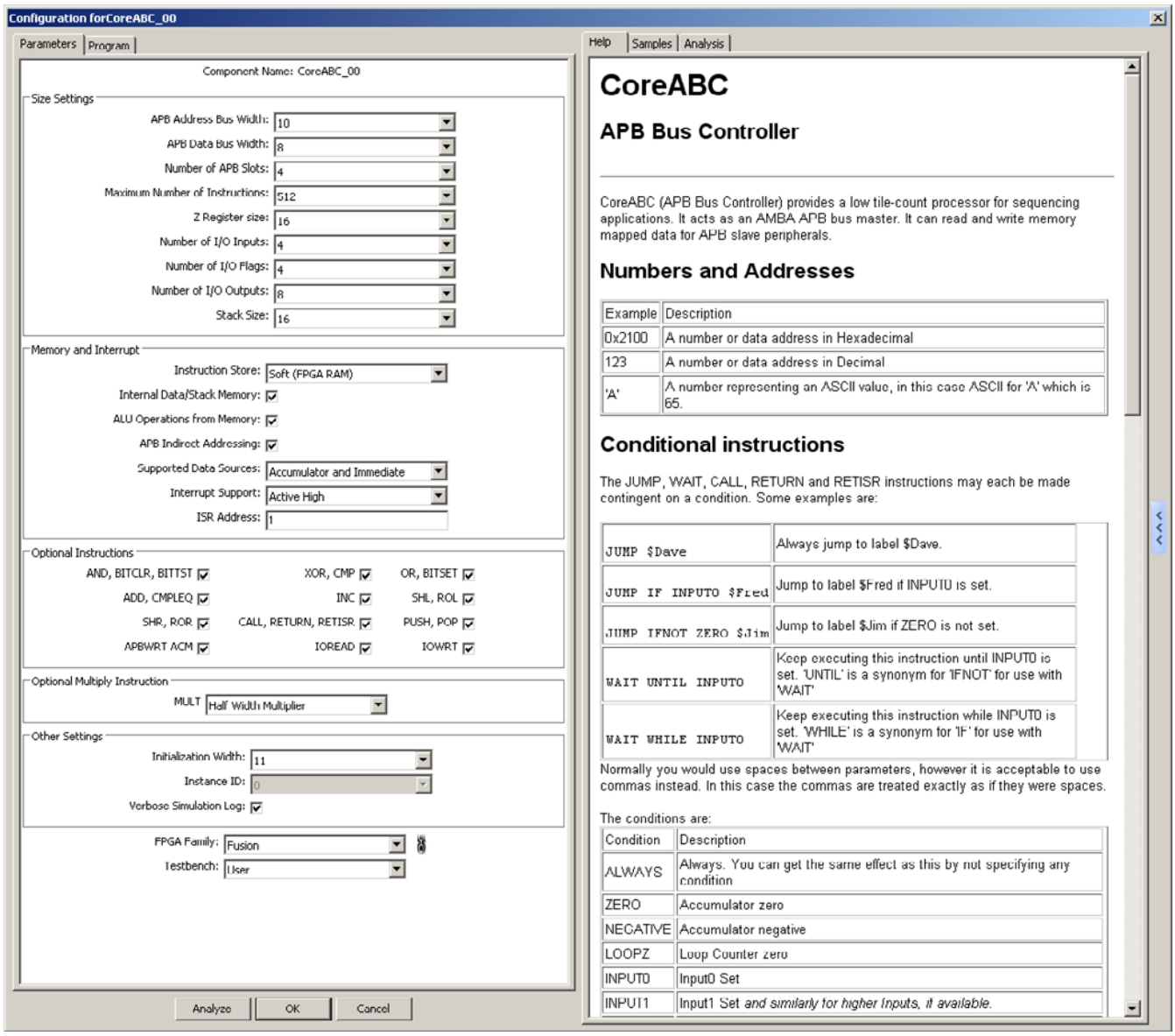


Figure 2-1 · CoreABC Configuration Screen

The Configurator has five tabs, two on the left and three on the right. Each tab is explained below.

## Parameters

This is the main configuration area where the size, selected instructions, etc. are selected. This is described in detail in "CoreABC Configuration" on page 35.

### Program

This is where the CoreABC program is created and analyzed.

### Help

This is the connection and instruction set help for CoreABC.

### Samples

These are sample programs that can be copied and pasted into the Program window and used for reference, or as a starting point for your own programs.

### Analysis

This provides advanced and in-depth analysis of your program to assist in debug.

## Importing into Libero IDE

After generating and exporting the core from CoreConsole, the core can be imported into Libero IDE. Create a new project in Libero IDE and import the CoreConsole project from the *LiberoExport* directory. Libero IDE will then install the core and the selected testbenches into its project, along with constraints and documentation.

Note: If two or more DirectCores are required, they can both be included in the same CoreConsole project and imported into Libero IDE at the same time.

## Simulation Flows

To run simulations, select the required testbench flow within CoreConsole through the core configuration GUI. Run **Save & Generate** from the Generate pane.

When CoreConsole generates the Libero IDE project, it will install the appropriate testbench files.

***To run the testbenches:***

1. **If using Libero IDE v8.0, switch the Libero IDE Design Explorer Hierarchy view to show Modules.**
2. **Set the design root to the CoreABC instantiation in the Libero IDE file manager and click the Simulation icon** in Libero IDE. This will invoke Model*Sim*® and automatically run the simulation.

## Synthesis in Libero IDE

To run Synthesis on the core with parameters set in CoreConsole, set the design root to the top of the project imported from CoreConsole. This is a wrapper around the core that sets all the generics appropriately. Click the **Synthesis** icon in Libero IDE. The Synthesis window appears, displaying the Synplicity project. To run Synthesis, click the **Run** icon.

## Place-and-Route in Libero IDE

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

# 3

# Interface Descriptions

## Interface Descriptions

CoreABC is available with the APB interface.

## Parameters

Note: The parameters described are those directly in the RTL. The recommended configuration flow is to use the configuration screens in CoreConsole, which will then instantiate these parameters correctly.

Table 3-1 · 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 to 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 12 | Sets the maximum number of supported instructions. Number of allowed instructions is $2^{ICWIDTH}$.<br>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. | | | |
| 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 |

Table 3-1 · CoreABC Parameters (continued)

| Parameter | Values | Description | Value | | |
|---|---|---|---|---|---|
| | | | Small | Medium | Large |
| 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 Generic" on page 19. | 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 4,095 | The address CoreABC should jump to when responding to an interrupt request. | 0 | 220 | 220 |
| INSMODE | 0 to 1 | 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. | 0 | 0 | 1 |
| 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 |

Table 3-1 · CoreABC Parameters (continued)

| Parameter | Values | Description | Value | | |
|---|---|---|---|---|---|
| | | | Small | Medium | Large |
| 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 |
| ID | 0 to 9 | Used to specify the CoreABC instance number when more than one CoreABC core is used within a single design. | 0 | 0 | 0 |

## EN_DATAM Generic

This allows various internal multiplexers to be optimized out of the core, lowering tile counts. The settings supported are given in Table 3-1 through Table 3-5 on page 20, and the tables show which instructions are allowed with each setting.

Table 3-2 · 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 3-3 · Immediate Only (EN_DATAM = 1)

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

Table 3-4 · 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 3-5 · Instruction-Dependent (EN_DATAM = 3)

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

# Ports

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

Table 3-6 · CoreABC Port Descriptions

| Name | Type | Description |
|---|---|---|
| PCLK | In | Master Clock input |
| NSYSRESET | In | Master Reset input (asynchronous active low) |
| PRESETN | Out | Reset output; synchronized version of NSYSRESET |
| PENABLE | Out | APB enable signal |
| PWRITE | Out | APB write signal |
| PSEL[x:0] | Out | One-hot-encoded APB select signals. One select is provided for each of the enabled slots. These outputs are intended for connection to the PSEL input on each of the connected APB devices. The width is controlled by APB_SDEPTH. |
| PADDR[x:0] | Out | APB address bus. The width is controlled by APB_AWIDTH. |
| PWDATA[x:0] | Out | APB write data bus. The width is controlled by APB_DWIDTH. |
| PRDATA[x:0] | In | APB read data bus. The width is controlled by APB_DWIDTH. |
| PREADY | In | Optional READY input. When deasserted (LOW), extends the second APB cycle until it is asserted. This input is not required if the APB devices are fully APB-compliant, in which case this input should be tied HIGH. PREADY is not part of the APB specification but is provided to simplify the design of peripherals that are connected to CoreABC. |
| 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. |
| INITDATAVAL | 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 client select signal. |
| 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 tied to logic 0. |

# 4

# 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.

## Address Spaces

The instruction address is a linear address space that is physically either a hard-coded instruction table (hard version) or an internal instruction RAM (soft version). This is implicitly accessed by control transfer instructions such as JUMP and CALL, but it cannot be directly read or written otherwise.

The data address spaces are shown in Figure 4-1. There are three separate, independent addressable areas. These are accessed by using instructions or instruction modes unique to each one.

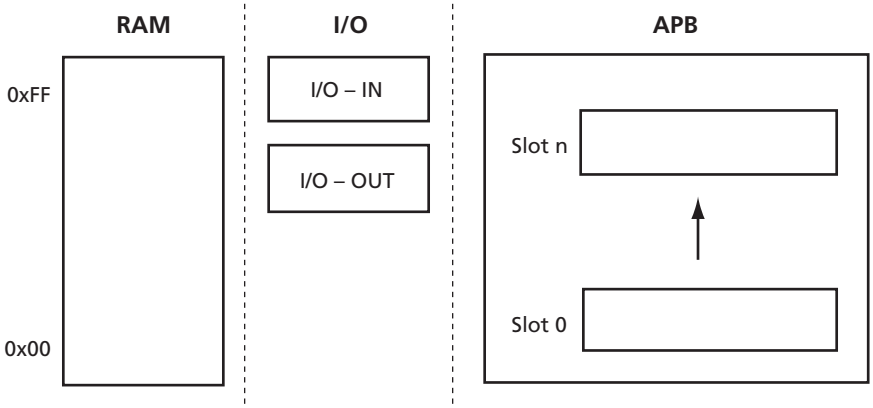


Figure 4-1 · CoreAPB Data Address Spaces

### Internal Data RAM Address Space

This is an 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.

### 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).

### APB Address Space

The APB has up to eight slots (individual connected peripherals), each of which can have an internal memory space of up to 64 k locations that is specific to each peripheral. These 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.

# Flags Register—Inputs and Condition Codes

CoreABC maintains a control register that is used in the conditional instructions; e.g., 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 4-2.

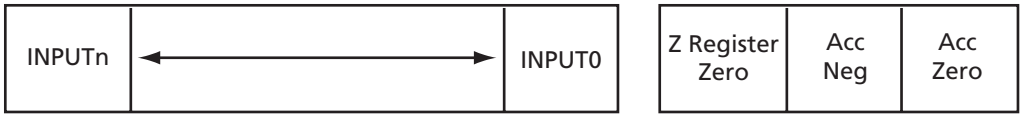


Figure 4-2 · Flags and Inputs Register

There are three condition code type flags:

| | |
|---|---|
| ZERO | Accumulator zero |
| NEGATIVE | Accumulator negative |
| ZZERO | Z Register zero |

And there are *n* INPUTS ($n \leq 28$), INPUT0 … INPUT*n*, which are directly mapped to the general-purpose inputs connected to CoreABC. 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.

# Instruction Set

Table 4-1 through Table 4-7 on page 28 list the supported instructions. More detail is given in "Adding User Instructions" on page 51.

Table 4-1 · The Boolean and Arithmetic Instruction Group

| Instruction[1, 2] | Description |
|---|---|
| LOAD DAT *Data* | Load accumulator with value. |
| LOAD RAM *Address* | Load accumulator with value. |
| AND DAT *Data* | Bitwise AND accumulator with immediate data. |
| AND RAM *Address* | Bitwise AND accumulator with RAM location. |
| OR DAT *Data* | Bitwise OR accumulator with immediate data. |
| OR RAM *Address* | Bitwise OR accumulator with RAM location. |
| XOR DAT *Data* | Bitwise XOR accumulator with immediate data. |
| XOR RAM *Address* | Bitwise XOR accumulator with RAM location. |
| ADD DAT *Data* | Add immediate data to accumulator. |
| ADD RAM *Address* | Add RAM location to accumulator. |

*Notes:*

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.*

Table 4-1 · The Boolean and Arithmetic Instruction Group (continued)

| Instruction[1, 2] | Description |
|---|---|
| SUB DAT *Data* | Subtract immediate data from accumulator.<br>Note: SUB RAM is not supported. |
| MULT DAT *Data* | Multiply accumulator by immediate data.<br>Note: Core parameters determine multiplier return value. |
| MULT RAM *Address* | Multiply accumulator by RAM location.<br>Note: Core parameters determine multiplier return value. |
| CMP DAT *Data* | Compare accumulator to immediate data.<br>ZERO set if equal; NEGATIVE set if MSBs differ. |
| CMP RAM *Address* | Compare accumulator to RAM location.<br>ZERO set if equal; NEGATIVE set if MSBs differ. |
| CMPLEQ_DAT *Data* | Compare accumulator to immediate data.<br>ZERO set if equal; NEGATIVE set if ACC < Data.<br>Note: CMPLEQ_RAM is not supported. |
| SHL0 | Shift accumulator left and infill with 0. |
| SHR0 | Shift accumulator right and infill with 0. |
| SHL1 | Shift accumulator left and infill with 1. |
| SHR1 | Shift accumulator right and infill with 1. |
| SHLE | Shift accumulator left and infill with LSB. |
| SHRE | Shift accumulator right and infill with MSB. |
| ROL | Rotate accumulator left. |
| ROR | Rotate accumulator right. |
| BITCLR *Data* | Clear one bit in accumulator specified by argument (AND). In this case, the data value specifies the bit position. |
| BITSET *Data* | Set one bit in accumulator specified by argument (OR). In this case, the data value specifies the bit position. |
| 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. |

*Notes:*

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.*

Table 4-2 · The Memory Instruction Group

| Instruction | Description |
|---|---|
| PUSH | Push the accumulator onto the stack. |
| PUSH ACC | Push the accumulator onto the stack. |
| PUSH DAT *Data* | Push immediate data onto stack. |
| POP | Pop data from the stack to the accumulator and update the flags. |
| RAMWRT *Address* ACC | Write accumulator to RAM address. |
| RAMWRT *Address* DAT *Data* | Write immediate data to RAM address. |
| RAMREAD *Address* | Read data from RAM address to the accumulator and update the flags. |

Table 4-3 · The Z Register* Instruction Group

| Instruction | Description |
|---|---|
| LOADZ ACC | Load Z with accumulator. |
| LOADZ DAT *Data* | Load Z with immediate value. |
| ADDZ ACC | Add accumulator to Z and store in Z.<br>Only ZZERO flag is affected. |
| ADDZ DAT *Data* | Add immediate data to Z and store in Z.<br>Only ZZERO flag is affected. |
| SUBZ DAT *Data* | Subtract immediate data from Z and store in Z.<br>Only ZZERO flag is affected<br>Note: SUBZ ACC is not supported. |
| INCZ | Increment Z. Only ZZERO flag is affected. |
| DECZ | Decrement Z. Only ZZERO flag is affected. |

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

Table 4-4 · The APB Instruction Group

| Instruction | Description |
|---|---|
| APBREAD *Slot Address* | Read from APB. |
| APBWRT ACC *Slot Address* | Write accumulator to APB at chosen address. |
| APBWRT ACM *Slot Address* | Write value of ACM table, at location given by accumulator, to APB at chosen address. |
| APBWRT DAT *Slot Address Data* | Write data to chosen address. |
| APBREADZ *Slot* | Read from APB. The Z register specifies the APB address. |
| APBWRTZ ACC *Slot* | Write accumulator to APB. The Z register specifies the APB address. |
| APBWRTZ ACM *Slot* | Write value of ACM table, at location given by accumulator. The Z register specifies the APB address. |
| APBWRTZ DAT *Slot Data* | Write data; the Z register specifies the APB address. |

Table 4-5 · The I/O Instruction Group

| Instruction | Description |
|---|---|
| IOWRT ACC | Write accumulator to I/O register. |
| IOWRT DAT *Data* | Write data value to I/O register. |
| IOREAD | Load the accumulator with the I/O input value. |

Table 4-6 · The Flow Control Instruction Group

| Instruction | Description |
|---|---|
| JUMP *Condition $Label* | Jump to label. |
| JUMP *IF/IFNOT Condition $Label* | Jump on condition to label. |
| WAIT UNTIL/WHILE *Condition* | Stop at this instruction until condition is TRUE. |
| CALL *$Label* | As JUMP, but puts return address on stack. |
| CALL *IF/IFNOT Condition $Label* | As JUMP, but puts return address on stack. |
| RETURN | Return from a CALL. |
| RETURN *IF/IFNOT Condition* | Return from a CALL on condition. |
| RETISR *Condition* | Return from an interrupt. |
| RETISR *IF/IFNOT Condition* | Return from an interrupt on condition. |
| HALT | Stop at this instruction. Interrupts will still be processed. HALT is a synonym for WAIT, and generally used without a condition. |

Table 4-7 · 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 4-8 · Other Instructions

| Instruction | Description |
|---|---|
| NOP | No operation |
| USER *scmd slot addr data* | User's extension instruction(s). To make use of this you will need to modify the RTL directly; the user instruction can make use of the subcommand, slot, address, and data fields. |

# 5 CoreABC Operation

## Instruction Encoding

The CoreABC instruction set is encoded as shown in Table 5-1.

These encodings are primarily of use to engineers programming CoreABC by modifying the RTL directly. It is *strongly recommended* that programming be done using the CoreABC configuration and programming interface in CoreConsole.

Table 5-1 · CoreABC Instruction Encoding

| Instruction | Description | Flags | | | Encoding | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Zero | Neg | Cmd | Scmd | Slot | Addr | Data | Cycles |
| NOP | No operation | – | – | 111 | xx1 | xxx | xxx | xxx | 3 |
| LOAD | Load accumulator. | Yes | Yes | 000 | 111 | xxx | xxx | DATA | 3 |
| INC | Increment accumulator. | Yes | Yes | 000 | 000 | xxs | xxx | xxx | 3 |
| AND | AND accumulator with data. | Yes | Yes | 000 | 001 | xxs | xxx | DATA | 3 |
| OR | OR accumulator with data. | Yes | Yes | 000 | 010 | xxs | xxx | DATA | 3 |
| XOR | XOR accumulator with data. | Yes | Yes | 000 | 011 | xxs | xxx | DATA | 3 |
| ADD | Add data to accumulator. | Yes | Yes | 000 | 100 | xxs | xxx | DATA | 3 |
| MULT | Multiply accumulator by data. | Yes | Yes | 000 | 000 | xxs | xxx | DATA | 3 |
| SUB | Subtract data from accumulator. | Yes | Yes | 000 | 100 | xxs | xxx | DATATC | 3 |
| SHL0 | Shift accumulator left; infill with 0. | Yes | Yes | 000 | 101 | xxx | xxx | xx00 | 3 |
| SHL1 | Shift accumulator left; infill with 1. | Yes | Yes | 000 | 101 | xxx | xxx | xx01 | 3 |
| SHLE | Shift accumulator left; infill with LSB. | Yes | Yes | 000 | 101 | xxx | xxx | xx10 | 3 |
| ROL | Rotate accumulator left. | Yes | Yes | 000 | 101 | xxx | xxx | xx11 | 3 |
| SHR0 | Shift accumulator right; infill with 0. | Yes | Yes | 000 | 110 | xxx | xxx | xx00 | 3 |
| SHR1 | Shift accumulator right; infill with 1. | Yes | Yes | 000 | 110 | xxx | xxx | xx01 | 3 |
| SHRE | Shift accumulator right; infill with MSB. | Yes | Yes | 000 | 110 | xxx | xxx | xx10 | 3 |
| ROR | Rotate accumulator right. | Yes | Yes | 000 | 110 | xxx | xxx | xx11 | 3 |

*Notes:*

1. *BITNS values will set BIT N in the data value with all other bits cleared; e.g., if BIT = 4, then DATA = 0x10.*
2. *BITNC values will clear BIT N in the data value with all other bits set; e.g., if BIT = 4, then DATA = 0xEF.*
3. *DATATC indicates that the data value contains the two's complement value of the required data value.*
4. *For the jump, call, and return instructions, the "s" in the Scmd field is 0 for NOTIF and 1 for IF.*
5. *For the wait instructions, the "s" in the Scmd field is 0 for UNTIL and 1 for WHILE.*
6. *The "s" in the Slot field is 0 for immediate data and 1 for either RAM or accumulator source data.*
7. *The USER instruction may be passed up to four parameters. These are expected to set the Scmd, Slot, Address, and Data instruction fields. Bit 0 of the Scmd field must be set to 1 to differentiate it from the NOP instruction.*

Table 5-1 · CoreABC Instruction Encoding (continued)

| Instruction | Description | Flags | | | Encoding | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Zero | Neg | Cmd | Scmd | Slot | Addr | Data | Cycles |
| CMP | Compare accumulator. ZERO flag set if the values are equal. NEG flag set if MSB of data is different from MSB of accumulator (XOR instruction must be enabled). | Yes | Yes | 001 | 011 | xxs | xxx | DATA | 3 |
| CMPLEQ_ | Compare accumulator. ZERO flag set if the values are equal. NEG flag set if the accumulator is less than the data value (ADD instruction must be enabled). | Yes | Yes | 001 | 100 | xxs | xxx | DATATC | 3 |
| BITCLR BIT | Clear a bit in the accumulator (AND instruction must be enabled). | Yes | Yes | 000 | 001 | xxx | xxx | BITNC | 3 |
| BITSET BIT | Seat a bit in the accumulator (OR instruction must be enabled). | Yes | Yes | 000 | 001 | xxx | xxx | BITNC | 3 |
| BITTST BIT | Test a bit in the accumulator. Sets ZERO flag if bit is zero (AND instruction must be enabled). | Yes | Yes | 001 | 001 | xxx | xxx | BITNS | 3 |
| APBREAD | Read from the APB and store value in the accumulator. Flags will not be altered by this instruction. | – | – | 010 | 011 | SLOT | ADDR | xxx | 5 |
| APBREADZ | Read from the APB and store value in the accumulator. Flags will not be altered by this instruction. | – | – | 010 | 111 | SLOT | xxx | xxx | 5 |
| APBWRT | Write to the APB. The second field specifies the data source.<br>ACC: Accumulator<br>ACM: ACM lookup<br>DAT: Immediate data<br>DAT8: Immediate data, 8 bits only<br>DAT16: Immediate data, 16 bits only | – | – | 010 | <br>000<br>010<br>001<br>001<br>001 | SLOT | ADDR | DATA | 5 |

*Notes:*

1. *BITNS values will set BIT N in the data value with all other bits cleared; e.g., if BIT = 4, then DATA = 0x10.*
2. *BITNC values will clear BIT N in the data value with all other bits set; e.g., if BIT = 4, then DATA = 0xEF.*
3. *DATATC indicates that the data value contains the two's complement value of the required data value.*
4. *For the jump, call, and return instructions, the "s" in the Scmd field is 0 for NOTIF and 1 for IF.*
5. *For the wait instructions, the "s" in the Scmd field is 0 for UNTIL and 1 for WHILE.*
6. *The "s" in the Slot field is 0 for immediate data and 1 for either RAM or accumulator source data.*
7. *The USER instruction may be passed up to four parameters. These are expected to set the Scmd, Slot, Address, and Data instruction fields. Bit 0 of the Scmd field must be set to 1 to differentiate it from the NOP instruction.*

Table 5-1 · CoreABC Instruction Encoding (continued)

| Instruction | Description | Flags | | | Encoding | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Zero | Neg | Cmd | Scmd | Slot | Addr | Data | Cycles |
| APBWRTX | Write to the APB. The second field specifies the data source.<br>ACC: Accumulator<br>ACM: ACM lookup<br>DAT: Immediate data<br>DAT8: Immediate data, 8 bits only<br>DAT16: Immediate data, 16 bits only | – | – | 010 | <br>100<br>110<br>101<br>101<br>101 | SLOT | xxx | DATA | 5 |
| LOADZ | Load the Z register. | – | – | 011 | 000 | xxs | xxx | DATA | 3 |
| DECZ | Decrement the Z register; will set the Z register zero flag when decrementing from 1 to 0. | – | – | 011 | 001 | xxx | xxx | –1 | 3 |
| INCZ | Increment the Z register; will set the Z register zero flag when incrementing to 0. | – | – | 011 | 001 | xxx | xxx | 1 | 3 |
| ADDZ | Add to the Z register; will set the Z register zero flag when result is 0. | – | – | 011 | 001 | xxx | xxx | DATA | 3 |
| SUBZ | Subtract from the Z register; will set the Z register zero flag when result is 0. | – | – | 011 | 001 | xxx | xxx | –DATA | 3 |
| IOWRT | Write data to I/O register. | – | – | 011 | 111 | xxs | xxx | DATA | 3 |
| IOREAD | Read data from I/O register. | – | – | 011 | 110 | xxs | xxx | xxx | 3 |
| RAMWRT | Write the accumulator to the storage RAM. | Yes | Yes | 011 | 011 | xxs | ADDR | xxx | 3 |
| RAMREAD | Read the storage RAM into the accumulator. Flags reflect the value read. | – | – | 011 | 010 | xxs | xxx | ADDR | 3 |
| PUSH | Push the accumulator onto the stack. | – | – | 011 | 100 | xxs | xxx | xxx | 3 |
| POP | Pop the accumulator from the stack. Flags reflect the value read. | Yes | Yes | 011 | 101 | xxs | xxx | xxx | 3 |
| JUMP | Jump | – | – | 100 | x01 | xxx | ADDR | xxx1 | 3 |
| JUMP | Jump on flag condition. | – | – | 100 | x0s | xxx | ADDR | FLAGS | 3 |
| CALL | Call | – | – | 101 | xx1 | xxx | ADDR | xxx1 | 3 |

*Notes:*

1. *BITNS values will set BIT N in the data value with all other bits cleared; e.g., if BIT = 4, then DATA = 0x10.*
2. *BITNC values will clear BIT N in the data value with all other bits set; e.g., if BIT = 4, then DATA = 0xEF.*
3. *DATATC indicates that the data value contains the two's complement value of the required data value.*
4. *For the jump, call, and return instructions, the "s" in the Scmd field is 0 for NOTIF and 1 for IF.*
5. *For the wait instructions, the "s" in the Scmd field is 0 for UNTIL and 1 for WHILE.*
6. *The "s" in the Slot field is 0 for immediate data and 1 for either RAM or accumulator source data.*
7. *The USER instruction may be passed up to four parameters. These are expected to set the Scmd, Slot, Address, and Data instruction fields. Bit 0 of the Scmd field must be set to 1 to differentiate it from the NOP instruction.*

Table 5-1 · CoreABC Instruction Encoding (continued)

| Instruction | Description | Flags | | | Encoding | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Zero | Neg | Cmd | Scmd | Slot | Addr | Data | Cycles |
| CALL | Call on flag condition. | – | – | 101 | xxs | xxx | ADDR | FLAGS | 3 |
| RETURN | Return | – | – | 110 | x01 | xxx | ADDR | xxx1 | 3 |
| RETURN | Return on flag condition. | – | – | 110 | x0s | xxx | ADDR | FLAGS | 3 |
| RETISR | Return from interrupt. | –– | – | 110 | x11 | xxx | ADDR | xxx1 | 3 |
| RETISR | Return from interrupt on flag condition. | – | – | 110 | x1s | xxx | ADDR | FLAGS | 3 |
| HALT | Halt operation. | – | – | 100 | x11 | xxx | ADDR | xxx1 | 8 |
| WAIT | Wait until flag condition. | – | – | 100 | x1s | xxx | ADDR | FLAGS | 3<br>8 |
| USER P1 P2 P3 P4 | User-defined instruction | ? | ? | 111 | P1 | P2 | P3 | P4 | 3 |

*Notes:*

1. *BITNS values will set BIT N in the data value with all other bits cleared; e.g., if BIT = 4, then DATA = 0x10.*
2. *BITNC values will clear BIT N in the data value with all other bits set; e.g., if BIT = 4, then DATA = 0xEF.*
3. *DATATC indicates that the data value contains the two's complement value of the required data value.*
4. *For the jump, call, and return instructions, the "s" in the Scmd field is 0 for NOTIF and 1 for IF.*
5. *For the wait instructions, the "s" in the Scmd field is 0 for UNTIL and 1 for WHILE.*
6. *The "s" in the Slot field is 0 for immediate data and 1 for either RAM or accumulator source data.*
7. *The USER instruction may be passed up to four parameters. These are expected to set the Scmd, Slot, Address, and Data instruction fields. Bit 0 of the Scmd field must be set to 1 to differentiate it from the NOP instruction.*

The JUMP, WAIT, CALL, and RETURN instructions check the contents of the flag condition register. This is shown in Table 5-2. The FLAGS value provided in the instruction is ANDed with the flag condition register. If the FLAGS value is 0x02, the ZERO flag is tested. When the FLAGS value is 0x01, the jump condition will always be true.

Table 5-2 · FLAGS Value

| (IIWIDTH + 4 – 1):0 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| IO_IN inputs | Z register zero | Accumulator negative (MSB set) | Accumulator zero | ALWAYS |

# ACM Lookup for Use with CoreAI

A separate RTL source file is provided to allow the ACM initialization values for CoreAI to be easily loaded by CoreABC using the APBWRT ACM instruction. This instruction uses the accumulator value to index into the ACM lookup table to generate the actual data value written. Instructions 6–12 in the "Example Instruction Sequence" on page 47 show the ACM registers in the Fusion device being initialized.

The ACM lookup table is a simple RTL file that provides a lookup for ACM data, as shown below:

```
process(ACMADDR)
variable ADDRINT  : integer range 0 to 255;
constant ADCCFG  : integer := 16#80#;
    -- PRE_SCALER_ON, POS_VOLTAGE, PRE_SCALER, RANGE_16V
```

```
begin
  ADDRINT := conv_integer(ACMADDR);
  ACMDO <= '1';
  case ADDRINT is
    when 1 to 20   => ACMDATA <= conv_std_logic_vector(ADCCFG, 8);
    when others    => ACMDATA <= ( others =>'-');  ACMDO <= '0';
  end case;
end process;
```

In the above example, only ACM locations 1 to 20 are written, and set to 16#80#.

When both CoreABC and CoreAI are used in a single CoreConsole project, CoreConsole automatically creates the correct ACM table lookup values to reflect those specified in the CoreAI configuration GUI. CoreConsole creates a custom *acmtable.vhd* or *acmtable.v* file when the project is exported to Libero IDE.

# 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 0F0 to 0FF hex. There is no underflow or overflow detection on the stack pointer, so it will simply wrap around from 0F0 to 0FF hex on push operations and 0FF to 0F0 hex 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.

# 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 output. When the ISR completes, the software should execute the RETISR 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 contents of the ZERO and NEGATIVE flags are saved on entry to the interrupt service routine and restored on the RETISR instruction. On entry to an ISR, 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.

An interrupt will occur when the HALT or WAIT instructions are being executed. After completion, the ISR will return to the HALT or WAIT instruction, unless the ISR does something to remove the reason for the WAIT or modifies the stack contents; e.g., it could POP the return address, modify it, and PUSH it back on the stack.

1. *Assumes PREADY is held HIGH.*

# User Instructions

The RTL code can be modified to add additional instructions to CoreABC (a full RTL license is required for this). These instructions could initiate APB cycles or add additional ALU functions, such as a multiply. "Adding User Instructions" on page 51 details how these instructions can be added.

# 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 <= 00   Flags: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, set the **Verbose Simulation Log** option on the CoreABC configuration screen in CoreConsole.

# 6 CoreABC Configuration

Configure CoreABC using the configuration screens in CoreConsole. Access these by instantiating CoreABC in your CoreConsole designs and then invoking the configurator as normal (see the *CoreConsole User's Guide* for more details). Because of the sophisticated nature of CoreABC, an external configurator is invoked, and you may see a slight delay in the screens opening.

When you invoke the CoreABC configurator, you are presented with a screen that has two tabs on the left side—Parameters and Program—and three on the right side, Help, Samples, and Analysis. The right side can be collapsed down entirely when not in use.

Configuration of CoreABC is done in the screen presented by selecting the **Parameters** tab on the left side. When you do this, you will see the screen shown in Figure 6-1.

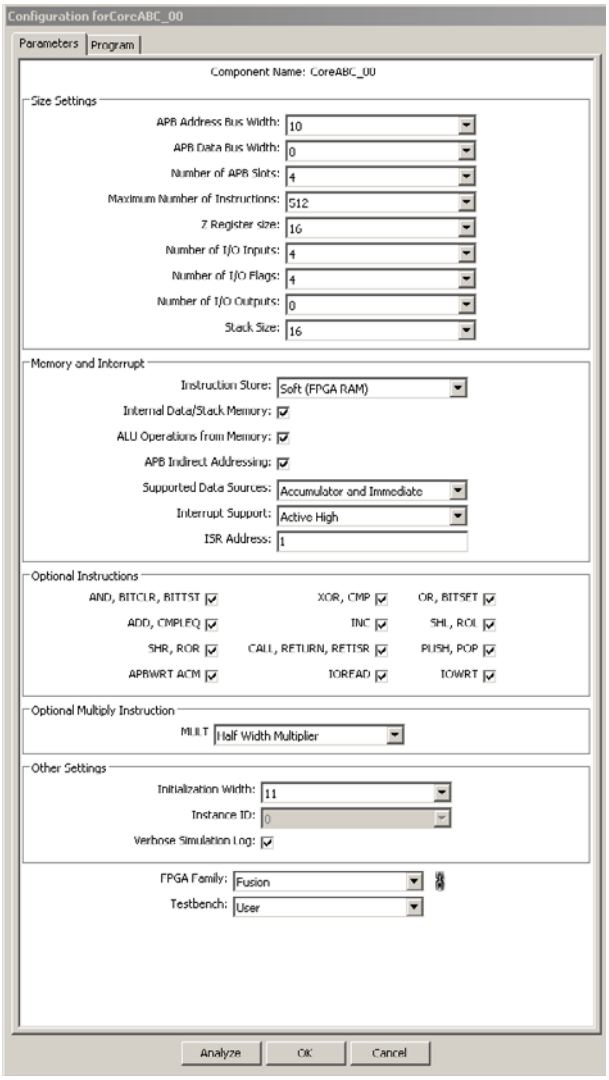


Figure 6-1 · Configuration Parameters

Each of the parameters is explained in the following sections.

## APB Address Bus Width

Selects the width of the APB address bus. This is independent of the number of APB slots. You should determine the appropriate width by reference to the number of addressable locations on the "largest" peripheral you anticipate connecting to this bus.

## APB Data Bus Width

Selects which size data bus to use on the APB mastered by CoreABC: 8, 16, or 32. Normally, you set this to the size of "widest" peripheral you anticipate connecting to this bus. However, you may occasionally decide to connect different-sized peripherals. This is stitched correctly in CoreConsole, but you should be aware of it when writing your CoreABC program.

## 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.

## Maximum Number of Instructions

This allocates the instruction space for your program (in a range from 2 to 4,096 instructions). You should not make this larger than necessary, as it is used for configuring MUXes and will directly impact the size of the core.

## 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.

## 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.

## 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.

## 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.

## 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.

## Instruction Store

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

Hard (FPGA tiles) – The program instructions are translated into RTL.

Soft (FPGA Ram) – The program instructions are generated as RAM files, and RAM is instantiated inside CoreABC, from which these instructions are executed.

## Initialization Width

When the soft version has been configured, this sets the size of the Init & Config interface for initializing the instruction RAM inside CoreABC.

## 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).

## 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.

## APB Indirect Addressing

This allows the Z register to be used as the APB address for the instructions that access the APB.

## Supported Data Sources

This controls the EN_DATAM parameter; refer to "EN_DATAM Generic" on page 19. Setting this to "Accumulator and Immediate" will increase tile counts.

## 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**.

## 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 is 1.

## 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.

## Instance ID

Set this value to allocate a unique instance ID to each CoreABC in your design. If you have only one CoreABC, you can leave this at its default value of 0.

## Testbench

Select this if you want a user testbench generated with your core.

### 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 "Simulation Logging" on page 34 for more details.

### FPGA Family

Select the target FPGA family from the drop-down list.

## 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 6-3 shows the symbols that are displayed to indicate a possible error. When you click the symbol (Figure 6-2), information is given as to the precise nature of the problem.

Figure 6-2 · Error Symbol

In the example shown in Figure 6-3, the Maximum Z Register has been set to Disabled, but there is an instruction in the program (on the other tab—LOADZ) which requires that the Z register features are available.

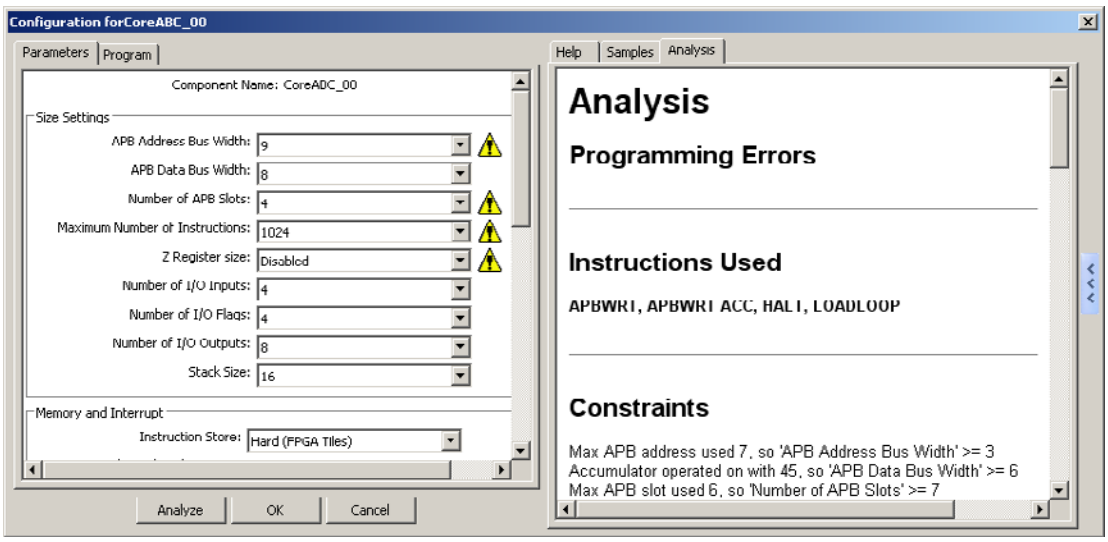


Figure 6-3 · CoreABC Configuration Validation

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 CoreABC Programming

CoreABC programs are written and assembled under the CoreABC programming tab in the CoreConsole configurator, as shown in Figure 7-1. The programs are written in the left pane, and you can view an analysis of your code in the right pane.

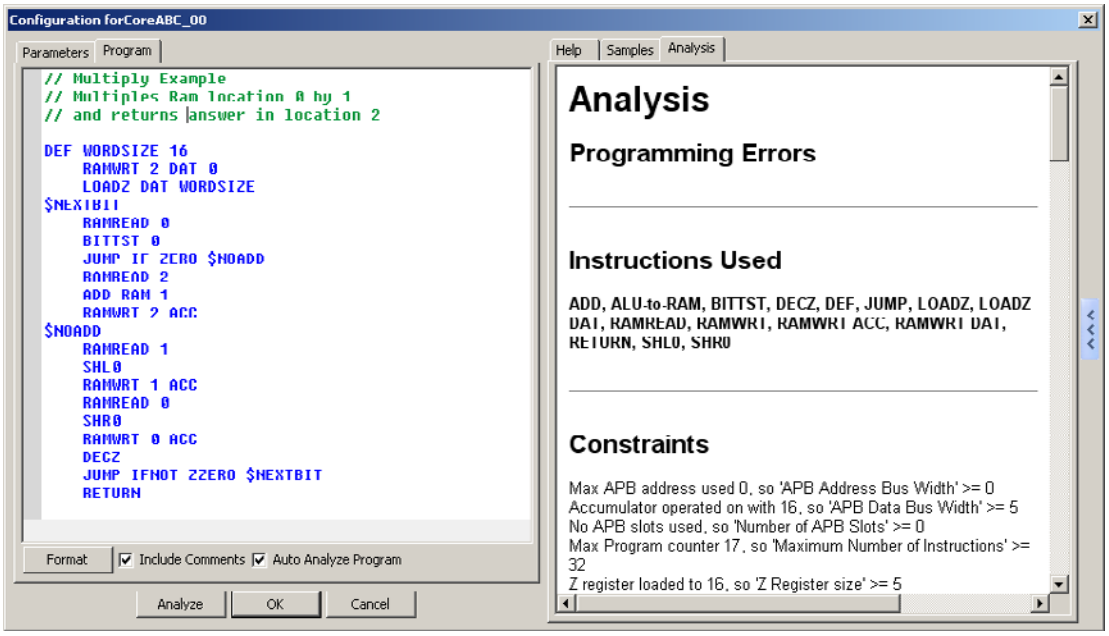


Figure 7-1 · CoreABC Programming Screen

## Analysis

The program you are writing is continuously analyzed as you write it, to detect any syntax or other errors. These are immediately flagged, and information about them is provided. At a certain point, the length of the program can slow down the analysis, which impacts usability. Therefore Auto Analyze Program will clear automatically. When Auto Analyze is off, you are required to click the **Analyze** button before you can complete and submit your program (the **OK** button will remain grayed out).

The Analysis window provides useful information and statistics on your program, most of which is self-explanatory. The key elements are covered below.

### Instructions Used

This lists the instructions used in your program. You can use this information to optimize your CoreABC by omitting any unused instructions in the Configuration screen if you want to minimize size.

## VHDL and Verilog Analysis

The CoreABC Assembler translates the program you write into RTL. The generated RTL can be seen if you scroll down the Analysis window (Figure 7-2).

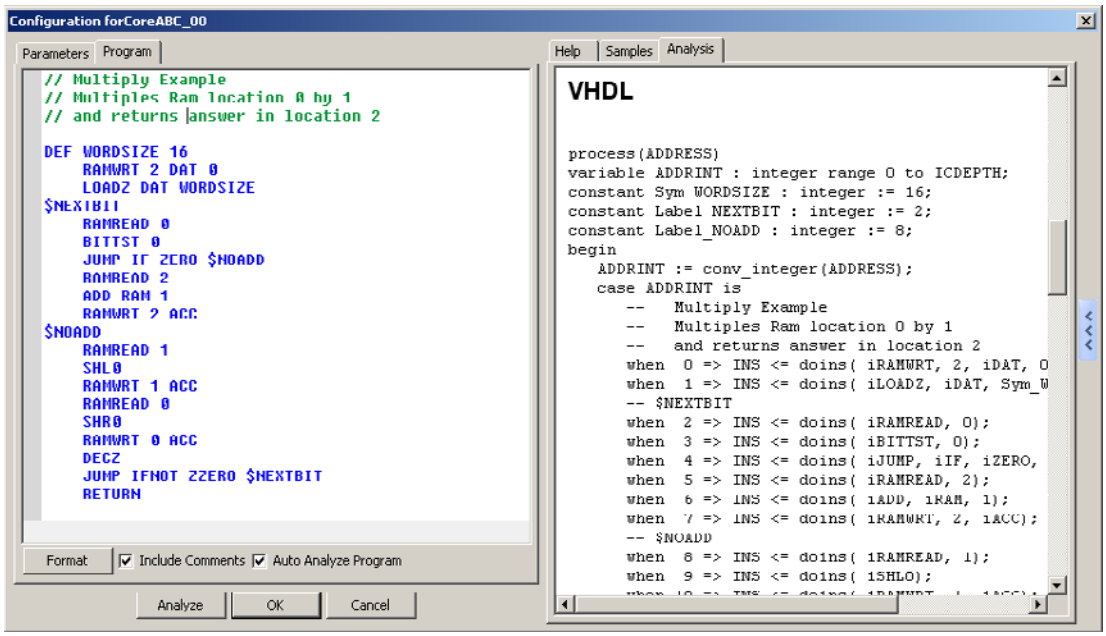


Figure 7-2 · VHDL Analysis

## Other Analysis information

Also presented in the Analysis window is the list of labels used in the program—the memory files that will be generated to support soft mode configuration and any symbols you have defined (using DEF).

# Modifying the RTL Code Directly

CoreABC can also be programmed by directly modifying the RTL code. When using the CoreConsole environment, the instructions can be implemented in logic gates or stored in RAM within the core, being initialized externally. For Fusion devices, the memory blocks can be initialized easily from the flash memory using the SmartGen memory initialization functions.

The instruction sequence is held in a simple lookup table that encodes the instruction sequence. Each instruction consists of command, subcommand, data, address, and slot fields. These five fields are collectively called the instruction.

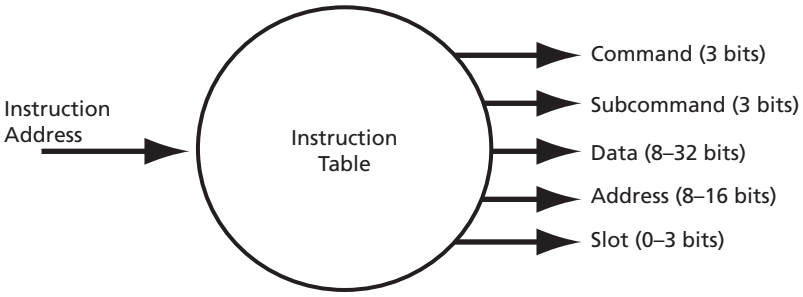


Figure 7-3 · The Instruction Encoder

The command and subcommand fields tell CoreABC what action to perform. These six bits are encoded as shown in Table 5-1 on page 29.

This structure is described by a VHDL or Verilog case statement, as shown below. To help in creating the instruction sequence, VHDL and Verilog function calls are provided to encode the instruction word correctly. These functions encode the parameters to create the instruction word. An example set of instructions follows. The parameters for the *doins* function call are as per Table 5-1 on page 29, but prefixed with an *i*.

```
process(ADDRESS)
variable ADDRINT : integer range 0 to ICDEPTH;
begin
    ADDRINT := conv_integer(ADDRESS);
    case ADDRINT is
        when 0   => INS <= doins( iJUMP,1);
        when 1   => INS <= doins( iLOAD,16#55#);
        when 2   => INS <= doins( iAND,16#0F#);
        when 3   => INS <= doins( iCMP,16#05#);
        when 4   => INS <= doins( iJUMP,iNOTIF,ZERO,30);
        …….
        when others => INS <= doins( iNOP); end case;
end process;

always @(ADDRESS)
begin
    case (ADDRINT)
        0  : INS <= doins2(iJUMP, 1) ;
        1  : INS <= doins2(iLOAD, 8'h55)
        2  : INS <= doins2(iAND, 8'h0F) ;
        : INS <= doins2(iCMP, 8'h05) ;
        4  : INS <= doins4(iJUMP, iNOTIF, ZERO, 30) ;
        ……..
        default  : INS <= doins1(iNOP) ;
    endcase
end
```

During implementation, this will be synthesized to logic tiles. To ensure minimal logic tile requirements, the above procedural calls make sure all unused instruction bits are set to “don’t care.”

# Soft Operation—Creating the Programming File

When using CoreABC in soft mode, the memory image must be created and loaded into the core. The program can be stored in the flash memory of a Fusion FPGA and automatically transferred to CoreABC at power-up. Alternatively, the memory image can be loaded by an external processor.

The automatic transfer from flash memory to CoreABC uses a SmartGen flash memory system initialization client. To use this flow, first create the CoreABC project in CoreConsole and connect the initialization ports to the top level of the design. Second, create the SmartGen core and connect it to CoreABC. Then complete the FPGA design. At this point, memory image files can be programmed into the flash memory.

## Creating a Memory Image File

There are two ways to create a memory image file: auto-generate one with the assembler tool or create one from the *instructions.vhd* or *instructions.v* RTL source file by the simulator. Both methods will tell you the size of the initialization memory and the value of the INITWIDTH parameter that sets the width of the initialization address port.

### Automatically Created Memory Image Files

The CoreConsole configuration GUI automatically creates the memory image files as per Table 7-1. These files are exported in the CoreConsole project to Libero IDE and automatically copied into the *Simulation* directory when Model*Sim* is invoked in Libero IDE.

CoreConsole will also place the *RAMABC_i.mem* file in its software export directory. This file can be directly imported by the SmartGen initialization client without reimporting the complete CoreConsole project.

#### *To manually create the memory image files from the RTL code:*

1. Make sure the required set of instructions is fully coded in the *instruction.vhd*/*instruction.v* file, and that the TESTMODE value was set to 0 when the core was configured in CoreConsole. If not, update the value by editing the *coreparameters.vhd* or *coreparamters.v* file found in the CoreConsole project in the Libero IDE file manager.
2. In Libero IDE, set the design root to the CoreABC instance within your design.
3. Invoke **Simulation**. After simulation completes, type **do makehex.do** at the Model*Sim* prompt. This will run a VHDL or Verilog simulation that will create the memory image files and report the size and width of the initialization port required.

Table 7-1 shows the memory image files created by the automatic system and the manual system.

Table 7-1 · Memory Image Files

| Memory Image | Description |
|---|---|
| RAMABC_*i*.mem | File for SmartGen or a processor to load through the initialization interface |
| RAMABC_*irc*.mem | Files that will be automatically loaded by the simulator when RTL simulation is performed. When these are located in the Libero IDE simulation directory, the instruction memories are automatically loaded at time 0 (not used for ProASIC device simulation). |
| RAMABC_*irc*L.mem<br>RAMABC_*irc*U.mem | These are generated only for ProASIC device designs and are used to initialize simulation memories as above. |

*Note: The* irc *values indicate the ID, row, and column numbers for the RAM. The ID value should be the same as the ID parameter. These values are automatically calculated and should not be changed. All three values are integers between 0 and 9.*

All memory image files are encoded using Actel binary format, containing ASCII 0 and 1 characters for each data bit in the memories.

## Creating and Using the SmartGen Initialization Client

### *To create the initialization client and use it within your design:*

Before creating the initialization client, you must create a memory image file and know the size of the initialization memory. You must create your CoreConsole system and import it into Libero IDE.

1. Create a new HDL file in Libero IDE and instantiate the entity/module from the wrapper file created by CoreConsole; this will become the new top level. You can also use this wrapper to connect other inputs and outputs of your system to glue logic, etc.
2. Choose the **SmartGen** tool within Libero IDE and create a "Flash Memory Builder Core"; select an **Initialization Client**.
3. Set the following parameters in the configuration GUI:
   - Set the **Client name** to **CoreABC** (or **CoreABCID** if using multiple cores in a single design).
   - Set the **Start address** to **0** and **Word size** to **9**.
   - Set the **Number of words** to the size of the initialization memory reported when the memory image files were created.
   - Browse and locate the memory image file. This should be found in the CoreConsole Software Export directory. Set the file type to **Actel-Binary Files**.
   - Do not enable on-demand save to flash memory.
   - Set the **Client Select Port Name** to **INITDATAVAL**.
   - Click **OK**.
4. Back in the main Flash Memory System GUI, click **Generate** and set the core name to **INITBLK**. This will generate the required RTL files.
5. Using the Libero IDE HDL editor, add INITBLK in your top-level design and connect the INITxxxx ports on INITBLK to the INITxxxx ports on CoreABC. Once this is done, you should see INITBLK displayed as part of your design hierarchy in Libero IDE.

## Programming the Flash Memory

### *To program the memory image file to the correct flash memory and locations:*

1. Under the Libero IDE File Manager tab, double-click **INITBLK SmartGen Core**. The SmartGen flash memory system will pop up.
2. Double-click the initialization client and verify that the correct memory image file is listed. Click **OK** and **Generate** in the first window. Click as appropriate on other windows that pop up.
3. Invoke **Designer** from Libero IDE. If the design has not already been through place-and-route, run it at this time.
4. Click the **Programming File** button in Designer. In the FlashPoint pop-up window, locate a small pane below the FlashROM button with Program and Instance Name fields. Click the **Program** button. Double-click the **Configuration File** field, and then locate and select the embedded memory configuration file located in the *SmartGen/INITBLK* directory in your CoreConsole project.
5. Click the **Finish** button. This will generate the STAPL file for programming. You can also program the main FPGA array at the same time and change the STAPL file name if required.
6. Using FlashPro, program the STAPL file into the FPGA.

## Updating the Program and Flash Memory Contents

### *To update the complete design and program contained in the Flash memory:*

1. In the CoreConsole project, open the CoreABC configurator and update your program. When complete, click **OK**. Click **Generate** in the main CoreConsole GUI.

2. In Libero IDE, reimport the complete Libero IDE project.
3. Under the Libero IDE File Manager tab, double-click the **INITBLK** SmartGen core. The SmartGen flash memory system will pop up. Check that the memory image file, *RAMABC_i.mem*, in the CoreConsole software export directory, is set in the GUI. Click **Generate** in the SmartGen GUI.
4. Re-run **Synthesis** and **Layout**.
5. Create the programming file in Designer, making sure that the *INITBLK.efc* file is correctly set.

### *To update only the program contained in the flash memory:*

1. In the CoreConsole project, open the CoreABC configurator and update your program. When complete, click **OK**. Click **Generate** in the main CoreConsole GUI.
2. Under the Libero IDE File Manager tab, double-click the **INITBLK** SmartGen core. The SmartGen flash memory system will pop up. Check that the memory image file, *RAMABC_i.mem,* in the CoreConsole software export directory, is set in the GUI. Click **Generate** in the SmartGen GUI. This will invalidate Synthesis in the Libero IDE flow manager.
3. Do not re-run Synthesis.
4. Re-create the programming file in Designer, making sure that the *INITBLK.efc* file is correctly set.

# 8 Testbench Operation

## Basic Testbench

Included with the releases of CoreABC is a testbench that verifies operation of the CoreABC macro. A block diagram of the testbench is shown in Figure 8-1. Identical testbenches are supplied for both the VHDL and Verilog versions of the core.

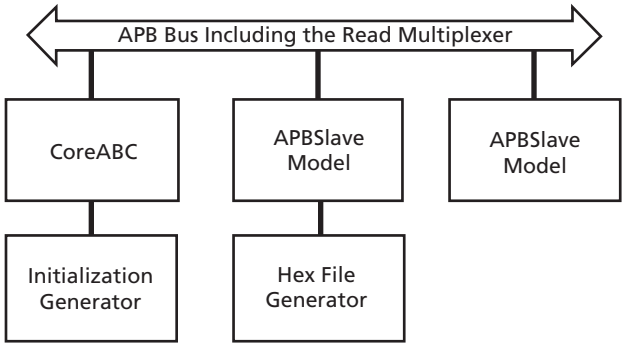


Figure 8-1 · CoreAI Verification Testbench

The testbench contains the blocks detailed in the following sections, in addition to CoreABC.

### APB Slave Models

These implement APB slaves containing memory, allowing basic APB read and write cycles to be verified.

### Initialization Generator

This block, when enabled, will control the initialization interface of CoreABC and will program the instruction RAM blocks within the core from the hex memory image files.

### Hex File Generator

This block, when enabled, will automatically create the hex memory image files from a hard version of the core.

### Simulation Options

The testbench automatically configures itself from the parameters set in the CoreConsole GUI, and uses the instruction sequences from the programming software.

When a hard core is used, the simulation simulates the instruction file that contains the program. When a soft core is being simulated, the simulation will automatically initialize the RAM contents with the required program. This is performed by the simulator and not through the initialization interface.

The operating mode of the simulation can be modified by typing the following at the Model*Sim* prompt after the initial compilation stage:

```
do runall.do +switch
```

The switch values are as follows:

+hard Core runs in hard mode (INSMODE = 0).

+gen Core runs in hard mode (INSMODE = 0); also, generate the hex image files required for soft operation.

+soft Core runs in soft mode (INSMODE = 1) with the instruction memories initialized by the simulator.

+init Core runs in soft mode (INSMODE = 1) with the instruction memories initialized through the initialization interface.

# Verification Tests

The basic testbench can be used to run the verification testbench for the core. To run the verification testbenches after Model*Sim* has been invoked and the core compiled, type the following at the Model*Sim* prompt:

```
do runall.do +all
```

This will, through top-level generics on the core, run a pre-programmed set of instruction sequences that verify the complete instruction set of the core and the external interfaces. This script will invoke the simulation multiple times. These tests are as follows:

1. 8-bit operation with a limited instruction set
2. 16-bit operation with a limited instruction set
3. 32-bit operation with a limited instruction set
4. 8-bit operation with a medium instruction set
5. 16-bit operation with a medium instruction set
6. 32-bit operation with a medium instruction set
7. 8-bit operation with a complete instruction set
8. 16-bit operation with a complete instruction set
9. 32-bit operation with a complete instruction set
10. 8-bit operation with a partial instruction set
11. 8-bit operation with a partial instruction set
12. 8-bit operation with a partial instruction set; generate hex files
13. 8-bit operation with a partial instruction set; soft mode
14. 8-bit operation with a partial instruction set; soft mode using the initialization interface
15. 8-bit operation with single APB slot
16. 8-bit operation with four APB slots

20–31. Various configurations with corner-case parameter settings; these tests do not execute any instructions.

Each test will display pass or fail status.

# A

# 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.

# B

# Adding User Instructions

When the RTL version of the core is being user-defined, additional instructions can be added to the core. Internally, the core encodes instructions using five fields:

| | |
|---|---|
| INSTR_CMD[2:0] | Command |
| INSTR_SCMD[2:0] | Subcommand |
| INSTR_SLOT[APB_SWIDTH–1:0] | APB slot |
| INSTR_ADDR[APB_AWIDTH–1:0] | APB address |
| INSTR_DATA[APB_DWIDTH–1:0] | APB data |

INSTR_CMD = 7, INSTR_SCMD = X in the base core is used as a NOP instruction. By a small change to the RTL source code, the NOP instruction is changed to use INSTR_CMD = 7, INSTR_SCMD = 0 encoding. This leaves INSTR_SCMD values 1–7 available for additional instructions, allowing seven additional instructions to be added.

### *To add user instructions:*

1. Change the EN_USER constant/parameter in *support.vhd* or *support.v* to 1. This remaps the NOP instruction to the following:

   ```
   INSTR_CMD = 111 INSTR_SCMD = xx0
   ```

2. In your program file, enter the following instruction:

   ```
   USER scmd, data, address, slot
   ```

3. The *scmd*, *data*, *address*, and *slot* values are optional values that will be encoded and passed to the core.
4. Add RTL code to the core source code (*CoreABC.vhd* or *CoreABC.v*) to implement your instructions. A comment indicates where the main code should be inserted. When the instruction is executed, INSTR_CMD will be 7. The INSTR_SCMD, INSTR_DATA, INSTR_ADDR, and INSTR_SLOT values will be set to the values used in the user-created program.

# C

# Instruction Summary

This section details all the CoreABC instructions. The encoding can be found in Table C-1 on page 67.

### NOP

Operation

No operation

Flags

Unchanged

Clock Cycles

3

### 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

### 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

### INC

Operation

Increment the accumulator.

Flags

ZERO: Set if resultant value is zero.

NEGATIVE: Set if resultant value is negative.

Clock Cycles

3

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### SUB RAM *Address*

Operation

Subtract the RAM location from the accumulator.

Flags

ZERO: Set if resultant value is zero.

NEGATIVE: Set if resultant value is negative.

Clock Cycles

3

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### CMPLEQ RAM *Address*

Operation

Compare the accumulator with the RAM location. Uses subtract operation.

Flags

ZERO: Set if values are equal.

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

Clock Cycles

3

### 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

### 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

### BITTST *N*

Operation

Tests accumulator bit *N*. Uses Boolean AND.

Flags

ZERO: Set if the bit is zero.

NEGATIVE: Undefined

Clock Cycles

3

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### 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

### LOADZ DAT *Data*

Operation

Loads the Z register with immediate data value.

Flags

ZZERO: Set if value is zero.

Clock Cycles

3

### LOADZ RAM *Address*

Operation

Loads the Z Register with RAM location.

Flags

ZZERO: Set if value is zero.

Clock Cycles

3

### DECZ

Operation

Decrements the Z register.

Flags

ZZERO: Set if the Z register decrements to zero.

Clock Cycles

3

### INCZ

Operation

Increments the Z register.

Flags

ZZERO: Set if the Z register Increments to zero.

Clock Cycles

3

### ADDZ Data

Operation

Adds Data to the Z register.

Flags

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

Clock Cycles

3

### IOREAD

Operation

Load the IO_IN port value into the accumulator.

Flags

Updated

Clock Cycles

3

### 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

### IOWRT *ACC*

Operation

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

Flags

Unchanged

Clock Cycles

3

### 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

### RAMWRT ACC *Address*

Operation

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

Flags

Unchanged

Clock Cycles

3

### RAMWRT DAT *Address Data*

Operation

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

Flags

Unchanged

Clock Cycles

3

### 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

### 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

### 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

### JUMP *Address*

Operation

Jumps always to specified instruction address.

Flags

Unchanged

Clock Cycles

3

### JUMP IF|IFNOT *Condition Address*

Operation

Jumps on or not on condition to specified instruction address. Conditions are specified in Table C-1 on page 67.

Flags

Unchanged

Clock Cycles

3

### 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

### 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 C-1 on page 67.

Flags

Unchanged

Clock Cycles

3

### RETURN

Operation

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

Flags

Unchanged

Clock Cycles

3

### 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 C-1 on page 67.

Flags

Unchanged

Clock Cycles

3

### 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

### 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

### 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 ∞

### HALT

Operation

Halt

Flags

Unchanged

Clock Cycles

∞

# Condition Codes

The conditions codes are shown in Table C-1.

Table C-1 · Condition Codes

| Condition | Description |
|---|---|
| ALWAYS | Always |
| 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) |

# D

# List of Document Changes

The following table lists critical changes that were made in the current version of the document.

| Previous Version | Changes in Current Version (v3.0) | Page |
|---|---|---|
| v2.1 | Supported core version updated in "Core Version" section. | 5 |
| | Supported version of Libero IDE updated in "Supported Tool Flows" section. | 5 |
| | The LOADLOOP register was renamed Z Register. The LOADZ condition flag was renamed ZZERO. | N/A |
| | Table 1 replaced and Table 2 created. | 6 |
| | "Utilization and Performance" section updated. | 6 |
| | Figure 1-1 updated. | 9 |
| | EQ_2 and EQ_4 updated. | 11 |
| | Figure 2-1 updated. | 14 |
| | The "Simulation Flows" section was updated. | 15 |
| | Table 3-1 updated, with numerous parameter changes, additions, and deletions. | 17 |
| | "EN_DATAM Generic" section added. | 19 |
| | "Internal Data RAM Address Space" and "I/O Address Space" sections updated. | 23 |
| | "Instruction Set" section replaced. | 24 |
| | Table 5-1 updated variously. | 29 |
| | "Simulation Logging" section updated. | 34 |
| | Figure 6-1 and Figure 6-3 were updated. | 35, 38 |
| | "Number of I/O Inputs" section added and "Number of I/O Flag Inputs" section modified. | 36 |
| | "ALU Operation from Memory", "APB Indirect Addressing", and "Supported Data Sources" sections added. | 37 |
| | Figure 7-1 and Figure 7-2 were updated. | 39, 40 |
| | "Verification Tests" section updated. | 46 |
| | "Example Instruction Sequence" appendix modified. | 47 |
| | Many instructions added or changed in "Instruction Summary" appendix. | 53 |
| v2.0 | The "Core Version" and "Supported Interfaces" sections are new. | 5 |
| | Values in the Configuration column were updated in Table 1 · CoreABC Utilization Data (Hard Mode—instructions held in tiles). | 6 |
| | The last paragraph was changed in the "ACM Lookup for Use with CoreAI" section. | 32 |
| | The "Automatically Created Memory Image Files" section is new. | 42 |
| | The "Updating the Program and Flash Memory Contents" section is new. | 43 |
| | The "Instruction Summary" section (Appendix D) is new. | 69 |

# E

# Product Support

Actel backs its products with various support services including Customer Service, a Customer Technical Support Center, a web site, an FTP site, electronic mail, and worldwide sales offices. This appendix contains information about contacting Actel and using these support services.

## Customer Service

Contact Customer Service for non-technical product support, such as product pricing, product upgrades, update information, order status, and authorization.

From Northeast and North Central U.S.A., call **650.318.4480**
From Southeast and Southwest U.S.A., call **650. 318.4480**
From South Central U.S.A., call **650.318.4434**
From Northwest U.S.A., call **650.318.4434**
From Canada, call **650.318.4480**
From Europe, call **650.318.4252** or **+44 (0) 1276 401 500**
From Japan, call **650.318.4743**
From the rest of the world, call **650.318.4743**
Fax, from anywhere in the world **650.318.8044**

## Actel Customer Technical Support Center

Actel staffs its Customer Technical Support Center with highly skilled engineers who can help answer your hardware, software, and design questions. The Customer Technical Support Center spends a great deal of time creating application notes and answers to FAQs. So, before you contact us, please visit our online resources. It is very likely we have already answered your questions.

## Actel Technical Support

Visit the Actel Customer Support website (www.actel.com/custsup/search.html) for more information and support. Many answers available on the searchable web resource include diagrams, illustrations, and links to other resources on the Actel web site.

## Website

You can browse a variety of technical and non-technical information on Actel's home page, at www.actel.com.

## Contacting the Customer Technical Support Center

Highly skilled engineers staff the Technical Support Center from 7:00 A.M. to 6:00 P.M., Pacific Time, Monday through Friday. Several ways of contacting the Center follow:

### Email

You can communicate your technical questions to our email address and receive answers back by email, fax, or phone. Also, if you have design problems, you can email your design files to receive assistance. We constantly monitor the email account throughout the day. When sending your request to us, please be sure to include your full name, company name, and your contact information for efficient processing of your request.

The technical support email address is tech@actel.com.

## Phone

Our Technical Support Center answers all calls. The center retrieves information, such as your name, company name, phone number and your question, and then issues a case number. The Center then forwards the information to a queue where the first available application engineer receives the data and returns your call. The phone hours are from 7:00 A.M. to 6:00 P.M., Pacific Time, Monday through Friday. The Technical Support numbers are:

**650.318.4460**
**800.262.1060**

Customers needing assistance outside the US time zones can either contact technical support via email (tech@actel.com) or contact a local sales office. Sales office listings can be found at www.actel.com/contact/offices/index.html.

# Index


## A

ACM lookup 32
Actel
  electronic mail 71
  telephone 72
  web-based technical support 71
  website 71
address spaces 23
ALU 9
APB
  bus master 10
  interface 5

## B

block diagram 9

## C

complete system 10
configuration 35
  parameters 35
contacting Actel
  customer service 71
  electronic mail 71
  telephone 72
  web-based technical support 71
CoreABC
  block diagram 9
  complete system 10
  configuration screen 14
  inputs 21
  overview 5
  programmer's model 23
  typical system 5
CoreAI 5, 10
CoreConsole IP Deployment Platform (IDP) 13
CorePWM 5
CoreUART 10
cross validation of configuration fields 38
customer service 71

## E

example instruction sequence 47

## F

flags 9

## I

importing into Libero IDE 15
instruction encoder 41
instruction encoding 29
instruction set 24
interface descriptions 17
internal architecture 9
interrupt operation 33

## L

Libero IDE
  importing into 15
  place-and-route in 15
  synthesis in 15

## O

Obfuscated 13

## P

ports 21
product support 71–72
  customer service 71
  electronic mail 71
  technical support 71
  telephone 72
  website 71
programmer's model 23
programming 39

## R

RTL 13

## S

simulation logging 34
SmartGen Initialization Client 43
soft configuration 11
stack 33

## T

technical support 71
testbench operation 45
tool flows 13

## U

user instructions 34, 51

utilization data 6, 7

V
verification tests 46
Verilog analysis 40
VHDL analysis 40

W
web-based technical support 71

***For more information about Actel's products, visit our website at http://www.actel.com***

***Actel Corporation*** *• 2061 Stierlin Court • Mountain View, CA 94043 USA*
*Customer Service: 650.318.1010 • Customer Applications Center: 800.262.1060*

***Actel Europe Ltd****. • River Court, Meadows Business Park • Station Approach, Blackwater • Camberley, Surrey GU17 9AB • United Kingdom*
*Phone +44 (0) 1276 609 300 • Fax +44 (0) 1276 607 540*

***Actel Japan*** *• EXOS Ebisu Bldg. 4F • 1-24-14 Ebisu Shibuya-ku • Tokyo 150 • Japan*
*Phone +81.03.3445.7671 • Fax +81.03.3445.7668 • www.jp.actel.com*

***Actel Hong Kong*** *• Suite 2114, Two Pacific Place • 88 Queensway, Admiralty Hong Kong*
*Phone +852 2185 6460 • Fax +852 2185 6488 • www.actel.com.cn*

50200085-2 /4.07