Xilinx CPLD Applications Handbook

Featuring CoolRunner-II and XC9500XL CPLDs
Xilinx is disclosing this Document and Intellectual Property (hereinafter “the Design”) to you for use in the development of designs to operate on, or interface with Xilinx FPGAs. Except as stated herein, none of the Design may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Any unauthorized use of the Design may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.

Xilinx does not assume any liability arising out of the application or use of the Design; nor does Xilinx convey any license under its patents, copyrights, or any rights of others. You are responsible for obtaining any rights you may require for your use or implementation of the Design. Xilinx reserves the right to make changes, at any time, to the Design as deemed desirable in the sole discretion of Xilinx. Xilinx assumes no obligation to correct any errors contained herein or to advise you of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or technical support or assistance provided to you in connection with the Design.

THE DESIGN IS PROVIDED “AS IS” WITH ALL FAULTS, AND THE ENTIRE RISK AS TO ITS FUNCTION AND IMPLEMENTATION IS WITH YOU. YOU ACKNOWLEDGE AND AGREE THAT YOU HAVE NOT RELIED ON ANY ORAL OR WRITTEN INFORMATION OR ADVICE, WHETHER GIVEN BY XILINX, OR ITS AGENTS OR EMPLOYEES. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DESIGN, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT OF THIRD-PARTY RIGHTS.

IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOST DATA AND LOST PROFITS, ARISING FROM OR RELATING TO YOUR USE OF THE DESIGN, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THE TOTAL CUMULATIVE LIABILITY OF XILINX IN CONNECTION WITH YOUR USE OF THE DESIGN, WHETHER IN CONTRACT OR TORT OR OTHERWISE, WILL IN NO EVENT EXCEED THE AMOUNT OF FEES PAID BY YOU TO XILINX HEREBUNDER FOR USE OF THE DESIGN. YOU ACKNOWLEDGE THAT THE FEES, IF ANY, REFLECT THE ALLOCATION OF RISK SET FORTH IN THIS AGREEMENT AND THAT XILINX WOULD NOT MAKE AVAILABLE THE DESIGN TO YOU WITHOUT THESE LIMITATIONS OF LIABILITY.

The Design is not designed or intended for use in the development of on-line control equipment in hazardous environments requiring fail-safe controls, such as in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support, or weapons systems (“High-Risk Applications”). Xilinx specifically disclaims any express or implied warranties of fitness for such High-Risk Applications. You represent that use of the Design in such High-Risk Applications is fully at your risk.

© 2005 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.
Table of Contents

Preface: About This Handbook
  Guide Contents ............................................................... x
  Additional Resources .................................................... xi

Chapter 1: Introduction to Digital Applications

IrDA and UART Design in a CoolRunner CPLD
  Introduction ................................................................. 3
  IrDA System .............................................................. 3
  UART and IrDA Design ................................................... 4
  UART Interface ........................................................... 5
  IrDA Interface ............................................................ 8
  CoolRunner Implementation! ............................................. 8
  Conclusion ................................................................. 12

Serial ADC Interface Using a CoolRunner CPLD
  Overview ................................................................. 13
  Usage ................................................................. 14
  TI ADS7870 ............................................................ 14
  CPLD Design ............................................................ 17
  Allowing the Visor to Read Conversion Results ..................... 24
  Hardware Implementation ............................................. 28
  Conclusion ................................................................. 38

Wireless Transceiver for the CoolRunner CPLD
  Introduction ................................................................. 41
  CoolRunner CPLD Transceiver Operation ............................. 41
  CoolRunner CPLD Transceiver Block Diagram ....................... 42
  CPLD Transmit Design ................................................ 45
  Receive Module Edge Detection ....................................... 46
  Hardware Description ................................................ 47
  Keyboard Entry Option ................................................ 48
  CoolRunner XPLA3 CPLD Implementation ............................. 49
  Conclusion ................................................................. 50

CoolRunner-II Smart Card Reader
  Introduction ................................................................. 53
  Smart Card Reader Block Diagram ................................... 54
# Digital Applications Handbook

## Interfacing to Mobile SDRAM with CoolRunner-II CPLDs
- **Introduction** ........................................... 149
- **Signal Definitions** .................................. 149
- **Mobile SDRAM** ....................................... 150
- **CPLD Design** .......................................... 150
- **Conclusion** ........................................... 154

## An SMBus/I2C-Compatible Port Expander
- **Summary** ............................................... 155
- **CoolRunner-II Advantages** ......................... 155
- **Using the CoolRunner-II SMBus/I2C Port Expander Design** ................. 155
- **Customizing the Design** ............................ 159
- **Conclusion** ........................................... 159

## Driving LEDs with Xilinx CPLDs
- **Introduction** .......................................... 161
- **Using Xilinx CPLDs to Drive LEDs** ............... 162
- **Guidelines for Driving Multiple LEDs with the Same CPLD** ............... 165
- **Conclusion** ........................................... 165

## CoolRunner-II CPLDs in Cell Phone Handsets/Terminals
- **Introduction** .......................................... 167
- **Quick look at a Handset** ............................ 168
- **Applications** ........................................... 165
- **Cell Phone Chipsets** ................................. 170
- **Picking the Right Mix** ............................... 171
- **MediPhone--A Speculative Example** ............... 172
- **Power and Packaging are Key!** ..................... 173
- **Conclusion** ........................................... 174

## Implementing Keypad Scanners with CoolRunner-II
- **Summary** ............................................... 177
- **Expanding I/O** ........................................ 178
- **Scanning and Encoding** ............................. 178
- **CPLD Design Details** ................................. 179
- **Implementation and Verification** .................... 180
- **Conclusion** ........................................... 180
Level Translation Using Xilinx CoolRunner-II CPLDs

Summary ........................................ 181
Configuring I/O to Use I/O Standards ......................... 182
Using the CPLD as a Level Shifter ........................ 182
Conclusion ......................................... 184

CoolRunner-II Character LCD Module Interface

Summary ........................................ 185
CPLD Implementation .................................. 186
Resource Summary .................................... 189
Conclusion ......................................... 189

Using Xilinx CPLDs to Interface to a NAND Flash Memory Device

Summary ........................................ 191
NAND Interface ...................................... 191
CPLD Design ........................................ 196
CPLD Implementation .................................. 197
Design Verification .................................... 198
Conclusion ......................................... 205

Cell Phone Security Demoboard

Summary ........................................ 207
Demonstration Overview .................................. 207
Setup ................................................. 208
Operation ............................................ 211
Design Information ..................................... 212
Conclusion ......................................... 214

Using CoolRunner-II with OMAP, XScale, i.MX & Other Chipsets

Introduction .......................................... 215
Level Translation ....................................... 215
Pin Expansion ......................................... 216
Pin Swizzling .......................................... 216
Power Control ......................................... 216
Power Reduction ....................................... 216
Logic Consolidation .................................... 217
Conclusion--The Future .................................. 218

Connecting Intel PXA27x Processors to Hard-Disk Drives

Summary ........................................ 219
PXA27x .............................................. 220
Conclusion ......................................... 221
## A Low-Power IDE Controller Design Using a CoolRunner-II CPLD

Summary ................................. 223  
CoolRunner-II IDE Controller Implementation ............................. 224  
Operational Flow Diagram .................................................. 238  
Verilog Test Bench and Functional Simulation ......................... 241  
CoolRunner-II CPLD Implementation ..................................... 242  
Post-Fit Timing Simulation .................................................. 243  
Conclusion .............................................. 244

## Using a Xilinx CoolRunner-II CPLD as a Data Stream Switch

Summary ....................................... 245  
MPEG-2 Data Sources ............................................. 245  
CPLD Background .............................................. 245  
MPEG-2 Multiplexer Design ........................................... 246  
Performance and Utilization ........................................... 247  
Conclusion .............................................. 247

## Supporting Multiple SD Devices with CoolRunner-II CPLDs

Summary ....................................... 249  
CPLD Design .............................................. 250  
Design Verification ............................................. 252  
Device Utilization ............................................. 253  
Voltage and Current Considerations .................................. 253  
Conclusion .............................................. 254

## Chapter 2: Introduction to Low Power Design

## The Real Value of CoolRunner-II DataGATE

Introduction ....................................... 257  
$V_{CCINT}$ Current Savings ............................................. 258  
What About $V_{CCIO}$ Current? ........................................... 260  
Conclusion .............................................. 263

## Power Evaluation Equation for CoolRunner-II CPLDs

Summary ....................................... 265  
Derivation .............................................. 265  
The Equation .............................................. 267  
Conclusion .............................................. 268

## Low Power Design with CoolRunner-II CPLDs

Introduction ....................................... 271  
Power Saving Features .............................................. 271
## Chapter 3: Xilinx Design Software

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Tools</td>
<td>281</td>
</tr>
<tr>
<td>Schematic Capture Process</td>
<td>281</td>
</tr>
<tr>
<td>HDL Design Process</td>
<td>283</td>
</tr>
<tr>
<td>HDL Synthesis</td>
<td>285</td>
</tr>
<tr>
<td>ISE Software</td>
<td>285</td>
</tr>
<tr>
<td>Design Verification</td>
<td>286</td>
</tr>
<tr>
<td>Functional Simulation</td>
<td>287</td>
</tr>
<tr>
<td>Device Implementation</td>
<td>288</td>
</tr>
<tr>
<td>Downloading or Programming</td>
<td>289</td>
</tr>
<tr>
<td>System Debug</td>
<td>290</td>
</tr>
<tr>
<td>Advanced Design Techniques</td>
<td>291</td>
</tr>
<tr>
<td>Embedded SW Design Tools Center</td>
<td>292</td>
</tr>
<tr>
<td>ISE WebPACK Software</td>
<td>292</td>
</tr>
<tr>
<td>Registration and Installation</td>
<td>293</td>
</tr>
<tr>
<td>Module Descriptions</td>
<td>294</td>
</tr>
<tr>
<td>Summary</td>
<td>296</td>
</tr>
</tbody>
</table>

## Chapter 4: WebPACK ISE Design Entry

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Entry</td>
<td>297</td>
</tr>
<tr>
<td>HDL Editor</td>
<td>300</td>
</tr>
<tr>
<td>Functional Simulation</td>
<td>303</td>
</tr>
<tr>
<td>State Machine Editor</td>
<td>307</td>
</tr>
<tr>
<td>Top-Level VHDL Designs</td>
<td>313</td>
</tr>
<tr>
<td>Simulate the Design</td>
<td>319</td>
</tr>
<tr>
<td>Top-Level Schematic Designs</td>
<td>321</td>
</tr>
<tr>
<td>Creating a Top Level Schematic Design</td>
<td>322</td>
</tr>
<tr>
<td>I/O Markers</td>
<td>325</td>
</tr>
<tr>
<td>Simulating the Top Level Schematic Design</td>
<td>327</td>
</tr>
</tbody>
</table>

## Chapter 5: Implementing CPLD Designs

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>331</td>
</tr>
<tr>
<td>Synthesis</td>
<td>331</td>
</tr>
<tr>
<td>Constraints Editor</td>
<td>333</td>
</tr>
<tr>
<td>Implementation</td>
<td>340</td>
</tr>
<tr>
<td>CPLD Reports</td>
<td>341</td>
</tr>
<tr>
<td>Timing Simulation</td>
<td>343</td>
</tr>
<tr>
<td>Configuration</td>
<td>344</td>
</tr>
<tr>
<td>Design Challenge</td>
<td>345</td>
</tr>
</tbody>
</table>
Chapter 6: Introduction to Logic Consolidation

The Advantages of Migrating from Discrete 7400 Logic Devices to CPLDs

Background ................................................. 349
CPLD: The Clear Discrete Logic Replacement Choice ............................................. 352
Discrete Logic versus CPLD Devices ............................................................... 354
Time-to-Market Benefits ............................................................... 357
Programmability: The Real Advantage .......................................................... 357
Reliability ............................................. 358
Electromagnetic Interference ............................................................... 358
Design Security Issues ............................................................... 359
Xilinx CPLD Advantages ............................................................... 359
Summary ............................................................... 360

TTL “Burn Rate” for Xilinx CPLDs

Introduction ............................................................... 361
Basics ............................................................... 362
Examples ............................................................... 369
Burn Rate Table ............................................................... 372
Conclusion ............................................................... 373

Appendix A: Xilinx CPLD Data Sheets

CoolRunner-II CPLD Family

Features ............................................................... 377
Family Overview ............................................................... 377
CoolRunner-II CPLD Family Parameters ............................................................... 377
Architecture Description ............................................................... 379
Function Block ............................................................... 380
Macrocell ............................................................... 381
Advanced Interconnect Matrix (AIM) ............................................................... 381
I/O Block ............................................................... 382
Output Banking ............................................................... 383
DataGATE ............................................................... 383
Global Signals ............................................................... 384
Additional Clock Options: Division, DualEDGE, and CoolCLOCK ............................................................... 385
Design Security ............................................................... 386
Timing Model ............................................................... 387
Programming ......................................................... 388
Power-Up Characteristics ....................................... 388
I/O Banking ............................................................. 389
Mixed Voltage, Power Sequencing, and Hot Plugging ........ 389
Development System Support ..................................... 389
ATE Support ........................................................... 389

XC9500XL High-Performance CPLD Family Data Sheet

Features ............................................................... 393
Family Overview ..................................................... 395
Architecture Description .......................................... 396
Macrocell ............................................................... 396
Function Block ....................................................... 396
Product Term Allocator ............................................ 399
FastCONNECT II Switch Matrix ................................. 402
I/O Block ............................................................... 403
5V Tolerant I/Os ...................................................... 404
Pin-Locking Capability ............................................. 404
In-System Programming .......................................... 405
External Programming ............................................. 405
Reliability and Endurance ......................................... 405
IEEE i49.1 Boundary-Scan (JTAG) ............................... 405
Design Security ...................................................... 406
Low Power Mode .................................................... 406
Timing Model ......................................................... 406
Power-Up Characteristics ......................................... 408
Development System Support .................................... 408
FastFLASH Technology ............................................. 408
About This Handbook

We wrote this handbook to share many of the solutions Xilinx has created supporting digital designers over the last few years. To that end, we have included many of the basic functions that you will find in digital designs, like keyboard and display interfaces; but, we also include solutions to extend beyond basic functionality. For instance, we include a compact flash interface that easily modifies to an IDE disk interface.

Xilinx has become a high volume supplier of products into the consumer electronics arena, with worldwide manufacturing, advanced product planning logistics and world class customer support – in local time zones with local languages. We ship tens of millions of CPLDs each quarter, all over the world.

Although many digital designs are well served by chipset solutions from Texas Instruments, Intel, Freescale and others, these chipsets take time to develop and get “right”. In that time, evolving applications arise making it impossible to deliver to the world wide customer base in real time. Some developers have time horizons of nine months to a year, whereas others develop within two to four months. Being able to respond to the latest application requirements can only be done with programmable logic.

Programmable logic gives you fastest time-to-market and flexible product life cycle management available in the silicon world. There is no other equivalent solution for reducing design time, and nothing that gives you this level of flexibility to respond to changing market requirements.

Incidentally, much of the information we provide was “discovered” by ASIC designers needing to fix their gate array solutions with small, low power CPLDs. This occurred so often that they extended the idea into their product development cycle, to account for late arriving, last minute market requirements. You might say: “there’s always room for CoolRunner™-II.”

Please scan the table of contents to find the applications you need today, or work your way through the handbook following the capabilities of CoolRunner-II CPLDs, delivering low power, inexpensive solutions in tiny packages.

You will find full explanations of CoolRunner-II advanced features like clock dividing and DataGATE. Explanations will show how adding a CoolRunner-II to a design can actually reduce the power being used on the whole board. The application notes are backed up with full designs you can download and use today. The designs are fully documented, allowing you to expand or collapse functionality as required.

We have included the CoolRunner-II family and individual data sheets – the XC2C32A, XC2C64A, XC2C128, XC2C256, XC2C384, and the XC2C512 CoolRunner-II CPLDs. We have also included the XC9500XL (3.3V) family data sheet. You can find all the Xilinx CPLD products on the Xilinx website, including the low power CoolRunner™ XPLA3 3.3V CPLD family, the high performance XC9500XL™ 3.3V CPLD family, and the XC9500™ 5.0V family.
Acknowledgements

This book would not have been possible without the efforts and cooperation of several applications engineers, and at least one FAE. I would like to thank them for their contributions to the ideas, designs, and hard work performed in the creation of application notes and white papers for the digital consumer marketplace. They are:

- Nick Mehta
- Jesse Jenkins
- Mark Ng
- Scott Lien
- Frank Wirtz
- John Hubbard
- Jennifer Jenkins
- Mike Gulotta

Additional appreciation goes to the Xilinx CPLD Marketing team for guidance and the creation of collateral marketing material to support the efforts of the digital consumer initiative. They are:

- Tony Grant
- Roger Seaman
- Betsy Thibault
- LaToya Parker

Additional Resources

To find additional documentation, see the Xilinx website at:


To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

Chapter 1

Introduction to Digital Applications

The chapter contains topics for the implementation of common digital functions, such as smart card readers. You will find white papers and application notes written to assist a digital designer in the creation of new products. The topics listed are among the design applications our CPLDs are fully capable of performing. In most cases, Xilinx provides free reference designs to help speed up your design process. The reference designs can be found at:


This Chapter contains the following topics:

• IrDA and UART Design in a CoolRunner CPLD
• Serial ADC Interface Using a CoolRunner CPLD
• Wireless Transceiver for the CoolRunner CPLD
• CoolRunner-II Smart Card Reader
• CoolRunner-II CPLD I²C Bus Controller Implementation
• CoolRunner-II Serial Peripheral Interface Master
• Design of a Digital Camera with CoolRunner-II CPLDs
• CompactFlash Card Interface for CoolRunner-II CPLDs
• Interfacing to Mobile SDRAM with CoolRunner-II CPLDs
• An SMBus/I²C-Compatible Port Expander
• Driving LEDs with Xilinx CPLDs
• CoolRunner-II CPLDs in Cell Phone Handsets/Terminals
• Implementing Keypad Scanners with CoolRunner-II
• Level Translation Using Xilinx CoolRunner-II CPLDs
• CoolRunner-II Character LCD Module Interface
• Using Xilinx CPLDs to Interface to a NAND Flash Memory Device
• Cell Phone Security Demoboard On The Fly Reconfiguration Technique
• Using CoolRunner-II with OMAP, XScale, i.MX & Other Chipsets
• Connecting Intel PXA27x Processors to Hard-Disk Drives with a CoolRunner-II CPLD
• A Low-Power IDE Controller Design Using a CoolRunner-II CPLD
• Using a Xilinx CoolRunner-II CPLD as a Data Stream Switch
• Supporting Multiple SD Devices with CoolRunner-II CPLDs
Summary

This application note illustrates the implementation of an IrDA and UART system using a CoolRunner™ CPLD. The fundamental building blocks required to create a half-duplex IrDA and full-duplex UART interface design is described. The source code for this design is available and can be found in the section HDL Code, page 11. This design fits an XC2C128 CoolRunner-II or XCR3128XL CPLD.

Introduction

IrDA devices provide a walk-up, point-to-point method of data transfer that is adaptable to a broad range of computing and communicating devices. The first version of the IrDA specification (version 1.0) provides communication at data rates up to 115.2 Kbps. Later versions (version 1.1) extended the data rate to 4 Mbps, while maintaining backward compatibility with version 1.0 interfaces. The protocol described in this application note is only for 115.2 Kbps. The 4 Mbps interface uses a pulse position modulation scheme which sends two bits per light pulse.

The IrDA standard contains three specifications. These relate to the Physical Layer, the Link Access Protocol, and the Link Management Protocol. This document provides information on the Physical Layer and does not provide a detailed explanation of the requirements for full IrDA conformity. For more information on IrDA see "References" on page 12.

IrDA System

Figure 1 illustrates the basic hardware building blocks for IrDA communication. The selection of UART interface, RS232, and microcontroller or microprocessor, depends upon the communication speed required. Data rates above 115.2 Kbps require a direct interface to the address and data lines of the microprocessor or microcontroller. Data rates below 115.2 Kbps can be implemented over a UART or RS232 port.

A UART interface is implemented in this design for data rates up to 115.2 Kbps. The IrDA specification is intended for use with a serial communications controller such as a conventional UART. The data is first encoded before being transmitted as IR pulses. As shown in Figure 2, the serial encoding of the UART is NRZ (non return to zero) encoding. NRZ encoded outputs do not transition during the bit period, and may remain High or Low for consecutive bit periods. This is not an efficient method for IR data transmission with LEDs. To limit the power consumption of the LED, IrDA requires pulsing the LED in a RZI (return to zero, inverted) modulation scheme so that the peak power to average power ratio can be increased. IrDA
IrDA and UART Design in a CoolRunner CPLD

Half Duplex and Latency

The IrDA link cannot send and receive data at the same time. The IrDA link is a half-duplex interface and a time delay must be allowed from when a link stops transmitting until it can receive data again. A time period with a duration of 10 ms must be allowed between transmitting and receiving data. The UART interface design is full-duplex, supporting simultaneous read and write operations from the microprocessor or microcontroller interface.

UART and IrDA Design

Figure 3 illustrates the system architecture for implementing a UART serial port interface with an IrDA module in a CoolRunner CPLD. The UART or a discrete device must provide a 16x clock for the IrDA 3/16 modulation scheme.

The Verilog code provided in this design for the UART interface consists of two HDL modules, TRANSMIT and RECEIVE. Data is written to the transmitter and data is read from the receiver through an 8-bit parallel data bus.

The Verilog code provided in this design for the IrDA emulates the operation of the Agilent Technologies HSDL-7000. The IrDA HSDL-7000 consists of logic for both encoding and decoding the transmit and receive data. Each encode and decode operation is driven by the clock, derived from the UART, or supplied from a discrete source. This clock must be initially configured to cope with the IrDA specified startup data rate of 9.6 Kbps, then adjusted to 16 times the desired baud rate.
UART Interface

Figure 4 illustrates the functionality of the UART interface. The data bus interface to the UART module is 8-bits. Even or odd parity can be selected on the serial data out, SOUT.

![UART Main Interface Logic](X345_04_080701)

The serial data out, SOUT follows the format shown in Figure 5.

![UART Data Out Format](X345_06_080701)

**UART Transmit Logic**

Data transfer in this design is controlled by the system microprocessor or microcontroller. The UART design must interface with the parallel processor bus and necessary control lines. The UART transmit logic consists of interpreting processor write commands, generating the transmit clock, TXCLK, at the desired baud rate, and shifting out data on SOUT. The UART logic must interpret the active Low write signal from the processor and read in data from the data bus. The data is read into the transmit hold register. Once the write signal is de-asserted, a flag is asserted to start shifting data out on SOUT. Figure 6 illustrates the logic of interpreting the write signal.

![Assigning Transmit Data](X345_06_080701)
The second part of control logic for the UART transmitter is dividing the 16x clock to transmit data at the desired baud rate. The transmit clock, TXCLK, is generated using a 3-bit counter that increments on the rising edge of the 16x input clock. TXCLK controls when data changes on the serial data output of the UART. Figure 7 illustrates this logic, TXCLK changes value when the 3-bit counter is equal to zero.

![TXCLK Generate Logic](image1)

The last main portion of the UART transmit logic is shifting out data on SOUT. Figure 8 illustrates the control logic to send data out according to the data format shown in Figure 5. The START TRANSMIT logic sends the start signal out on SOUT. The SHIFT OUT logic shifts the transmit shift register and sends data out on SOUT. When the paritycycle signal is asserted, the parity bit is transmitted. Once the data and parity has been transmitted, the done bit is sent by the STOP OUT logic.

![SOUT Control Logic](image2)

**UART Receive Logic**

The UART receive logic must interpret the incoming data from the IrDA module on SIN, as well as present a parallel byte of data to the microprocessor or microcontroller in the system. To interpret the incoming SIN data, the receive logic must search for the start bit in the data.
stream. The start bit is indicated by an active Low signal for eight clock cycles after a falling edge on SIN.

![UART Receive Logic Diagram]

**Figure 9: UART Receive Logic**

A falling edge on SIN is read by the DETECT EDGE logic as shown in Figure 9. To receive data, the receive clock must be centered on the low leading start bit. The receive clock, RXCLK is generated by dividing the 16x clock using a 4-bit counter.

Once a valid start bit is detected, the data is sampled on SIN at each RXCLK rising edge. The receive shift register is shifted with the incoming SIN data. Running parity is generated with each incoming data bit. When a stop bit is detected, any error flags are set. This includes parity, overrun, and framing error flags.

A main function of the UART receive logic is interfacing with the processor. The CPLD detects a valid edge on the READ signal asserted by the processor. The CPLD then places the received parallel data on the system data bus.
IrDA Interface

Figure 10 illustrates the input and output requirements of the IrDA module in this design. RXD and TXD are the serial connections to the UART SIN and SOUT data lines respectively. IRRXD and TXRXD are the IrDA 3/16th pulse signals that are fed into the LED driver/receiver circuitry as shown in Figure 10.

![IrDA Interface Diagram](image)

Figure 10: IR HDL Block Diagram

The encoding scheme shown in Figure 2 sends a pulse for every space or “0” that is sent on the TXD line. On a High-to-Low transition of the TXD line, the generation of the pulse is delayed for seven clock cycles of the 16XCLK before the pulse is set High for three clock cycles (or 3/16th of a bit time) and then subsequently pulled low.

The decoding scheme shown in Figure 11 seeks a High-to-Low transition of the IRRXD line which signifies a 3/16th pulse. This pulse is stretched to accommodate one bit time (16 clock cycles). Every pulse that is received is translated into a “0” on the RXD line equal to one bit period.

![IrDA Decoding Scheme Diagram](image)

Figure 11: IrDA Decoding Scheme

CoolRunner Implementation

The UART and IrDa design was implemented in Verilog as described above. Xilinx Project Navigator was used for compilation, fitting, and simulation of the design in a CoolRunner CPLD. Xilinx Project Navigator, which includes the ModelTech simulation tool, is available free-of-charge from the Xilinx website at [www.xilinx.com/products/software/webpowered.htm](http://www.xilinx.com/products/software/webpowered.htm). The design was targeted for a 3.3V, 128 macrocell CoolRunner XPLA3 CPLD (XCR3128XL-VQ100). The UART and IrDA design utilization is shown in Table 1. These utilizations were
achieved using certain fitting parameters, actual results may vary. As shown, there is area remaining in the CPLD for the implementation of other logic in the system.

**Table 1: UART and IrDA XPLA3 128-Macrocell Utilization**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Available</th>
<th>Used</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Blocks</td>
<td>8</td>
<td>6</td>
<td>75.0%</td>
</tr>
<tr>
<td>Macrocells</td>
<td>128</td>
<td>88</td>
<td>68.75%</td>
</tr>
<tr>
<td>Product Terms</td>
<td>384</td>
<td>129</td>
<td>33.60%</td>
</tr>
<tr>
<td>Foldback NANDs</td>
<td>64</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>I/O Pins</td>
<td>80</td>
<td>17</td>
<td>21.25%</td>
</tr>
</tbody>
</table>

The Verilog IrDA design can also be targeted as a stand alone module in a 3.3V 32-macrocell CoolRunner XPLA3 CPLD (XCR3032XL). CPLD utilization for implementing the IrDA design is shown in **Table 2**.

**Table 2: Standalone IrDA XPLA3 32-Macrocell Utilization**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Available</th>
<th>Used</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Blocks</td>
<td>2</td>
<td>1</td>
<td>50.0%</td>
</tr>
<tr>
<td>Macrocells</td>
<td>32</td>
<td>14</td>
<td>43.75%</td>
</tr>
<tr>
<td>Product Terms</td>
<td>96</td>
<td>26</td>
<td>27.10%</td>
</tr>
<tr>
<td>I/O Pins</td>
<td>32</td>
<td>4</td>
<td>12.50%</td>
</tr>
</tbody>
</table>

**Design Verification**

The UART/IrDA transmit and receive Verilog design verification has been done through simulation using ModelSim XE in Project Navigator. The design has been verified both functionally and with the timing model generated when fitting in a CoolRunner CPLD. The implemented test bench drove the data, control, and timing necessary to test a transmit operation from the UART to the IrDA output and test the received data from the IrDA and UART modules. Implementation in an actual system may require modification of the control signals used in the source code and test benches provided.

**ModelSim Implementation**

Notes:

Please refer to [XAPP338: Using Xilinx WebPack and ModelTech ModelSim Xilinx Edition](#) as a guide to using ModelSim with Project Navigator. The ModelSim Quick Start demo provides a good first step for getting familiar with ModelSim.

**Figure 12** illustrates the test environment for transmitting a data byte using the UART and IrDA modules. Upon receiving an active WRITE signal, the UART TXRDY signal is asserted. Data is sent to the UART module and transmitted as shown on the SOUT signal. TXCLK is the internal divided clock signal for the UART module. IRTXD is the data transmitted from the IrDA module.
The IR transmitted data is in the form as shown in Figure 2 and includes the start bit, eight data bits, a parity bit, and a stop bit in each data transmission.

**Figure 12: UART and IrDA Transmit Simulation**

Figure 13 illustrates receiving data on the IrDA IRRXD input and presenting the parallel data byte from the UART to the system. The IrDA receive module recognizes the format of incoming data and sends the translated serial stream to the UART, as illustrated in Figure 11 on the SIN.
signal. The UART module shifts the incoming serial data into a holding register. Upon the UART receiving an active READ signal, the UART places the parallel data onto the data bus.

**HDL Code**

THIRD PARTIES MAY HAVE PATENTS ON IRDA. BY PROVIDING THIS HDL CODE AS ONE POSSIBLE IMPLEMENTATION OF THIS DESIGN, XILINX IS MAKING NO REPRESENTATION THAT THE PROVIDED IMPLEMENTATION OF THIS DESIGN IS FREE FROM ANY CLAIMS OF INFRINGEMENT BY ANY THIRD PARTY. XILINX EXPRESSLY DISCLAIMS ANY WARRANTY OR CONDITIONS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, AND XILINX SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE, THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OR REPRESENTATION THAT THE IMPLEMENTATION IS FREE FROM CLAIMS OF ANY THIRD PARTY. FURTHERMORE, XILINX IS PROVIDING THIS REFERENCE DESIGNS "AS IS" AS A COURTESY TO YOU.

Conclusion

IrDA is a low cost, walk-up, point-to-point method of IR communication protocol used in applications ranging from laptops to phones to fax machines. This design is an example implementation of an IrDA interface for data ranges less than 115.2 Kbps connected to a UART interface. Version 1.1 extends the IrDA specification to 4 Mbps and can be implemented using pulse position modulation.

References

1. Infrared Data Association (IrDA).
3. Agilent HSDL-7000 Data Sheet: IR 3/16 Encode/Decode IC.
5. Xilinx Application Note XAPP341: UARTs in Xilinx CPLDs.
6. QuickLogic Application Note: QAN20. Digital UART Design in HDL.
7. Faulkner, Lawrence. IrDA "More than Wireless".
8. Evans, David James. IrDA Applications in CoolRunner CPLDs.

Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>08/08/01</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
<tr>
<td>09/30/02</td>
<td>1.1</td>
<td>Minor edits.</td>
</tr>
<tr>
<td>05/15/03</td>
<td>1.2</td>
<td>Minor corrections.</td>
</tr>
<tr>
<td>12/23/03</td>
<td>1.3</td>
<td>Updated links.</td>
</tr>
</tbody>
</table>
Summary

This document describes the design implementation for controlling a Texas Instruments ADS7870 Analog to Digital Converter (ADC) in a Xilinx CoolRunner™ XPLA3™ CPLD. CoolRunner CPLDs are the lowest power CPLDs available and the ideal target device for controlling a serial ADC in a portable handheld application. This document provides an explanation of the VHDL code for the CoolRunner CPLD.

All related source code is provided for download. To obtain the VHDL code described in this document, go to section VHDL Code Download, page 39 for instructions.

Overview

Figure 1 illustrates the high-level block diagram for the data acquisition system. The system includes an XPLA3 CoolRunner CPLD, a Texas Instruments ADS7870 ADC, and a Toshiba SRAM. The Texas Instrument ADS7870 ADC is initialized and controlled by the CoolRunner CPLD. The CoolRunner CPLD takes conversion data from the ADC and writes the data to SRAM. The SRAM used in the Insight Handspring Springboard development board is a 4 Mbit Toshiba SRAM, TC55V400AFT. The Toshiba SRAM is a 16-bit word size SRAM, and is used for storing data in a conversion cycle. Once conversion data is written into SRAM, the CoolRunner CPLD allows the system processor to access the data.

Figure 1: High Level Block Diagram
Usage

The VHDL code distributed with this document is designed such that minimal knowledge of VHDL language is required. A designated "constants" section of the VHDL code can be edited to specify various aspects of the ADS7870 which include:

- Initialization of internal registers
- Specification of the number of conversions for any (or all) of the eight single-ended channels
- Specification of SRAM locations where conversion results should be written

Designers who do not wish to understand the VHDL code in detail can simply edit this designated VHDL "constants" section, compile the design and program the CoolRunner CPLD. For more information, refer to section High Level Control Logic, page 18.

The following sections will detail the ADS7870 interface for those who wish to understand the VHDL implementation of the CPLD ADC interface.

For a complete Handspring design example, refer to "XAPP146: Designing an 8 Channel Digital Volt Meter with the Insight Springboard Kit".

TI ADS7870

Introduction

The ADS7870 ADC is a low power 12-bit, serial, 8-channel analog to digital converter. The ADS7870 ADC is ideal for portable and handheld applications. The ADS7870 contains an integrated PGA (Programmable Gain Amplifier) as well as a 2.5 MHz clock source (CCLK) that is used internally and can be divided for conversion cycles. The CCLK can be configured as an output for use with multiple ADCs and other system devices. The CoolRunner CPLD uses the CCLK from the ADC as its system clock.

The information presented in this section is provided for convenience. For more information on the Texas Instruments ADC, see References, page 38.

Figure 2 shows a detailed block diagram of the ADS7870.
**Functional Description**

Each functional block shown in Figure 2 in the ADS7870 is described in detail in Table 1.

**Table 1: ADS7870 Functional Blocks**

<table>
<thead>
<tr>
<th>Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MUX</strong></td>
<td>The ADS7870 has eight analog-signal input pins, LN0 through LN7. These input pins are connected to a multiplexer (a network of analog switches). This multiplexer is controlled by four bits in the Gain/Mux register. LN0 through LN7 can each be configured as a single ended input or as a differential input. The M2 bit in the MUX address will enable the user to choose the polarity of the input. The input signal at any of the LN0 through LN7 pins can range between –0.2V and 3.5V.</td>
</tr>
<tr>
<td><strong>Clock Divider/Oscillator</strong></td>
<td>CCLK, the conversion clock, is used by the A/D. CCLK can either function as an input pin (the user supplies an external clock) or an output pin (the ADS7870 will output a 2.5 MHz clock on the CCLK pin and use this signal as its conversion clock). The OSC ENABLE pin controls whether CCLK is an input or an output. When pulled high, CCLK is an output. When OSC ENABLE = “0”, the user may supply an external clock.</td>
</tr>
<tr>
<td><strong>REF (Voltage Reference)</strong></td>
<td>The Voltage Reference block can generate an output voltage of 1.15V, 2.048V, or 2.5V on the VREF pin. In single-ended operation, the Voltage Reference will determine the maximum positive full scale input. For instance if VREF = 2.5V, an input of 2.5V will yield a result of +2047. In differential mode, VREF will determine the center point. Register 7 controls whether the reference is turned on or off. On the Insight Springboard, the VREF pin is tied to the BUFIN pin.</td>
</tr>
<tr>
<td><strong>BUF (Buffer Amplifier)</strong></td>
<td>The Buffer Amplifier takes the internally generated Voltage Reference as an input and outputs it to the A/D block. A Buffer is used in order to increase the output current capability of the VREF pin. The BUFE bit in Register 7 can turn the Buffer on or off. When the buffer is on, the ADS7870 will use the internal reference. If the Buffer is turned off, the ADS7870 will accept an external reference.</td>
</tr>
<tr>
<td><strong>PGA (Programmable Gain Amplifier)</strong></td>
<td>The PGA is a Programmable Gain Amplifier that can amplify the input signal before it is applied to the A/D Block. This is useful if the dynamic range of the input signal is small. The PGA is capable of providing gains of 1, 2, 4, 5, 8, 10, 16, and 20 V/V. The PGA gain is set by bits G2 through G0 of Register 4.</td>
</tr>
<tr>
<td><strong>Serial Interface</strong></td>
<td>The ADS7870 communicates with the CoolRunner though this digital serial port interface. The serial interface is comprised of four pins: SCLK (Serial Data Clock), DIN (Serial Data Input), DOUT (Serial Data Output), and CS (Chip Select). The RISE/FALL pin, also controlled by the CoolRunner, determines whether the ADS7870 will latch serial data on the rising or falling edge of SCLK. In this design, SCLK is active on the rising edge (the CoolRunner device always drives the RISE/FALL pin High).</td>
</tr>
</tbody>
</table>
The ADS7870 has a total of 10 user addressable registers. These registers control various aspects of the ADS7870. For example, the registers can control operation of the A/D, set the PGA gain, or control the Digital I/O pins. A complete list of registers is available on page 16 of the ADS7870 Datasheet.

On the Insight Springboard, the CoolRunner A/D Interface will drive the serial port and will configure and/or read these registers.

Digital I/O

The ADS7870 provides four Digital I/O pins that can independently function as an input or output. All four of these I/O pins are connected to the CoolRunner device.

These Digital I/O pins are configured through the Serial Interface.

A write to Register 6 will configure the Digital I/O pins as inputs or outputs. If any of the pins are configured as outputs, a write to Register 5 will determine whether the pin will output a "1" or a "0". Alternately, if configured as an input, a read from Register 5 will reveal the state of the pin.

12-bit A/D

The serial interface configures and controls operation of the 12-bit A/D Converter. The output of the converter is 2's complement format. This result is stored in registers 0 and 1. These registers are read through the serial interface.

For a plot of Output Codes vs. Input Voltage, refer to Figure 2 on page 10 of the Texas Instruments ADS7870 Data Sheet.

<table>
<thead>
<tr>
<th>Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers and Control</td>
<td>The ADS7870 has a total of 10 user addressable registers. These registers control various aspects of the ADS7870. For example, the registers can control operation of the A/D, set the PGA gain, or control the Digital I/O pins. A complete list of registers is available on page 16 of the ADS7870 Datasheet. On the Insight Springboard, the CoolRunner A/D Interface will drive the serial port and will configure and/or read these registers.</td>
</tr>
<tr>
<td>Digital I/O</td>
<td>The ADS7870 provides four Digital I/O pins that can independently function as an input or output. All four of these I/O pins are connected to the CoolRunner device. These Digital I/O pins are configured through the Serial Interface. A write to Register 6 will configure the Digital I/O pins as inputs or outputs. If any of the pins are configured as outputs, a write to Register 5 will determine whether the pin will output a &quot;1&quot; or a &quot;0&quot;. Alternately, if configured as an input, a read from Register 5 will reveal the state of the pin.</td>
</tr>
<tr>
<td>12-bit A/D</td>
<td>The serial interface configures and controls operation of the 12-bit A/D Converter. The output of the converter is 2's complement format. This result is stored in registers 0 and 1. These registers are read through the serial interface. For a plot of Output Codes vs. Input Voltage, refer to Figure 2 on page 10 of the Texas Instruments ADS7870 Data Sheet.</td>
</tr>
</tbody>
</table>

**ADS7870 Interface**

The ADS7870 has four conventional serial interface pins: SCLK (serial data clock), DOUT (serial data out), DIN (serial data in), and CS (chip select function) as shown in **Figure 3**. A wide variety of microcontrollers can interface to this conventional serial port.

In this particular design, the CoolRunner CPLD is used to handle the serial interface. The condition of the SCLK pin (active level logic "1" or logic "0") is explicitly controlled. The ADC is configured to latch data on the active edge of SCLK by holding a logic "1" to the RISE/FALL*
pin. Thus, the ADC interface ensures that data is available on the DIN pin when SCLK is "0" and holds it when SCLK is "1". Figure 4 illustrates this timing.

![ADC Serial Timing Diagram](image)

Control and configuration of the ADS7870 is accomplished by command bytes written to internal registers through the serial port. Command register device control includes MUX channel selection, PGA gain, and Reference Input control. One must use the “register mode” in order to configure a register.

**Register Mode**

In register mode, the first eight bits transmitted to the ADC specify the address of a particular register, whether to perform a read or a write operation and whether the data will be sixteen bits or eight bits. Immediately after these eight bits are sent, eight or 16 more bits (depending upon what was specified) are sent or received. For a write, data is sent through DIN. For a read, data will appear on the DOUT pin. For a complete list of available registers please refer to the ADS7870 datasheet. The VHDL code in this design allows the user to customize the register usage.

CS must remain Low in order to activate the serial interface. Once CS is brought Low, an internal counter residing in the ADS7870 begins counting the number of active SCLK edges. Raising the CS pin will put the DOUT pin in high impedance and will resynchronize the internal counter. It is possible to keep CS Low throughout an entire chain of serial commands (i.e., write to all address registers), but doing so will require careful management of the serial interface pins. One must be extremely careful when attempting to do so, as one error will cascade throughout the entire sequence.

*Therefore, in this design, and in future designs, the CoolRunner CPLD briefly pulls the CS pin High after the completion of every serial command. This ensures that errors will not cascade.*

**Direct Mode**

A conversion can be initiated on the ADS7870 by issuing a direct mode command. In this mode, a single 8-bit instruction byte is sent. The direct mode command will specify the input channel and the PGA gain. Immediately after this 8-bit instruction is sent, the ADS7870 will perform a conversion on the specified channel, with the specified PGA gain. The results will be written to Registers 0 and 1 and can be retrieved using a register mode read. However, in this design, the ADC is configured to use Read Back Mode 1. In this mode, the conversion result will automatically clock out on the next active edge of SCLK, after the last bit of the direct mode command is sent. Configuring the ADS7870 for Read Back Mode 1 will increase throughput since a separate read instruction is not required to read the result in registers 0 and 1.

---

**CPLD Design**

**Operational Flow**

The CoolRunner CPLD controls the initialization of the ADC and the reading of conversion results. The CoolRunner CPLD then writes the conversion results of each conversion cycle to SRAM. This interface is implemented using two state machines. The state machines control the sending and receiving of parallel data and the configuring of internal ADC registers. After
the CPLD initializes the ADC, it sends multiple "direct mode" instructions to initiate consecutive conversion cycles. The 12-bit serial data in a conversion cycle is read in by the CPLD and deserialized for a 16-bit word write to SRAM.

Figure 5 illustrates at a high level the operational flow for the ADC interface. The CPLD must initialize the ADS7870 registers that are pertinent to the design. This includes specifying each address register and the corresponding data to write. The CPLD then initializes the ADC for performing a "direct mode" conversion cycle for a specific input channel. The CPLD must send the direct mode command before reading out the ADC conversion data. The CPLD brings in the serial data and presents the deserialized data word to SRAM. The CPLD continues to issue the same direct mode command while reading the same input channel on the ADC. To read another input channel on the ADC a different direct mode instruction must be sent. The direct mode instruction includes control bits to specify the input channel on the ADC.

**Figure 5: ADC Interface Operational Flow**

**High Level Control Logic**

The high level control logic for the ADC interface is implemented through the MAIN state machine. The state machine is responsible for sequencing through the following steps:

1. Specify the address of the register to be written.
2. Send the appropriate address over the serial interface.
3. Specify the data to write to the specified register.
4. Send the data via the serial interface to write.
5. Continue steps 1–4 until all registers have been initialized.
6. Specify direct mode instruction for a conversion on a specific input channel.
7. Read data in and deserialize for the conversion cycle.
8. Continue steps 6–7 until all data is read from the specific input channel.
9. Repeat steps 6–8 to read from all input channels that are specified and enabled.

To implement this functionality, the MAIN state machine as shown in Figure 6 has been designed and implemented. During the register mode states, the MAIN state machine specifies the parallel 8-bit data word to write to the ADC. The MAIN state machine loads the 8-bit data register and initiates the go_shift command. The SHIFT state machine, described in Shift Control Logic, page 23, takes the parallel data word and sends data out the serial interface to the ADC. The mode_flag signal is specified in the MAIN state machine for use by the SHIFT state machine.
Table 2 describes the functionality of each state in the MAIN state machine control logic shown in Figure 6. Note the mode_flag = "0" for the SHIFT control logic to designate the shift size for data. When mode_flag = "0", an 8-bit data value is shifted out. This is either a register address,
data to write to an address register, or the direct mode command. When mode_flag = "1", a 16-bit data value is shifted in from the ADC. This is for capturing the conversion data which consists of 12 bits of data, three zero bits, and the overflow bit.

Table 2: MAIN State Machine Description

<table>
<thead>
<tr>
<th>State</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>Initializes specific combinatorial signals.</td>
</tr>
<tr>
<td>WRITE_ADDR</td>
<td>Specifies the address of the register to write to. Loads the 8-bit shift register with this address. Asserts the go_shift signal to the SHIFT control logic to start shifting the address out. This state assigns wr_reg_num which specifies the address currently being written to for use in later states. State also deasserts the enable flag (which is TRUE for a write) for a specific register. Once the flag is disabled, the state machine will not write to that register again.</td>
</tr>
<tr>
<td>WAIT_ADDR</td>
<td>Waits for SHIFT control logic to complete shifting out data to ADC on DIN. The signal shift_done will be asserted to represent this event.</td>
</tr>
<tr>
<td>ADDR_DATA</td>
<td>Checks the value of wr_reg_num to compare which address was specified earlier to the ADC. The data to write to that register is loaded into the 8-bit shift register. Asserts the go_shift signal to the SHIFT control logic to start shifting data out.</td>
</tr>
<tr>
<td>WAIT_DATA</td>
<td>Waits for SHIFT control logic to complete shifting out data to ADC on DIN. The signal shift_done will be asserted to represent this event. Checks to see if any remaining flags are set to TRUE, which indicates more registers need to be written to. The state machine then loops back to the WRITE_ADDR state. If all flags are set to FALSE, the state machine proceeds to the direct mode sequence.</td>
</tr>
<tr>
<td>DIRECT_MODE</td>
<td>Specified the direct mode command to send to the ADC. Loads the 8-bit shift register with this direct mode command. Asserts the go_shift signal to the SHIFT control logic to start shifting the direct command out on DIN to the ADC. Based on direct mode command that is sent (which represents which input channel to read from), the SRAM address pointer is updated.</td>
</tr>
<tr>
<td>WAIT_DM</td>
<td>Waits for SHIFT control logic to complete shifting out the direct mode command to ADC on DIN. The signal shift_done will be asserted to represent when the entire data word has been shifted out.</td>
</tr>
<tr>
<td>ASSIGN_DATA</td>
<td>Sets mode_flag = &quot;1&quot;. This signals the SHIFT control logic to count for 16 SCLK cycles to bring in the conversion data on DOUT. Asserts the go_shift signal to the SHIFT control logic to start counting the incoming data.</td>
</tr>
<tr>
<td>READ_DATA</td>
<td>Waits for SHIFT control logic to assert shift_done to represent when the 16-bit conversion data is done being shifted into the system. Conversion data is written into SRAM at the specified location represented in sram_address. Checks if the sram_address is at the max address space for the specified ADC input channel. If so, then loops back to DIRECT_MODE for the next input channel. If not, then it loops back to DIRECT_MODE for the same input channel. If there remains no ADC input channels to read from, the state machine proceeds to the DONE state.</td>
</tr>
<tr>
<td>DONE</td>
<td>End of state machine. Release control of bus to Handspring Visor.</td>
</tr>
</tbody>
</table>
Customizing the MAIN State Machine

The following is a description for customizing the MAIN state machine VHDL code for a specific application. The designation of register mode is for initializing the ADC registers. This allows the CPLD to configure the ADC. After, the ADS7870 has been configured, the MAIN state machine sends the “direct mode” command. Direct mode represents when the ADC is executing conversion cycles. The ADC will issue a direct mode command and then wait to receive the conversion data for the next 16 clock cycles.

Register Mode

The MAIN state machine continues to remain in the register mode, for initialization, until all registers have been set up and written to. The VHDL code enables the user to specify which registers to write to and the data to write to each register. The following VHDL code illustrates how to specify a write to ADDR3 in the ADC interface VHDL code.

```
constant WR_ADDR3_EN: BOOLEAN := TRUE;

-- Write/Read to Control Register
constant ADDR3: STD_LOGIC_VECTOR(7 downto 0) := '00000011';

-- Data to be written
constant DATA_WR_ADDR3: STD_LOGIC_VECTOR(7 downto 0) := '00000100';
```

Note the variable WR_ADDR3_EN can be set to either TRUE or FALSE, enabling or disabling a write to ADDR3. If WR_ADDR3_EN is set to TRUE, then the data to write to that register must also be specified. This is done in the DATA_WR_ADDR3 constant. In this example, we are writing "0000 0100" to ADDR3, which specifies Read Back Mode 1 (MSB read back first) and sets CCLK division factor = 1. For more information on the data that can be written to each register, refer to References, page 38.

When writing to a register, not only is the register address specified, but additionally a read or write operation and the data word size can be specified.

The data written to the control registers allows the ADC to set up features such as: reading MSB or LSB first, the division factor of CCLK, PGA gain for a specific input channel, enabling the use of the digital I/O, and many more features that can be found in the ADC data sheet.

Direct Mode

Once all the control registers have been initialized in the ADC in the register mode, the ADC can now operate in the direct mode. The direct mode allows the external ADC controller to specify the input channel and read the conversion data. The VHDL code in this design has been constructed to ease the implementation for any application. The VHDL code enables the designer to specify which input channels to read from and how many conversions are requested on each input. The VHDL code for enabling/disabling register mode conversions is similar to the set up for register mode initialization. The following VHDL code illustrates how to perform eight conversions from the ADC single-ended input channel 0 and read eight conversions from the ADC single-ended input channel 1.

```
-- *********** DIRECT MODE CONVERSION SINGLE ENDED CHANNEL 0 ***********
constant DM_SNG_LN0_EN : BOOLEAN := TRUE;
constant DM_SNG_LN0 : STD_LOGIC_VECTOR(7 downto 0) := '10001000';
constant SRAM_OFFSET0 : STD_LOGIC_VECTOR(22 downto 0) := '00000000000000000000000';
constant SRAM_HIGH0 : STD_LOGIC_VECTOR(22 downto 0) := '00000000000000000000111';

-- *********** DIRECT MODE CONVERSION SINGLE ENDED CHANNEL 1 ***********
constant DM_SNG_LN1_EN : BOOLEAN := TRUE;
constant DM_SNG_LN1 : STD_LOGIC_VECTOR(7 downto 0) := '10001001';
```

constant SRAM_OFFSET1 : STD_LOGIC_VECTOR (22 downto 0) := '00000000000000000001000';
constant SRAM_HIGH1    : STD_LOGIC_VECTOR (22 downto 0) := '00000000000000000001111';

The number of conversions in this particular example is controlled by counting each conversion write to SRAM. Therefore, once the address counter for the SRAM reaches the SRAM_HIGH offset, the specified number of conversions are counted. To enable a conversion from a specific input channel, the VHDL constant DM_SNG_LN0_EN set to TRUE will enable multiple conversions from LN0 in the ADC. Note that for only reading from one single-ended input channel, all other input channel enable constants must be set to FALSE. The constant DM_SNG_LN0 stores the value to send a direct mode command for a conversion on LN0. Refer to the ADS7870 data sheet for more information on sending a direct mode command.

**Shift Control Logic**

The shift control logic is initiated by the main control logic state machine. The SHIFT state machine is used for shifting out and shifting in data to/from the ADC. The SHIFT state machine is responsible for sending out data for a register address write, a register data write, and a direct mode instruction write. The shift state machine also shifts in data during the direct mode conversion cycles. For all register and direct mode instructions to the ADC, the data word to write is eight bits. For shifting in the 12-bit conversion data, a 16-bit shift register is used. The data format in a conversion cycle is specified in ADDR3, and includes 12 data bits, three zero bits, and an overflow bit.

The SHIFT control logic is shown in Figure 7. The SHIFT state machine is activated on the rising edge of go_shift. The go_shift signal is asserted from the MAIN state machine and initiates a write or read to/from the ADC. The mode_flag is used to interpret whether the CPLD is writing to a register, sending a direct mode command, or reading in a conversion cycle. The mode_flag signal is equal to "0" during a register or direct mode command and mode_flag = "1" when reading in a conversion cycle.

![SHIFT State Machine Control Logic](image)

**Figure 7: SHIFT State Machine Control Logic**

The SHIFT state machine is responsible for generating the shift clock, SCLK, and setting up the appropriate data to send out. As previously described, the data to send must be on the DIN line before the active edge of SCLK. In the SHIFT state machine, the data register holding the word to send is enabled in the SC0 state (SCLK = "0"). Then in the SC1 state, SCLK = "1" and the data bit is shifted out on DIN.

During a direct mode conversion cycle, the SHIFT state machine controls SCLK. The SHIFT state machine reads in data on the DOUT line at the active edge of SCLK, in the SC1 state.

For more information on implementing these state machines to initialize and read conversion data from the ADC, see section Hardware Implementation, page 28.
Allowing the Visor to Read Conversion Results

After conversion results have been written to SRAM, the Handspring Visor must be given access to read the conversion result from SRAM. This transfer of control occurs once the MAIN state machine has written all conversion results to SRAM. This is specified in the DONE state of the MAIN state machine.

This section will detail how the CoolRunner CPLD releases control of the bus to the Handspring Visor.

On the Insight Springboard Development Kit, all address, data and control signals originating from the Springboard expansion area are routed into the CoolRunner CPLD. These signals are then internally routed to a brand new set of pins, which are then externally connected to the SRAM, A/D, and Flash. XAPP147: “Low Power Handspring Springboard Module Design with CoolRunner CPLDs”, illustrates this routing scheme. In the most basic case, the CoolRunner would simply act as a buffer for all signals, all signals would go directly into and then out of the CPLD, without being manipulated.

In this case, however, the functionality of the CoolRunner has increased because it has the added task of controlling the ADS7870. The CoolRunner must allow both the Visor and the A/D to be able to write (and read) to SRAM. Therefore, the simple interface shown in XAPP147 must be slightly modified to include multiplexers. These multiplexers are controlled by the ADS7870 interface. When the interface is active, the multiplexers allow for the CPLD to write conversion results to SRAM. When conversions are finished, the Visor is allowed to read these conversions from SRAM, or alternately write new values to SRAM.

Data[15:0]

Figure 8 below shows the functionality that would allow for data to be passed to/from the Visor, through the CoolRunner CPLD.

In Figure 8, “SP_D[15:0]” is the name given to the data lines originating from the Visor. “D[15:0]” is the name of the buffered signal. These buffered data lines are routed to the data lines of the SRAM and Flash.

A multiplexer is inserted between the input buffer of “SP_D[15:0]” and the output buffer for “D[15:0]”. The multiplexer’s inputs are “SRAM_Write_Data” and “AD_DATA”. SRAM_Write_Data is a 16-bit signal that represents the data value present on the Springboard Data lines. AD_DATA, is a-16 bit signal that is output from the A/D Interface. AD_DATA is the value of a conversion result.

Mux_Sel, the select line for the multiplexer, controls which of the two inputs will be potentially applied to SRAM and/or Flash. The output value of the multiplexer is not guaranteed to be applied since the output of the MUX is tied to the input of a tri-state buffer. Therefore, the value of the tri-state control signal, “SRAM_Write_Enable” will determine if data will be output.

Figure 8: Data Bus Multiplexing in CoolRunner CPLD
When Mux_Sel is "0" the Handspring Visor will have control of the data lines. Alternately, when Mux_Sel is "1", the A/D Interface is allowed to write data to SRAM. The A/D Interface controls the value of Mux_Sel so that when it is active, the value of Mux_Sel will be "1" and when it is complete, the value will be set to "0".

SRAM_Write_Enable is a tri-state control signal that determines if the "D[15:0]" pins will function as an input or as an output. D[15:0] will function as outputs if the value of SRAM_Write_Enable is a "1". On the other hand, the D[15:0] pins will be inputs if the tri-state control signal is "0".

The SRAM_Write_Enable equation is equal to:

\[(SM\_WE) + (SM\_WE) \& [ WE \& (CS0 + CS1) ]\]

Table 3 describes each literal in the SRAM_Write_Enable equation.

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
</table>
| SM_WE  | Write Enable generated by A/D Interface | "0" when A/D Interface is inactive  
"1" when A/D Interface is active |
| WE     | Write Enable signal generated by Visor | "0" when Visor executes a write  
"1" when others |
| CS0    | Chip Select 0 signal generated by Visor | "0" when Visor writes or reads to CS0  
memory region  
"1" when others |
| CS1    | Chip Select 1 signal generated by Visor | "0" when Visor writes or reads to CS1  
memory region  
"1" when others |

Notes:
1. By default, the CS0 memory region is mapped to address locations 0x28000000 to 0x28FFFFFF. This region corresponds to the Flash address locations.
2. By default, the CS1 memory region is mapped to address locations 0x29000000 to 0x29FFFFFF. This region corresponds to the SRAM address locations.

In the SRAM_Write_Enable equation, the SM_WE literal is generated by the A/D Interface. SM_WE is declared to be "1" when the A/D interface is running, thereby making the entire equation equal to "1". This enables the output buffer and since MUX_Sel is "1" when the A/D Interface is active, the A/D conversion results can be written to SRAM.

When the conversions are complete, the A/D Interface declares SM_WE to be "0" and Mux_Sel to be "0" so that the Handspring can either read the conversion results stored in SRAM or write new data to SRAM.

Once the Visor is given control of the bus (i.e., SM_WE becomes "0"), the Visor can enable the output buffer by executing a write to SRAM. A write operation to an address between 0x29000000 and 0x29FFFFFF (the default memory mapped region for CS1) will cause CS1 and WE to become "0", making the SRAM_Write_Enable equation true.

If needed, the Visor may also write to the Flash memory region that corresponds to CS0 (address 0x28000000 to 0x28FFFFFF). Doing this will create a falling edge on CS0 and WE.

The Visor can retrieve contents in SRAM by executing a read operation. Once again, an output buffer will determine if the SP_D[15:0] pins will provide data to the Visor or if these pins will send data to external components. This output buffer is controlled by the SRAM_Read_Enable signal.

The equation for SRAM_Read_Enable is equal to:

\[OE \& (CS0 + CS1)\]

If the Visor executes a read operation from SRAM, the CS1 and OE signals will go Low. The SP_D[15:0] pins will then be allowed to output data to the Visor.
Address Lines

The address lines originating from the Springboard Expansion Connector, “SP_A[23:1]”, are routed through the CoolRunner. The SP_A[23:1] pins are connected internally to one input of the address multiplexer, as shown in Figure 9. This Multiplexer has two inputs, one of which is SP_A[23:1] and the other which is ADDR_COUNT. MUX_SEL, the same select signal for the other multiplexers in this design, is used for the select line of this multiplexer.

A[23:1] is the output of this switch. This output bus is tied to external pins which are then routed to the address lines of Flash and SRAM. Figure 9 illustrates this.

When the A/D Interface is active, MUX_SEL is set to "1". This allows the value of SM_ADDRESS to determine the value of A[23:1].

The VHDL signal SM_ADDRESS is assigned for each write to SRAM. The value of SM_ADDRESS is initialized for a specific input channel as specified in the VHDL "constants" section discussed in section, Direct Mode, page 22. This address counter, SM_ADDRESS is increment for each subsequent write to SRAM. The ADC will stop reading from the current input channel once the SM_ADDRESS counter reaches the max address space for the current input channel.

The Digital Volt Meter design shown in Hardware Implementation, page 28 writes to address locations 0, 1, and 2 of SRAM for the ADC input channel 0.

Chip Select 1

Figure 10 shows the Chip Select 1 multiplexer. A switch is needed in order to give the A/D Interface the ability to control the CS pin of the SRAM. The SRAM CS pin is an active Low signal that enables the SRAM chip. When CS is Low and RW (Write Enable) is Low, data will be written to SRAM. When CS is Low and OE (Output Enable) is Low, the SRAM will output data so a read operation can occur.

SPRING_CHIP1_ENn is the CS1 pin originating from the Visor’s expansion area. STATE_MACHINE_SRAM_ENn is an internal signal that is controlled by the A/D Interface. The SRAM_CHIP1_ENn signal is externally routed to the CS pin of the SRAM.

When the A/D Interface is active, MUX_SEL is "1" and hence the value of STATE_MACHINE_SRAM_EN determines the value of the CS pin on the SRAM. When the A/D Interface writes a conversion result to SRAM, it pulls the STATE_MACHINE_SRAM_EN signal and the STATE_MACHINE_WE (see next page) signal Low.
After the conversions are complete, MUX_SEL is set to "0" and the Handspring Visor can once again perform its own read and write operations.

Write Enable

Figure 11 shows the Write Enable Multiplexer. This multiplexer is needed in order to give the A/D Interface the ability to control the RW (Write Enable) pin of SRAM. The RW pin is an active low signal that, when used in conjunction with the CS pin, will enable a write operation to occur.

SPRING_WRITE_ENn is the Write Enable pin originating from the Visor's expansion area. STATE_MACHINE_WE is an internal signal that is controlled by the A/D Interface. The output of the multiplexer, READ_WRITEn is externally routed to the RW pin of the SRAM.

When the A/D Interface is active, MUX_SEL is "1" and the value of STATE_MACHINE_WE will determine the value of the SRAM RW pin. When the A/D Interface writes a conversion result to SRAM, it pulls the STATE_MACHINE_WE signal Low and the STATE_MACHINE_SRAM_ENn signal low. (See Chip Select 1, page 26 for an explanation of the STATE_MACHINE_SRAM_ENn signal.)

After the conversions are complete, MUX_SEL is set to "0" and the Handspring Visor can once again perform its own read and write operations.

Output Enable

Figure 12 shows the Output Enable multiplexer. This multiplexer is needed in order to give the A/D Interface the ability to control the OE (Output Enable) pin of SRAM. The OE pin is an active low signal that, when used in conjunction with the CS pin, will allow a read operation to occur.

SPRING_OUTPUT_ENn is the Output Enable pin originating from the Visor’s expansion area. STATE_MACHINE_OE is an internal signal that is controlled by the A/D Interface. The output of the multiplexer, OUTPUT_ENn is externally routed to the OE pin of the SRAM.

When the A/D Interface is active, MUX_SEL is "1" and the value of STATE_MACHINE_OE will determine the value of the SRAM OE pin. When the A/D Interface writes a conversion result to SRAM, it pulls the STATE_MACHINE_OE signal Low and the STATE_MACHINE_SRAM_ENn signal low. (See previous page for an explanation of the STATE_MACHINE_SRAM_ENn signal.)
After the conversions are complete, MUX_SEL is set to "0" and the Handspring Visor can once again perform its own read and write operations.

![Output Enable MUX](image)

**Figure 12: Output Enable MUX**

### Hardware Implementation

#### Usage Example

The following section is provided as an example for modifying the VHDL code to target a specific application. Assume that in this application, the user wants to configure ADDR3, ADDR5, ADDR6, and ADDR7 and that the ADS7870 must perform a conversion on all eight input channels.

The following VHDL register "constants" have been edited for the following hardware implementation. Note we are writing "0000 0100" to ADDR3, "0000 0101" to ADDR5, "0000 1111" to ADDR6, and "0011 1100" to ADDR7.

In the VHDL direct mode "constants" section, flags can be set to enable a single ended conversion on a specific input channel of the ADC. For example, the DM_SNG_LN0_EN constant is set to TRUE to enable a single ended conversion on input channel 0. To specify the SRAM address space for each input channel, the SRAM_OFFSET0 constant is set to "00000000000000000000000000000000000000000111". This represents that eight samples of channel 0 will be written to SRAM. Due to the pipelined nature of the ADC, the conversion data stored at address 0 should be discarded. Therefore, SRAM address 1 will store the first sample of channel 0. Also note, the SRAM address specified in the VHDL code is 23 bits wide. This is because the Springboard address 0 (A0) is set to 0. This means that A0 is appended to the SRAM address, and data is written to SRAM locations 0, 2, 4, etc.

```vhdl
--************************* ADDR0  (ADC OUTPUT REGISTER) *************************
-- Description: ADDR0 stores the LS Byte of the conversion result.
-- R/W : READ ONLY
constant RD_ADDR0_EN : BOOLEAN := FALSE;
constant ADDR0    : STD_LOGIC_VECTOR(7 downto 0) := '01000000'; -- Read ADDR 0

--************************* ADDR1  (ADC OUTPUT REGISTER) *************************
-- Description: ADDR1 stores the MS Byte of the conversion result
-- R/W : READ ONLY
constant RD_ADDR1_EN : BOOLEAN := FALSE;
constant ADDR1    : STD_LOGIC_VECTOR(7 downto 0) := '01000001'; -- Read ADDR 1

--************************* ADDR2  (PGA VALID REGISTER) *************************
-- Description: ADDR2 reveals if PGA has exceeded allowable values
-- R/W : READ ONLY
constant RD_ADDR2_EN : BOOLEAN := FALSE;
constant ADDR2    : STD_LOGIC_VECTOR(7 downto 0) := '01000010'; -- Read ADDR2

--************************* ADDR3  (A/D CONTROL REGISTER) *************************
-- Description: ADDR3 configures CCLK Divider and read back mode operation
-- R/W : R/W
constant WR_ADDR3_EN : BOOLEAN := TRUE;
```
-- Write/Read to Control Register
constant ADDR3 : STD_LOGIC_VECTOR(7 downto 0) := '00000011';

-- Data to be written
constant DATA_WR_ADDR3: STD_LOGIC_VECTOR(7 downto 0) := '000000100';

--********************* ADDR4  (GAIN/MUX REGISTER)    ****************
-- Description: ADDR4 configures the PGA gain and the input channel
-- selection. (A direct mode operation will accomplish this as well)
-- R/W      :  R/W
constant WR_ADDR4_EN: BOOLEAN := FALSE;

-- Write/Read to Gain/Mux Register
constant ADDR4 : STD_LOGIC_VECTOR(7 downto 0) := '00000100';

-- Data to be written
constant DATA_WR_ADDR4: STD_LOGIC_VECTOR(7 downto 0) := '00000000';

--****************** ADDR5  (DIGITAL I/O STATE REGISTER) ***************
-- Description:  ADDR5 sets/reveals the state of the digital IO pins.
-- R/W      :  R/W
constant WR_ADDR5_EN: BOOLEAN := TRUE;

-- Write/Read Digital I/O State Reg
constant ADDR5  : STD_LOGIC_VECTOR(7 downto 0) := '00000101';

-- Data to be written
constant DATA_WR_ADDR5: STD_LOGIC_VECTOR(7 downto 0) := '00000101';

--**************** ADDR6  (DIGITAL I/O CONTROL REGISTER) **************
-- Description: ADDR6 determines whether each of the four IO pins will be
-- an output or and output
-- R/W      :  R/W
constant WR_ADDR6_EN: BOOLEAN := TRUE;
constant ADDR6  : STD_LOGIC_VECTOR(7 downto 0) := '00000110';
constant DATA_WR_ADDR6: STD_LOGIC_VECTOR(7 downto 0) := '00001111';

--**************** ADDR7  (REF/OSCILLATOR CONTROL REGISTER)************
-- Description: ADDR7 determines:
-- a) Whether the internal oscillator is used for the conversion clock
-- b) Whether the internal voltage reference and buffer are ON or OFF
-- c) Whether the voltage reference is 2.5V, 2.048V or 1.15V
-- R/W      :  R/W
constant WR_ADDR7_EN: BOOLEAN := TRUE;
constant ADDR7  : STD_LOGIC_VECTOR(7 downto 0) := '00000111';
constant DATA_WR_ADDR7: STD_LOGIC_VECTOR(7 downto 0) := '00111100';

--*************** ADDR24 (SERIAL INTERFACE CONTROL REGISTER) **********
-- Description: ADDR24 allows certain aspects of the serial interface to be
-- changed by the user
-- R/W      :  R/W
constant WR_ADDR24_EN: BOOLEAN := FALSE;

-- Serial Interface Control
constant ADDR24 : STD_LOGIC_VECTOR(7 downto 0) := '00011000';
constant DATA_WR_ADDR24: STD_LOGIC_VECTOR(7 downto 0) := '00000000';

--******************** ADDR31 (ID REGISTER)   ********************
-- Description: ADDR31 reveals which version of ADS7870 is being used
-- R/W      :  READ ONLY
constant WR_ADDR31_EN: BOOLEAN := FALSE;

-- ID Register
constant ADDR31 : STD_LOGIC_VECTOR(7 downto 0) := '00011111';

-- ********** DIRECT MODE CONVERSION SINGLE ENDED CHANNEL 0 **********
constant DM_SNG_LN0_EN : BOOLEAN := TRUE;
constant DM_SNG_LN0 : STD_LOGIC_VECTOR(7 downto 0) := '10001000';
constant SRAM_OFFSET0 : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000000000';
constant SRAM_HIGH0   : STD_LOGIC_VECTOR (22 downto 0) :=
                      '000000000000000000000111';

-- ********** DIRECT MODE CONVERSION SINGLE ENDED CHANNEL 1 **********
constant DM_SNG_LN1_EN : BOOLEAN := TRUE;
constant DM_SNG_LN1 : STD_LOGIC_VECTOR(7 downto 0) := '10001001';
constant SRAM_OFFSET1 : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000001000';
constant SRAM_HIGH1   : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000001111';

-- ********** DIRECT MODE CONVERSION SINGLE ENDED CHANNEL 2 **********
constant DM_SNG_LN2_EN : BOOLEAN := TRUE;
constant DM_SNG_LN2 : STD_LOGIC_VECTOR(7 downto 0) := '10001010';
constant SRAM_OFFSET2 : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000010000';
constant SRAM_HIGH2   : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000010111';

-- ********** DIRECT MODE CONVERSION SINGLE ENDED CHANNEL 3 **********
constant DM_SNG_LN3_EN : BOOLEAN := TRUE;
constant DM_SNG_LN3 : STD_LOGIC_VECTOR(7 downto 0) := '10001011';
constant SRAM_OFFSET3 : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000011000';
constant SRAM_HIGH3   : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000011111';

-- ********** DIRECT MODE CONVERSION SINGLE ENDED CHANNEL 4 **********
constant DM_SNG_LN4_EN : BOOLEAN := TRUE;
constant DM_SNG_LN4 : STD_LOGIC_VECTOR(7 downto 0) := '10001100';
constant SRAM_OFFSET4 : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000100000';
constant SRAM_HIGH4   : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000100111';

-- ********** DIRECT MODE CONVERSION SINGLE ENDED CHANNEL 5 **********
constant DM_SNG_LN5_EN : BOOLEAN := TRUE;
constant DM_SNG_LN5 : STD_LOGIC_VECTOR(7 downto 0) := '10001101';
constant SRAM_OFFSET5 : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000101000';
constant SRAM_HIGH5   : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000101111';

-- ********** DIRECT MODE CONVERSION SINGLE ENDED CHANNEL 6 **********
constant DM_SNG_LN6_EN : BOOLEAN := TRUE;
constant DM_SNG_LN6 : STD_LOGIC_VECTOR(7 downto 0) := '10001110';
constant SRAM_OFFSET6 : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000110000';
constant SRAM_HIGH6   : STD_LOGIC_VECTOR (22 downto 0) :=
                      '00000000000000000110111';

-- ********** DIRECT MODE CONVERSION SINGLE ENDED CHANNEL 7 **********
Serial ADC Interface Using a CoolRunner CPLD

```vhdl
constant DM_SNG_LN7_EN : BOOLEAN := TRUE;
constant DM_SNG_LN7    : STD_LOGIC_VECTOR(7 downto 0) := '10001111';
constant SRAM_OFFSET7  : STD_LOGIC_VECTOR (22 downto 0) :=
  '00000000000000000111000';
constant SRAM_HIGH7    : STD_LOGIC_VECTOR (22 downto 0) :=
  '00000000000000000111111';
```

**ADC Initialization (Register Mode)**

Using these VHDL "constants", the following ADC interface will be implemented:

1. Write data to Address 3, the “ADC Control Register” (ends with first rising edge of CS).
2. Write data to Address 6, the “Digital I/O Control Register” (ends with second rising edge of CS).
3. Write data to Address 5, the “Digital I/O State Register” (ends with third rising edge of CS).
4. Write data to Address 7, the “Reference Oscillator Register” (ends with fourth rising edge of CS).
5. Initiate three consecutive conversions on ADC input channel 0.

Note that CS goes High and SCLK temporarily stops in between commands (i.e., whenever data has been written to ADDR3, ADDR6, and ADDR5). This is done because a rising edge on CS will resynchronize the serial interface.

Note that in the beginning, DOUT tends to follow the CS pin. This is expected because of two factors: first, the DOUT pin enters high impedance when CS is held High and second, the DOUT pin is externally pulled up.

**Step 1: Writing to ADDR3**

Upon reset, the state machine will execute a register mode write to Address 3, the ADC Control Register (See ADS7870 Datasheet). A value of DIN = “0000 0100” written to ADDR3 will configure the A/D for Read Back Mode 1. In this mode, the serial interface configures itself to clock out a conversion result as soon as a conversion is started. A read instruction is not required to retrieve the result, thereby increasing the throughput rate by saving eight SCLK cycles. The very first data read back will be discarded, but subsequent values pipeline the conversion and readback activities.

This sequence requires a total of 16 SCLK cycles — eight bits to specify the Address, and eight more to write data to that address. After these 16 bits are sent, the state machine will enter a
wait state to resynchronize the serial interface. In this wait state, SCLK remains Low, and CS is momentarily raised High. Figure 13 shows a logic analyzer trace of this sequence.

Step 2: Writing to ADDR6

After exiting a wait state, the state machine executes a register mode write to Address 6, the Digital I/O Control Register (See page 19 of A/D Datasheet). The ADS7870 configures all four digital I/O pins as outputs by writing a data value of “0000 1111”.

As above, 16 more SCLK cycles are required, followed by a wait state, to resynchronize the serial port. Figure 14 shows a logic analyzer trace of this sequence.

Figure 13: Writing to ADDR3
Step 3: Writing to ADDR5

Next, the value of each I/O pin is configured to output a "1" or a "0". A pattern of DIN = “0000 0110” is transmitted to initiate an 8-bit write to Address 5, the Digital I/O State Register. The ADS7870 will then output a “0” on I/O1, a “1” on I/O2, a “0” on I/O3, and a “1” on I/O4 by sending “0000 0101” on the next sequence on DIN.

In this example, this test case is used with the Insight Handspring Development Board. Since all four Digital I/O pins are routed into the CoolRunner CPLD, they are used to control the four LEDs on the Insight Springboard Development Card. If the serial interface is working properly, the LEDs should read On, Off, On, Off.

Writing to this register requires another sixteen SCLK cycles and a wait state. Note that it is not absolutely necessary to write to ADDR6 and ADDR5. These two registers do not affect the conversion result. However, these registers have been configured to illustrate how to use the serial interface. In addition, they provide a convenient way to check the serial interface through the LEDs. A logic analyzer trace of this sequence is provided below in Figure 15.

Figure 14: Writing to ADDR6
Step 4: Writing to ADDR7

The Reference/Oscillator Register, ADDR7, configures the reference and the buffer (page 19 of ADS7870 Datasheet). After a pattern of “0000 0111” is sent to specify an 8-bit write to ADDR7, a sequence of “0011 1100” is written to this register. The OSCR and OSCE bits are now set to “1”. Enabling the OSCE bit will power the internal oscillator, and CCLK will output a 2.5 MHz signal. Setting the OSCR bit configures the ADS7870 to use this 2.5 MHz internal clock for the reference. The REFE and BUFE bits are also enabled. This turns on the Reference and the Buffer. And finally, by setting R2V and RBG bits to “0”, VREF is set to 2.5V. This sets the maximum full-scale input to 2.5V in single ended.
mode. Sixteen more SCLK cycles and a wait state are needed, and a logic analyzer trace of this sequence is shown in Figure 16.

![Image of logic analyzer trace](image)

**Figure 16: Writing to ADDR7**

**Direct Mode Conversions**

**Direct Mode Command 1**

At this point, all registers have been properly configured, and the state machine is ready to send the first direct mode command to initiate a single conversion. Assuming that the LN0 input (an analog input of the ADS7870) is tied to the voltage site (test point), eight bits, “1000 1000” are sent through the DIN pin. This commands the A/D to start a conversion on input channel LN0, which has been configured as single ended, with the PGA (Programmable Gain Amplifier) gain set to “1”.

Since Address 3 is configured for Read Back Mode 1, the ADS7870 will begin clocking out the result of the previous conversion immediately after the 8-bit direct mode command. Therefore, 16 more SCLK cycles are sent to the ADS7870. Thus, for this entire sequence, a total of 24 SCLK cycles are needed, eight for the direct mode command, and 16 for the result.

Note however, that on the last 16 clock cycles, DOUT remains Low. This is expected. Remember that the first result coming out of the ADS7870 is always invalid, due to the fact that the result is from the previous conversion.

**Figure 17** actually shows more than 24 SCLK cycles instead of the 16 SCLK cycles that have been shown in the previous figures. This is done in order to show BUSY going High and then Low after the first direct mode command.
In reality, when we are chaining several conversions together, the CoolRunner does not need to monitor the BUSY pin. BUSY is shown just to confirm that a conversion is taking place.

![Figure 17: Direct Mode Command 1](image)

**Direct Mode Command 2**

The second direct mode command is issued on the next rising edge of SCLK (i.e., on the 25th clock edge starting from when the first direct mode command sequence was issued).

Again, 24 SCLK cycles are needed for this second frame. The same 8-bit direct mode command of “1000 1000” is sent, but this time, notice that DOUT is driving data. This data is
the result of the first conversion. The result of this second conversion is returned in the next frame as shown in Figure 18.

Within the frame of this example, DOUT reads “0000 1001 1011 0000”. Since the ADS7870 is set for Read Back Mode 1, the MS Byte of the conversion result is returned first. In other words, ADDR1 will clock out first, followed by ADDR0. (The Texas Instruments ADS7870 datasheet provides details of ADDR1 and ADDR0).

Table 4: Contents of ADDR1, the MS Byte

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC11</td>
<td>ADC10</td>
<td>ADC9</td>
<td>ADC8</td>
<td>ADC7</td>
<td>ADC6</td>
<td>ADC5</td>
<td>ADC4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5: Contents of ADDR2, the LS Byte

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC3</td>
<td>ADC2</td>
<td>ADC1</td>
<td>ADC0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>OVR</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The 12-bit output code in this example is “0000 1001 1011”. This is equal to +155. The corresponding measured voltage would then equal:

\[
(155 / 2047) \times 2.5 \approx 0.189 \text{ Volts}
\]

It may also be of interest to see that this second direct mode command was issued when the first conversion was still in progress (Note the BUSY pin). The ADS7870 places this next
conversion in queue and allows the current conversion to finish. Maximum throughput is obtained through this method, as the next conversion will begin immediately after the previous one finished. Again, note how the BUSY pin goes low then high during the conversion cycle.

Direct Mode Command 3

Figure 19 shows the third direct mode command. Like the previous direct mode command, this frame initiates a third consecutive conversion and retrieves the result of the second conversion.

![Figure 19: Direct Mode Command 3](image)

Note, only three direct mode conversion cycles are shown for single ended input channel LN0. The implemented design allows for eight direct mode conversion cycles on each input channel of the ADC.

Conclusion

The ADS7870 interface presented in this document is an easy to use reference design that will allow for quick customizing of the Insight Springboard Development Card. Regardless of whether a designer understands the VHDL language, the designated “constants” section of the VHDL code can be modified to configure the ADS7870 in a way that best complements a specific Springboard design. After modification, simply implement the design and program the CoolRunner CPLD. The inherent low power characteristics of the CoolRunner CPLD will come at no cost, and users will recognize the advantages of programmable logic.

References

VHDL Code Download

VHDL source code and test benches are available for this design. THE DESIGN IS PROVIDED TO YOU "AS IS". XILINX MAKES AND YOU RECEIVE NO WARRANTIES OR CONDITIONS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, AND XILINX SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. While this design has been verified on hardware, it should be used only as an example design, not as a fully functional core. XILINX does not warrant the performance, functionality, or operation of this design will meet your requirements, or that the operation of the design will be uninterrupted or error free, or that defects in the design will be corrected. Furthermore, XILINX does not warrant or make any representations regarding use or the results of the use of the design in terms of correctness, accuracy, reliability or otherwise.


Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>09/25/01</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
<tr>
<td>01/03/02</td>
<td>1.1</td>
<td>Minor revisions.</td>
</tr>
</tbody>
</table>
Summary

This document focuses on the design of a wireless transceiver using an XPLA3™ CoolRunner™ CPLD. The wireless transceiver is implemented using the CoolRunner XPLA3 demo board from Insight Electronics. The wireless transceiver is the perfect application of the low power capabilities of a CoolRunner CPLD. To obtain the VHDL code described below go to the section titled “VHDL Disclaimer and Download Instructions” on page 11.

Introduction

A wireless transceiver consists of two modules; receive, and transmit. One CoolRunner demo board comprises the receive portion while the second demo board comprises the transmit portion. The design transmits the text string “CooLrunnEr,” which is displayed on both the transmit and receive demo boards. The wireless communication is controlled by an RF module designed by RF Monothilics, Inc. (RFM®).

The protocol designed for the wireless transceiver obeys a custom wireless communication protocol. A designer could change the protocol has needed to meet the needs of a specific application.

The addition of keyboard control is also covered in this document. The VHDL code is not provided for this portion of the design. With keyboard control, a user can enter a text string into the transmitter and the string would be display on the receive side of the transceiver. The keyboard described is manufactured by Fujitsu Takamisawa America, Inc. (FBK7603) (www.fujitsu.takmisawa.com/pdf/EvalKits.pdf).

Figure 1: CoolRunner Wireless Transceiver

CoolRunner CPLD Transceiver Operation

This section describes the operation of the transceiver. The communication protocol is a custom transmit and receive scheme, using Manchester encoding and Bit-Oriented Protocol (BOP) theory.

Communication Protocol

The communication protocol is show in Figure 2. The preamble and postamble are used to contain the data to be transmitted. The total transmission is 36 bits. For error checking, the data is transmitted four times and compared to insure the proper data was received.
A Manchester encoding scheme is used between the transmit and receive modules. Manchester coding ensures that each bit of the data is D.C. balanced. Also, this coding scheme provides an edge within each bit period that can be used to align the receiver’s clock if needed. However, Manchester coding requires twice the bandwidth as compared to NRZ (Non-Return-to-Zero) codes. To decrease bandwidth, a symbol table is used. It consists of sixteen different symbols that can be generated using six bits which guarantees that no more than four consecutive bits are the same. This scheme requires only 1.5 times the bandwidth when compared with NRZ coding. For more information on Manchester and NRZ coding schemes, refer to the application note XAPP339 “Manchester Encoder-Decoder for Xilinx CPLDs” (http://www.xilinx.com/xapp/xapp339.pdf).

BOP is utilized on the receive side of the transceiver. BOP takes advantage of opening and closing flag insertion and deletion and zero bit insertion and deletion. Once an edge is detected, the incoming data is sampled and stored in a shift register. Once the most significant bits are equal to the postamble, the 12-bit data is stored in a register. This process occurs four times. This ensures the data has time to be displayed on the LCD of the CPLD demo board and allows for more accurate error checking.

The transmit block diagram is shown in Figure 3. Transmission comprises of three VHDL entities; DISPLAY_COUNT, SHIFT_ENABLE, and SHIFT_OUT. These three logic modules are controlled by the top level module, TX_MODULE. DISPLAY_COUNT controls the LCD common line, LCD_COM, which minimizes charging in the LCD. DISPLAY_COUNT also controls the time between display states. Each state determines which two digits are displayed on the LCD. It pulses the SWITCH_EN_H signal when it is time to change to the next state. This control line tells the SHIFT_ENABLE module to output the next state number, CUR_STATE, to the CHANGE_STATE look up table. When this is completed, it pulses the LOAD_DATA_H signal to tell the SHIFT_OUT module to load the current state data, CUR_STATE_DATA, output by the CHANGE_STATE look up table. This module also keeps track of how many transmissions have been sent. It pulses the LOAD_DATA_H signal four times for each state, controlling the time between transmissions. The data is sent four times to provide error checking on the receive side (See Receive). When SHIFT_OUT observes that LOAD_DATA_H has been pulsed, it loads the current state data, and begins to send the data, with a preamble and postamble, one bit at a time, to the RF module.
The CONTROL signal is controlled by the TX MODULE which enables the RF MODULE to be in transmit mode. SYS_CLK_H and SYS_RST_L are external signals that are used as the system clock and the global system reset.

Receive

The receive block diagram is shown in Figure 4. The data is read on the RX pin and shifted into a 3-bit shift register, RXIN, on every clock cycle. When an edge is detected (a logic 1) in the least significant bits of RXIN, a counter is enabled. This counter counts to approximately 3/4 of the bit period (due to non-ideal conditions, see Figure 5), samples the data, and shifts the bit into a 36-bit data register, SHIFT_DATA (see Figure 10). If there are consecutive bits in the stream, the counter continues to count 3/4 into the next bit period and samples the data again. If there is another edge detected, it restarts the counter, to keep the possibility of error due to drift to a minimum. Once the postamble is seen in the most significant 12 bits of the 36-bit shift register, the 12 bits of data are stored into a temporary register, REG1 through REG4, and the module gets ready for the next transmission. After the fourth transmission, if any two of the temporary registers are equal, the data is symbolized using the RX_SYMBOLIZE function, and the data is sent to the LCD.

LCDCOM minimizes charging in the LCD. The CONTROL signal is controlled by the receive MODULE which enables the RF MODULE to be in receive mode. SYS_CLK_H and SYS_RST_L are external signals that are used as the system clock and the global system reset.
Figure 4: Receive Module Block Diagram

Ideal Sampling Illustration

Sample Period (1/2 Bit Period)

Non-Ideal Sampling Illustration

Sample Period (~3/4 Bit Period)

Figure 5: Receive Module Block Diagram
CPLD Transmit Design

Transmit module contains the look up tables: CHANGE_STATE, RX_SYMBOLIZE, BIN7SEG. The latter two are used to display the letters being transmitted. CHANGE_STATE changes the current state of TX_MODULE (the data to be transmitted), which is sent from the SHIFT_ENABLE logic module. The logic function RX_SYMBOLIZE is a look up table to convert 6-bits of each digit of data into a 4-bit number. BIN7SEG is a lookup table that takes the 4-bit symbolized number from the RX_SYMBOLIZE function and converts it into an 8-bit number sent to the LCD digits. The block diagram for TX_MODULE is shown in Figure 6.

![TX_MODULE Block Diagram](image)

Figure 6: TX_MODULE Block Diagram

Display Count

The DISPLAY_COUNT block diagram is shown in Figure 7. This logic module controls the time between each state and the LCDCOM signal. STATE_COUNT is incremented and then enables SWITCH_EN_H. SWITCH_EN_H then enables the logic module SHIFT_ENABLE to change state (transmit new data).

![Display Count Block Diagram](image)

Figure 7: Display Count Block Diagram
**Shift Enable**

The SHIFT_ENABLE logic module increments the state variable to change states, and sends an edge to an enable signal (LOAD_DATA_H) to update a register in the SHIFT_OUT module with the new state value. The block diagram is shown in Figure 8.

TRANS_BETWEEN_COUNT determines the time between each state. TRANS_COUNT controls the number of transmissions between states.

![SHIFT_ENABLE Block Diagram](image)

**Figure 8: SHIFT_ENABLE Block Diagram**

**Shift Out**

The SHIFT_OUT logic module sends the TX_DATA to TX_MODULE for transmission. LOAD_DATA_H enables the SHIFT_OUT module to load the current data. The block diagram is shown in Figure 9.

![SHIFT_OUT Block Diagram](image)

**Figure 9: SHIFT_OUT Block Diagram**

**Receive Module**

**Edge Detection**

The receiver operation is included in one receive VHDL entity shown in Figure 4. Figure 10 shows the edge detection and sampling scheme of the ideal sampling model. Once an edge is detected, a counter insures the correct sampling and thus the storing of transmitted data. If non-ideal conditions exist, the location of sampling may need to be changed (see Figure 5).
The counter size and value used to sample the incoming bits is determined by the system clock and the baud rate. The RF module allows for a baud rate between 2.4 Kbps to 19.2 Kbps. With a 32.7 KHz clock, a 2.4 Kbps can be accurately modeled with a 5-bit counter. If the user wishes to change the baud rate, the value of the sampling counter must also be changed.

Further, the counter is re-initialized when an edge is detected. As previously discussed, this allows drift to be reduced to a minimum. Therefore, it is recommended that an encoding scheme which does not allow for long lengths of consecutive bits in the stream be used.

**Figure 10: Receive Edge Detection**

---

**Hardware Description**

The following describes the hardware used to develop the CoolRunner CPLD wireless transceiver.

**RF Hardware**

The RF transmission was performed by the DR3000 module, manufactured RFM. The DR3000 is designed for short-range and low power applications with a carrier frequency of 916.5 MHz. Both On-Off Keyed (OOK) and Amplitude-Shift Keyed (ASK) modulation schemes are supported by the DR3000 module. The transceiver utilizes an Amplifier-Sequenced Hybrid (ASH) architecture and supports 2.4 to 19.2 Kbps baud rates. The baud rates can be controlled with additional hardware changes to the RF module. The CoolRunner transceiver utilizes the 2.4 Kbps transmission. The 2.4 baud rate was chosen due to the clock frequency available on the CPLD demo board.

**CPLD Hardware**

The CoolRunner XPLA3 demo board from Insight Electronics is used for the CoolRunner wireless transceiver. The demo board contains a two-digit LCD, 32.768 KHz clock, prototyping area and the Xilinx CoolRunner XPLA3 XCR3256XL TQ144 CPLD.
Hardware Setup

If using the AC adapter provided with the CoolRunner demo board, ensure that the resistor, R7 (refer to the DR300 data sheet), is removed from the DR3000. If R7 is not removed, the DR3000 will heat up and no longer function properly. Also, ensure the RF module is attached to a proper power/ground plane to minimize ground loops.

The DR3000 requires a level shifter to correctly drive the CPLD I/O pin (see Figure 11). The RF module can not drive loads stronger than 500k ohms.

Keyboard Entry Option

The following is a design implementation option for using keyboard entry with the CoolRunner wireless transceiver. CPLD design implementation is left to the user to develop.

PS/2® Protocol

The keyboard interfaces with the CPLD using the PS/2 protocol. The PS/2 protocol works on serial communication between a host and a peripheral device. The bus can be in three states: idle, inhibit, and request to send. The device can transmit a byte to the host only when the bus is idle. In order for the bus to be idle, both the CLK and DATA pins must be high (logic 1). Table 1 is the pin layout for the PS/2 cable.

Table 1: PS/2 Cable Pin Configuration

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PS/2 DATA</td>
</tr>
<tr>
<td>2</td>
<td>N/C</td>
</tr>
<tr>
<td>3</td>
<td>GROUND (0V)</td>
</tr>
<tr>
<td>4</td>
<td>POWER (+5V)</td>
</tr>
<tr>
<td>5</td>
<td>PS/2 CLK</td>
</tr>
<tr>
<td>6</td>
<td>N/C</td>
</tr>
</tbody>
</table>

The byte transmission includes a start bit (logic 0), eight data bits (LSB first), a parity bit (odd parity), and a stop bit (logic 1). The transmission occurs by having the device transmit a byte of
data by pulsing the CLK low and high 11 times, sampling the DATA line. Figure 12 depicts the waveform for one PS/2 transmission.

**Figure 12: PS/2 Transmission Waveform**

**Hardware Description**

In order to use a keyboard, a keyboard encoder must be used to manipulate data. The keyboard encoder used for this implementation is the Semtech Grencoder™ (UR5HCFJL) Zero Power™ Keyboard Encoder for Portable Systems. This keyboard encoder is the device used between the keyboard and the peripheral device. It works on a matrix (8 X 16) format with the capability to support a 128 key keyboard. The keyboard encoder has three states that it operates in: sleep, stand by, and active. These states are used to efficiently manage power consumption, making this device a good fit for use with CoolRunner. The keyboard encoder used for this design implementation can function using 3V, 3.3V, or 5V and uses the PS/2 protocol to receive data from the keyboard.

The CoolRunner transceiver is built using the CoolRunner XPLA3 Development Kit from Insight Electronics. Table 2 details the I/O pins on the demo board to the pins used on the XPLA3 256 macrocell part in the TQ144 package.

**Table 2: Prototyping Area I/O Cross Reference**

<table>
<thead>
<tr>
<th>Transceiver Signal</th>
<th>Prototyping Area I/O</th>
<th>XPLA3 Pin Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX</td>
<td>I/O 99</td>
<td>119</td>
</tr>
<tr>
<td>TX</td>
<td>I/O 106</td>
<td>138</td>
</tr>
<tr>
<td>CONTROL</td>
<td>I/O 104</td>
<td>136</td>
</tr>
</tbody>
</table>

The wireless transceiver Receive module utilization in an XPLA3 256-macrocell device is shown in Table 3. The total utilization for the Receive Module allows room for additions and/or improvements to the design.

**Table 3: CoolRunner XPLA3 256-Macrocell Utilization for Receive**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Available</th>
<th>Used</th>
<th>Utilization (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macrocells</td>
<td>256</td>
<td>168</td>
<td>65.63</td>
</tr>
<tr>
<td>P-terms</td>
<td>768</td>
<td>465</td>
<td>60.55</td>
</tr>
<tr>
<td>I/O Pins</td>
<td>116</td>
<td>20</td>
<td>17.25</td>
</tr>
</tbody>
</table>
The Transmit module utilization in an XPLA 256-macrocell device is shown in Table 4. Again, the total utilization for the transmit portion of the design allows room for addition and/or improvements to the design.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Available</th>
<th>Used</th>
<th>Utilization (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macrocells</td>
<td>256</td>
<td>118</td>
<td>46.10</td>
</tr>
<tr>
<td>P-terms</td>
<td>768</td>
<td>202</td>
<td>26.31</td>
</tr>
<tr>
<td>I/O Pins</td>
<td>116</td>
<td>20</td>
<td>17.25</td>
</tr>
</tbody>
</table>

### Design Verification

The design was verified in simulation and hardware implementation described previously in this document.

---

**VHDL Disclaimer and Download Instructions**

VHDL source code and test benches are available for this design. THE DESIGN IS PROVIDED TO YOU “AS IS”. XILINX MAKES AND YOU RECEIVE NO WARRANTIES OR CONDITIONS, EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, AND XILINX SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. XILINX DOES NOT WARRANT THE PERFORMANCE, FUNCTIONALITY, OR OPERATION OF THIS DESIGN WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE DESIGN WILL BE UNINTERRUPTED OR ERROR FREE, OR THAT DEFECTS IN THE DESIGN WILL BE CORRECTED. FURTHERMORE, XILINX DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING USE OR THE RESULTS OF THE USE OF THE DESIGN IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY OR OTHERWISE.


---

### Conclusion

This document has detailed the design of the CoolRunner CPLD logic for a wireless transceiver. The design is targeted for a 3.3V, 256 macrocell CoolRunner CPLD (XCR3256XL TQ144). This device, as well as the RF module discussed in this paper, has extremely low static and dynamic power dissipation and therefore is ideally suited for this application. The design of the CoolRunner wireless transceiver is also provided as an example of using a CoolRunner CPLD in a portable application and can be extended to many other types of portable applications.

### References

1. Zetez Semiconductors Data Sheet - ZVNL110A N-Channel Enhancement Mode Vertical D(Double Diffused) MOS FET
2. USAR GreenCoder™ Evaluation Board Data Sheet - EVK5-FJL-7603-200
4. RF Monolithics Data Sheet - DR3000 916.5 MHz Transceiver Module

### Acknowledgements

The CoolRunner wireless transceiver was development with the senior design team (May 01) of the University of New Mexico (UNM), Electrical and Computer Engineering Department. Design team included: Erin Isaacson (Xilinx), Lisa Burckel (UNM), Jeremy Dencklau (UNM), Kristina Miller (UNM), Parveen Sidu (UNM).

Additional thanks to Jim Beneke, Dennis Schlaht, and Lara Kielykta of Insight Electronics and Bruce DeVisser of Fujitsu Takamisawa who donated time and equipment to the transceiver project.
The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/25/01</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
</tbody>
</table>
Summary

This application note describes the implementation of a Smart Card Reader design with a CoolRunner™-II CPLD. Different from most of the software-based smart card reader computer systems, this CoolRunner-II CPLD implementation is a hardware solution. There is no software development needed in this design. This application note explains the low-level protocol of the Smart Card Reader and its hardware implementation.

CoolRunner-II devices are the latest CPLDs from Xilinx that offer both low power and high-speed. A VHDL code for Smart Card Reader design is available with this application note: see Source Code, page 67

Introduction

A smart card is a credit card sized plastic card with an embedded microprocessor and memory and is used for identification, access, and conducting financial transactions.

Figure 1: Smart Card Reader
Acting like a mini-computer, smart cards allow money and information to be electronically stored and transferred in a secure but portable medium. When inserted into a reader or passed over a scanner, the smart card transfers data to and from a central computer. Overall, it is a replacement for old means of retaining data and transacting business.

There are two fundamental sides of development for any smart card application: the host-side and card-side. Software programming tends to comprise most of the effort involved in smart card development. The host software runs on a computer connected to a smart card, and the card software runs on the card as a counterpart to the host software.

In this application note we use a CoolRunner-II CPLD to build a smart card reader, creating a host-side design to replace a host computer system. The function of a smart card reader is to read the card content and display the decoded information on a character LCD display. Unlike standard smart card readers, though, this CoolRunner-II reader relies on hardware design rather than software programming to perform these tasks.

The block diagram of the CoolRunner-II smart card reader is shown in Figure 2. The dashed area shows the logic that is contained in the CoolRunner-II CPLD. All the other blocks are external devices that can be obtained commercially. The CPLD logic blocks and the external devices in this diagram are briefly described in the following section.

Figure 2: Smart Card Reader Block Diagram

CoolRunner-II CPLD Modules

Main Control Logic

The Main Control Logic block provides the system flow control. It reads data from smart card control, writes to SRAM, activates LCD control and sends decoded information to the LCD control.

Smart Card Control

Smart Card Control logic communicates with the smart card through the external level shifter. It uses a predefined communication protocol and commands. The data length for each field is also hard coded in this design.
SRAM Interface

SRAM Interface logic interfaces to the external SRAM. This logic block controls the SRAM read/write addressing. Smart card data is saved in SRAM and later fetched and decoded for the LCD display.

LCD Control

LCD control logic interfaces to the LCD Display. This logic block accepts the decoded data and writes to the LCD Display.

External Devices

Level Shifter

The On semiconductor NCN6011 is a level shifter (analog device) to translate the voltages between a smart card and CoolRunner-II CPLD. This device handles all the 5V signals needed for the smart card and 1.8V signals for the CoolRunner-II CPLD. It is transparent to the smart card control logic in the CPLD.

LCD Display

The OKAYA RC1602ARS is a 16-character x 2-line, dot matrix, liquid crystal display module. It has an on-board controller and LSI drivers, which display alpha numerics, Japanese KATA KANA characters and a wide variety of other symbols in 5 x 7 dot matrix.

SRAM

A low power, 32k x 8 ISSI IS61LV256 SRAM is used in the module memory. The functionality of the SRAM does not require additional explanation. For more in-depth SRAM information refer to the ISSI SRAM data sheet.

Smart Card Acceptor

The Amphenol C702 10M008 2834 is a low cost smart card acceptor. It provides a direct sliding contact between the acceptor and the smart card. This acceptor has a “NC closed card present switch” which opens when the card is fully inserted. The switch activates the smart card reader operation.

Smart Card Standard ISO 7816

Smart Card is defined by the international standard ISO 7816. The first two parts cover smart card’s physical dimensions and locations of the chip contacts. A smart card image and its contacts are shown in Figure 3.
ISO 7816-3 & -4 govern the electronic signals, transmission protocols and inter-industry commands for interchange. In this application note we limit the discussion to its transmission protocols and some basic commands.

ISO 7816-5 to –8 cover the number system, data elements, card SQL and security commands. These parts are not used by this reference design and will not be discussed in this application note.

Operating Procedure

This section details the ISO 7816-3 inter-operation between the smart card and the host device.

- Connection and activation of the contacts
- Reset of the card
- Answer to Reset
- The T=0 communication protocol

Connection and activation of the contacts

The activation of the contacts by the host device consists of the consecutive operations:

- RST is L
- VCC is powered
- I/O in the interface device is in reception mode
- VPP is raised to idle state
- CLK is provided with a suitable, stable clock

Reset of the card

The host device initiates a reset to smart card and the card responds with an Answer to Reset within 40000 clock cycles with Reset in state H. If an Answer to Reset does not occur, the Reset returns to state L and the smart card contacts are deactivated by the host.

Answer to Reset

There are two types of transmission defined in ISO 8616-3 for answer to reset: asynchronous and synchronous transmission. In this application note we only discuss asynchronous transmission.

In asynchronous transmission, characters are transmitted on the I/O line in an asynchronous half-duplex mode. The normal bit duration used on I/O is defined as one Elementary Time Unit (etu). The initial etu is 372/ frequency second where frequency is in Hertz. All are initially operated with frequency in the range of 1 MHz to 5 MHz.

A character consists of ten consecutive bits plus a guard time as follows:

- Start bit, used for character frame synchronization
- 8 data bits of information
- Parity bit, even parity for error detection

The guard time is separation between characters. Figure 4 is a diagram for asynchronous character frame.
The Answer to Reset is at most 33 characters and consists of 5 fields,

- The initial character (TS)
- The format character (TO)
- The interface characters (TAji, TBji, TCji, TDji)
- The historical characters (T1, T2 ... TK)
- The check character (TCK)

Each of these fields is sent in order as shown in Figure 5.

---

**Figure 4: Asynchronous Character Frame**

![Asynchronous Character Frame Diagram](image-url)
The initial character TS determines the data transmission rate and also determines the sense of the logic. The format of the TS character is shown in Figure 6. This shows the two possibilities of the direct and inverse convention, where logic level one is A, Ba is MSB for inverse and logic level one is Z, and Ba is LSB for the direct convention.

Figure 5: Answer to Reset Configuration

The initial character TS determines the data transmission rate and also determines the sense of the logic. The format of the TS character is shown in Figure 6. This shows the two possibilities of the direct and inverse convention, where logic level one is A, Ba is MSB for inverse and logic level one is Z, and Ba is LSB for the direct convention.
The format character TO provides information necessary to interpret the remaining answer to reset characters. See TO in Figure 7. the most significant half byte, b8 to b5, indicates the presence of TA1 to TD1. The least significant half byte, b4 to b1, indicates the number of historical characters.

TA1 defines the basic characters of the serial transmission. F1 is the clock rate conversion factor and D1 is the bit-rate adjustment factor. F1 and D1 are compared against a table in the ISO 7816-3 standard to achieve actual values of F and D in the table to define the actual work etu.

TB1 is used to define the EPROM programming voltage and current. TC1 provides the value of N, which defines the extra guard time to be used between successive characters. The first half byte of TD1 indicates the presence of TA2 to TD2. The second half byte of TD1 indicates the protocol type T=0 to T=15.
The historical characters may be used to convey information relating to the life cycle of the card. The check character should not be sent when only the T=0 protocol is indicated in the answer to reset. In all other cases TCK is sent as the last character of the answer to reset. The value of TCK is such that the exclusive-or of all bytes from T0 to TCK included is equal to zero.

**The T=0 Communication Protocol**

The interface device always initiates the command for the T=0 protocol. Interaction between the interface device and the card results in successive commands and responses. The message flow for the T=0 protocol is shown in Figure 8.
In Figure 8, IFD is the smart card controller and ICC is the smart card. The command header consists of the following 5 bytes,

- CLA, the instruction class
- NS, the instruction code
- P1, instruction code qualifier (e.g. memory address)
- P2, additional INS code qualifier
- P3, the length of the data block

The response from the card has two status bytes, SW1 and SW2, to indicate the current card status. The normal response is SW1, SW2 = 90, 00 hex. When SW1 = 6x or 9x various error conditions are reported by the card.

Table 1 and Table 2 show some of the CPA classes and INS commands. In this design we use ISO 7816-4 instruction class 80 and some basic INS codes such as A4, Select File, B2, Read Record and C0, and Get Response to read all the information we need.
CoolRunner-II Smart Card Reader

**Table 1: CLA Instruction Set**

<table>
<thead>
<tr>
<th>CLA Type</th>
<th>Instruction Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0X</td>
<td>ISO 7816-4 instructions</td>
</tr>
<tr>
<td>10 to 7F</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>8X or 9X</td>
<td>ISO 7816-4 instructions</td>
</tr>
<tr>
<td>AX</td>
<td>Application/Vender specific instructions</td>
</tr>
<tr>
<td>B0 to CF</td>
<td>ISO 7816-4 instructions</td>
</tr>
<tr>
<td>D0 to FE</td>
<td>Application/Vender specific instructions</td>
</tr>
<tr>
<td>FF</td>
<td>Reserved for protocol type selection</td>
</tr>
</tbody>
</table>

**Table 2: INS Commands**

<table>
<thead>
<tr>
<th>INS Value</th>
<th>Command Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0E</td>
<td>Erase Binary</td>
</tr>
<tr>
<td>20</td>
<td>Verify</td>
</tr>
<tr>
<td>82</td>
<td>External Authentication</td>
</tr>
<tr>
<td>88</td>
<td>Internal Authentication</td>
</tr>
<tr>
<td>A4</td>
<td>Select File</td>
</tr>
<tr>
<td>B0</td>
<td>Read Binary</td>
</tr>
<tr>
<td>B2</td>
<td>Read Record</td>
</tr>
<tr>
<td>C0</td>
<td>Get Response</td>
</tr>
<tr>
<td>C2</td>
<td>Envelope</td>
</tr>
<tr>
<td>D0</td>
<td>Write Binary</td>
</tr>
</tbody>
</table>

**CoolRunner-II Implementation**

The CoolRunner-II smart card reader design uses the Advanced Card System ACOS1 microprocessor-based card. The information read from the card includes name, gender, status, age, and bank balance. Gender, status and age are encoded in the same data record.

The initial character TS was programmed to direct convention, and format character and interface characters are predefined. T=0 protocol is used so there is no TCK in answer to reset. There are 19 bytes, including historical characters transmitted for answer to reset. To simplify the design, we count bytes received from the smart card with the CoolRunner-II to determine the valid data to be used.

There will be no parity check for each character frame and no branch operations to handle different protocol or bytes received. In this situation only the ACOS1 card can be used for this smart card reader.

**Smart Card Control**

A smart card control block diagram and state machine are shown in Figure 9 and Figure 10. When the card is inserted into the card acceptor, the switch turns on to enable the smart card state machine. By following the ISO 7816-3 standard sequence to activate smart card contacts,
the CoolRunner-II device sets Reset to high, goes to the Wait state, and waits for a low signal from the card, the start bit of the TS for Answer to Reset.

Figure 9: Smart Card Control Block Diagram

Figure 10: Smart Card Control State Machine
There is a baud rate counter counting to 372 for every bit received or sent to synchronize the data transmission. The received bits are sampled at the 186th count, which is in the center of each bit received.

Two other counters are used, one to count bits for each character and one to count bytes received or sent. The byte number is also used to determine the end of the read data cycle or the send command cycle.

For this preprogrammed ACOS1 smart card, the state machine will set to the send command state after 19 Answer to Reset characters are received. The smart card controller is now ready to send commands and receive responses based on the T=0 protocol. There is decoder logic to check the character counts to determine when to send a command, what command to send, and how many characters will be received by the request.

The first command sent to the smart card contains 5 bytes: 80, A4, 00, 00 and 02. After one procedure byte (A4) is echoed from the smart card, the controller sends two data bytes (F0, 00) and the smart card responds with two status bytes (91, 00). As is clear from the T=0 command table, A4 selects the file address and F0, 00 is the selected file address.

The subsequent commands are to select data records and to retrieve data. All the commands and data lengths are already defined and fixed. In this design, io_rw controls the shift register to read data from card_io or shift data from data encoder to card_io. All operations are based on the byte count.

In this design, the name record is between bytes 38 and 69, gender, status and age records are bytes 92, 93 and 94. The bank balance record are bytes 126 and 127. A data_ready signal is enabled for these bytes to filter out unused data. This signal is used for the SRAM interface to write the smart card data to the SRAM.

**Main Control Logic**

A main state machine in this block controls the ordering of the functions. It also reads the SRAM data and decodes it before sending it to the LCD control logic. There is also a delay counter to separate the data sent to the LCD display, so there will be 1 to 2 seconds between each piece of information displayed on the LCD.

A block diagram is shown in Figure 11 and its flow chart is shown in Figure 12. The state machine powers up in an idle state and waits for the smart card controller to read the data and save it to SRAM. When the smartcard_done bit goes high it will go to the standby state and wait for the lcd_ready.

The card name has the same ASCII format as the LCD display so there is no need to decode the characters. All characters read from SRAM will be dumped to the LCD Display.
Figure 11: Main Control Logic Block Diagram
The next data record is decoded gender information. The data value is one for male, two for female. After gender comes status information, also decoded as one and two (one for single, two for married).

The age information is saved as binary value in the smart card. It has to be converted to ASCII digits before getting sent to the LCD controller. A binary-to-digital module is separated from the top module, which is for the ASCII coding function.

**SRAM Interface**

The SRAM interface is controlled by the main control logic. The signal sram_w is always kept high as write enable during smart card reading. The address counter will be reset when the main control state machine enters the standby state for writing data to the LCD display. The data is then retrieved in the order it was saved.

![Main Control Logic Flow Chart](chart.png)
LCD Control

LCD control logic is used to initialize and pass decoded data to the LCD display. This module uses simplified timing to control the LCD display. Every character write cycle is set to 30000 clock periods and is much higher than the few hundred microsecond LCD controller specification.

Source Code

THIRD PARTIES MAY HAVE PATENTS ON THE CODE PROVIDED. BY PROVIDING THIS CODE AS ONE POSSIBLE IMPLEMENTATION OF THIS DESIGN, XILINX IS MAKING NO REPRESENTATION THAT THE PROVIDED IMPLEMENTATION OF THIS DESIGN IS FREE FROM ANY CLAIMS OF INFRINGEMENT BY ANY THIRD PARTY. XILINX EXPRESSLY DISCLAIMS ANY WARRANTY OR CONDITIONS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, AND XILINX SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE, THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OR REPRESENTATION THAT THE IMPLEMENTATION IS FREE FROM CLAIMS OF ANY THIRD PARTY. FURTHERMORE, XILINX IS PROVIDING THIS REFERENCE DESIGN "AS IS" AS A COURTESY TO YOU.


Conclusion

This document has explained outlined some of the smart card protocol and has explained how to implement it using a CoolRunner-II, hardware-based solution. Although this Smart Card Reader design is a simplified version with no error checking or frills, it is a good reference for users needing to approach for smart card applications with minimal software development.

References

1. International Standards Organization, ISO 7816
2. Scott B. Guthery & Timothy M. Jurgensen, Smart Card Developer’s Kit
3. David B Everett, Smart Card Tutorial
4. Advanced card systems Ltd., ACS software development kit

Further Reading

Application Notes

http://direct.xilinx.com/bvdocs/appnotes/xapp371.pdf (Galois Field GF (2^m) Multiplier)
http://direct.xilinx.com/bvdocs/appnotes/xapp374.pdf (CryptoBlaze)
http://direct.xilinx.com/bvdocs/appnotes/xapp375.pdf (Timing Model)
http://direct.xilinx.com/bvdocs/appnotes/xapp376.pdf (Logic Engine)
http://direct.xilinx.com/bvdocs/appnotes/xapp379.pdf (High Speed Design)
http://direct.xilinx.com/bvdocs/appnotes/xapp380.pdf (Cross Point Switch)
http://direct.xilinx.com/bvdocs/appnotes/xapp381.pdf (Demo Board)
http://direct.xilinx.com/bvdocs/appnotes/xapp382.pdf (I/O Characteristics)
http://direct.xilinx.com/bvdocs/appnotes/xapp384.pdf (DDR SDRAM Interface)
http://direct.xilinx.com/bvdocs/appnotes/xapp388.pdf (On the Fly Reconfiguration)
http://direct.xilinx.com/bvdocs/appnotes/xapp389.pdf (Powering CoolRunner-II CPLDs)
CoolRunner-II Smart Card Reader

http://direct.xilinx.com/bvdocs/appnotes/xapp393.pdf (8051 Microcontroller Interface)
http://direct.xilinx.com/bvdocs/appnotes/xapp394.pdf (Interfacing with Mobile SDRAM)
http://direct.xilinx.com/bvdocs/appnotes/xapp395.pdf (Using DataGATE)
http://direct.xilinx.com/bvdocs/appnotes/xapp398.pdf (CompactFlash Card Interface)

CoolRunner-II Data Sheets

CoolRunner-II White Papers
http://direct.xilinx.com/bvdocs/whitepapers/wp198.pdf (Cell Phone Handsets)

Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/19/03</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
<tr>
<td>12/18/03</td>
<td>1.1</td>
<td>Minor errata.</td>
</tr>
</tbody>
</table>
Summary

This document details the VHDL implementation of an I²C controller in a Xilinx CoolRunner™-II 256-macrocell CPLD. CoolRunner-II CPLDs are the lowest power CPLDs available, making this the perfect target device for an I²C controller. To obtain the VHDL code described in this document, go to section VHDL Code Download, page 87 for instructions. This design fits both XPLA3 and CoolRunner-II CPLDs. For the CoolRunner XPLA3 CPLD version, please refer to XAPP333, CoolRunner CPLD I²C Bus Controller Implementation.

Introduction

The I²C bus is a popular serial, two-wire interface used in many systems because of its low overhead. The two-wire interface minimizes interconnections so ICs have fewer pins, and the number of traces required on printed circuit boards is reduced. Capable of 100 KHz operation, each device connected to the bus is software addressable by a unique address with a simple Master/Slave protocol.

The CoolRunner-II I²C Controller design contains an asynchronous microcontroller (μC) interface and provides I²C Master/Slave capability. It is intended to be used with a microcontroller (μC) or microprocessor (μP) as shown in Figure 1.

I²C Background

This section will describe the main protocol of the I²C bus. For more details and timing diagrams, please refer to the I²C specification.

The I²C bus consists of two wires, serial data (SDA) and serial clock (SCL), which carry information between the devices connected to the bus. The number of devices connected to the same bus is limited only by a maximum bus capacitance of 400 pF. Both the SDA and SCL lines are bidirectional lines, connected to a positive supply voltage via a pull-up resistor. When the...
bus is free, both lines are High. The output stages of devices connected to the bus must have an open-drain or open-collector in order to perform the wired-AND function.

Each device on the bus has a unique address and can operate as either a transmitter or receiver. In addition, devices can also be configured as Masters or Slaves. A Master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. Any other device that is being addressed is considered a Slave. The I²C protocol defines an arbitration procedure that insures that if more than one Master simultaneously tries to control the bus, only one is allowed to do so and the message is not corrupted. The arbitration and clock synchronization procedures defined in the I²C specification are supported by the CoolRunner-II I²C Controller.

Data transfers on the I²C bus are initiated with a START condition and are terminated with a STOP condition. Normal data on the SDA line must be stable during the High period of the clock. The High or Low state of the data line can only change when SCL is Low. The START condition is a unique case and is defined by a High-to-Low transition on the SDA line while SCL is High. Likewise, the STOP condition is a unique case and is defined by a Low-to-High transition on the SDA line while SCL is High. The definitions of data, START, and STOP insure that the START and STOP conditions will never be confused as data. This is shown in Figure 2.

![Figure 2: Data Transfer on the I²C Bus](image)

Each data packet on the I²C bus consists of eight bits of data followed by an acknowledge bit so one complete data byte transfer requires nine clock pulses. Data is transferred with the most significant bit first (MSB). The transmitter releases the SDA line during the acknowledge bit and the receiver of the data transfer must drive the SDA line low during the acknowledge bit to acknowledge receipt of the data. If a Slave-receiver does not drive the SDA line Low during the acknowledge bit, this indicates that the Slave-receiver was unable to accept the data and the Master can then generate a STOP condition to abort the transfer. If the Master-receiver does not generate an acknowledge, this indicates to the Slave-transmitter that this byte was the last byte of the transfer.

Standard communication on the bus between a Master and a Slave is composed of four parts: START, Slave address, data transfer, and STOP. The I²C protocol defines a data transfer format for both 7-bit and 10-bit addressing. The implementation of the I²C controller in the Xilinx CoolRunner-II CPLD supports the seven-bit address format. After the START condition, a Slave address is sent. This address is seven bits long followed by an eighth-bit which is the read/write bit. A “1” indicates a request for data (read) and a “0” indicates a data transmission (write). Only the Slave with the calling address that matches the address transmitted by the Master responds by sending back an acknowledge bit by pulling the SDA line Low on the ninth clock.

Once successful Slave addressing is achieved, the data transfer can proceed byte-by-byte as specified by the read/write bit. The Master can terminate the communication by generating a STOP signal to free the bus. However, the Master may generate a START signal without generating a STOP signal first. This is called a repeated START.
CoolRunner-II I2C Controller

The CoolRunner-II CPLD implementation of the I2C Controller supports the following features:

- Microcontroller interface
- Master or Slave operation
- Multi-master operation
- Software selectable acknowledge bit
- Arbitration lost interrupt with automatic mode switching from Master to Slave
- Calling address identification interrupt with automatic mode switching from Master to Slave
- START and STOP signal generation/detection
- Repeated START signal generation
- Acknowledge bit generation/detection
- Bus busy detection
- 100 KHz operation

Signal Descriptions

The I/O signals of the CoolRunner-II I2C controller are described in Table 1. Pin numbers have not been assigned to this design, this can be done to meet the system requirements of the designer.

Table 1: CoolRunner-II I2C Controller Signal Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDA</td>
<td>Bidirectional</td>
<td>I2C Serial Data.</td>
</tr>
<tr>
<td>SCL</td>
<td>Bidirectional</td>
<td>I2C Serial Clock.</td>
</tr>
<tr>
<td>ADDR_BUS[23:0]</td>
<td>Input</td>
<td>μC Address Bus.</td>
</tr>
<tr>
<td>DATA_BUS[7:0]</td>
<td>Bidirectional</td>
<td>μC Data Bus.</td>
</tr>
<tr>
<td>AS</td>
<td>Input</td>
<td>Address Strobe. Active Low μC handshake signal indicating that the address present on the address bus is valid.</td>
</tr>
<tr>
<td>DS</td>
<td>Input</td>
<td>Data Strobe. Active Low μC handshake signal indicating that the data present on the data bus is valid or that the μC is no longer driving the data bus and the I2C Controller can place data on the data bus.</td>
</tr>
<tr>
<td>R_W</td>
<td>Input</td>
<td>Read/Write. &quot;1&quot; indicates a read, &quot;0&quot; indicates a write.</td>
</tr>
<tr>
<td>DTACK</td>
<td>Output</td>
<td>Data Transfer Acknowledge. Active Low μC handshake signal indicating that the I2C Controller has placed valid data on the data bus for a read cycle or that the I2C Controller has received the data on the bus for a write cycle.</td>
</tr>
</tbody>
</table>
### Table 1: CoolRunner-II I2C Controller Signal Description

<table>
<thead>
<tr>
<th>SIG</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ</td>
<td>Output</td>
<td>Interrupt Request. Active Low.</td>
</tr>
<tr>
<td>MCF</td>
<td>Output</td>
<td>Data Transferring Bit. While one byte of data is being transferred, this bit is cleared. It is set by the falling edge of the ninth clock of a byte transfer. This bit is used to signal the completion of a byte transfer to the μC.</td>
</tr>
<tr>
<td>CLK</td>
<td>Input</td>
<td>Clock. This clock is input from the system. The constants used in generating a 100 KHz SCL signal assumes the frequency to be 1.832 MHz. Different clock frequencies can be used, but the constants in the VHDL source code must be recalculated.</td>
</tr>
</tbody>
</table>

### Block Diagram

The block diagram of the CoolRunner-II I2C Controller, shown in Figure 3 was broken into two major blocks, the μC interface and the I2C interface.

![CoolRunner-II I2C Controller Block Diagram](image_url)

**Figure 3: CoolRunner-II I2C Controller**
Microcontroller Logic

The μC interface for the I²C controller design supports an asynchronous byte-wide bus protocol. This protocol is the method in which the μC reads and writes the registers in the design and is shown in Figure 4.

Address Decode/Bus Interface Logic

The μC bus protocol is implemented in the CoolRunner-II I²C Controller in the state machine shown in Figure 5.

Figure 4: μC Read/Write Protocol

Figure 5: μC Bus Interface State Machine
In the first cycle, the µC places the address on the address bus, sets the read/write line to the correct state, and asserts address strobe (AS) and data strobe (DS). Address strobe indicates that the address present on the address bus is valid. If this is a write cycle, the µC also places the data on the data bus and DS indicates that valid data is present on the data bus. If this is a read cycle, the µC 3-states the data bus and DS indicates that the CoolRunner-II I²C Controller can place data on the data bus.

Upon the assertion of AS, the CoolRunner-II I²C Controller transitions to the ADDR state to decode the address and determine if it is the device being addressed. The enables for the internal registers are set in this state. If the CoolRunner-II I²C Controller is being addressed and DS is asserted, the CoolRunner-II I²C controller progresses to the DATA_TRS state. If this is a read cycle, the requested data is placed on the bus and if this is a write cycle, the data from the data bus is latched in the addressed register. The CoolRunner-II I²C Controller automatically progresses to the ASSERT_DTACK state and asserts DTACK indicating that the data requested is ready if a read cycle or that the data has been received if a write cycle.

Upon the assertion of DTACK, the µC either removes data from the bus if this is a write cycle, or latches the data present on the bus if this is a read cycle. The read/write line is set to read and AS and DS are negated to indicate that the data transfer is complete. The negation of AS and DS causes the CoolRunner-II I²C Controller to negate DTACK and transition to the IDLE state.

### CoolRunner-II I²C Controller Registers

The base address used for address decoding is set in the VHDL code via the constant BASE_ADDRESS. The base address is the upper 16 bits of the address bus. The lower address bits determine which register is being accessed.

The registers supported in the CoolRunner-II I²C Controller are described in the Table 2. The µC interface logic of the CoolRunner-II I²C Controller handles the reading and writing of these registers by the µC and supplies and/or retrieves these bits to/from the I²C interface logic.

#### Table 2: I²C Controller Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBASE + $8Dh</td>
<td>MADR</td>
<td>I²C Address Register</td>
</tr>
<tr>
<td>MBASE + $91h</td>
<td>MBCR</td>
<td>I²C Control Register</td>
</tr>
<tr>
<td>MBASE + $93h</td>
<td>MBSR</td>
<td>I²C Status Register</td>
</tr>
<tr>
<td>MBASE + $95h</td>
<td>MBDR</td>
<td>I²C Data I/O Register</td>
</tr>
</tbody>
</table>

#### Address Register (MADR)

This field contains the specific Slave address to be used by the I²C Controller. This register is read/write. (Table 3).

#### Table 3: Address Register Bits

<table>
<thead>
<tr>
<th>Bit Location</th>
<th>Name</th>
<th>µC Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-1</td>
<td>Slave Address</td>
<td>Read/Write</td>
<td>Address used by the I²C controller when in Slave mode.</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Unused</td>
</tr>
</tbody>
</table>
Control Register (MBCR)
This register contains the bits to configure the I²C controller. (Table 4).

**Table 4: Control Register Bits**

<table>
<thead>
<tr>
<th>Bit Location</th>
<th>Name</th>
<th>μC Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>MEN</td>
<td>Read/Write</td>
<td><strong>I²C Controller Enable.</strong> This bit must be set before any other MBCR bits have any effect. “1” enables the I²C controller. “0” resets and disables the I²C controller.</td>
</tr>
<tr>
<td>6</td>
<td>MIEN</td>
<td>Read/Write</td>
<td><strong>Interrupt Enable.</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“1” enables interrupts. An interrupt occurs if MIF bit in the status register is also set. “0” disable interrupts but does not clear any currently pending interrupts.</td>
</tr>
<tr>
<td>5</td>
<td>MSTA</td>
<td>Read/Write</td>
<td><strong>Master/Slave Mode Select.</strong> When the μC changes this bit from &quot;0&quot; to &quot;1&quot;, the I²C controller generates a START condition in Master mode. When this bit is cleared, a STOP condition is generated and the I²C controller switches to Slave mode. If this bit is cleared, however, because arbitration for the bus has been lost, a STOP condition is not generated.</td>
</tr>
<tr>
<td>4</td>
<td>MTX</td>
<td>Read/Write</td>
<td><strong>Transmit/Receive Mode Select.</strong> This bit selects the direction of Master/Slave transfers. “1” selects an I²C Master transmit. “0” selects an I²C Master receive.</td>
</tr>
<tr>
<td>3</td>
<td>TXAK</td>
<td>Read/Write</td>
<td><strong>Transmit Acknowledge Enable.</strong> This bit specifies the value driven onto the SDA line during acknowledge cycles for both Master and Slave receivers. “1” - ACK bit = “1” - no acknowledge. “0” - ACK bit = “0” - acknowledge. Since Master receivers indicate the end of data reception by not acknowledging the last byte of the transfer, this bit is the means for the μC to end a Master receiver transfer.</td>
</tr>
<tr>
<td>2</td>
<td>RSTA</td>
<td>Read/Write</td>
<td><strong>Repeated Start.</strong> Writing a &quot;1&quot; to this bit generates a repeated START condition on the bus if the I²C controller is the current bus Master. This bit is always read as &quot;0&quot;. Attempting a repeated START at the wrong time if the bus is owned by another Master results in a loss of arbitration.</td>
</tr>
<tr>
<td>1-0</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Status Register (MBSR)

This register contains the status of the I²C controller. This status register is read-only with the exception of the MIF and MAL bits, which are software clearable. All bits are cleared upon reset except the MCF and RXAK bits. (Table 5).

Table 5: Status Register Bits

<table>
<thead>
<tr>
<th>Bit Location</th>
<th>Name</th>
<th>μC Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>MCF</td>
<td>Read</td>
<td>Data Transferring Bit. While one byte of data is being transferred, this bit is cleared. It is set by the rising edge of SCL during the acknowledge cycle of the transfer and is only High for this SCL clock period.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>1</em> transfer is complete</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>0</em> transfer in progress</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note that in the CoolRunner-II I²C controller, this bit is also an output pin so that a register read cycle is not required to determine that a transfer is complete.</td>
</tr>
<tr>
<td>6</td>
<td>MAAS</td>
<td>Read</td>
<td>Addressed as Slave Bit. When the address on the I²C bus matches the Slave address in the MADR register, the I²C controller is being addressed as a Slave and switches to Slave mode.</td>
</tr>
<tr>
<td>5</td>
<td>MBB</td>
<td>Read</td>
<td>Bus Busy Bit. This bit indicates the status of the I²C bus. This bit is set when a START condition is detected and cleared when a STOP condition is detected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>1</em> indicates the bus is busy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>0</em> indicates the bus is idle</td>
</tr>
<tr>
<td>4</td>
<td>MAL</td>
<td>Read</td>
<td>Arbitration Lost Bit. This bit is set by hardware when arbitration for the I²C bus is lost. This bit must be cleared by the μC software writing a &quot;0&quot; to this bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software Clearable</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Register (MBDR)

This register contains data to/from the I²C bus. Physically, this register is implemented by two byte-wide registers at the same address, one for the I²C transmit data and one for the I²C received data. This eliminates any possible contention between the μC and the CoolRunner-II I²C Controller. Since these registers are at the same address they appear as the same register to the μC and will continue to be described as such. In transmit mode, data written into this register is output on the I²C bus, in receive mode, this register contains the data received from the I²C bus. Note that in receive mode, it is assumed that the μC will be able to read this register during the next I²C transfer. The received I²C data is placed in this register after each complete transfer, the I²C interface logic does not wait for an indication from the μC that this register has been read before proceeding with the next transfer. (Table 6)

Table 6: I²C Data Register Bit

<table>
<thead>
<tr>
<th>Bit Location</th>
<th>Name</th>
<th>μC Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-0</td>
<td>D7:D0</td>
<td>Read/Write</td>
<td>I²C Data</td>
</tr>
</tbody>
</table>

I²C Interface Logic

The I²C bus interface logic consists of several different processes as seen in Figure 3. Control bits from the μC interface registers determine the behavior of these processes.

Arbitration

Arbitration of the I²C bus is lost in the following circumstances:

- The SDA signal is sampled as a "0" when the Master outputs a "1" during an address or data transmit cycle
- The SDA signal is sampled as a "0" when the Master outputs a "1" during the acknowledge bit of a data receive cycle
- A start cycle is attempted when the bus is busy
• A repeated start cycle is requested in Slave mode
• A STOP condition is detected when the Master did not request it

If the CoolRunner-II I²C Controller is in Master mode, the outgoing SDA signal is compared with the incoming SDA signal to determine if control of the bus has been lost. The SDA signal is checked only when SCL is High during all cycles of the data transfer except for acknowledge cycles to insure that START and STOP conditions are not generated at the wrong time. If the outgoing SDA signal and the incoming SDA signals differ, then arbitration is lost and the MAL bit is set. At this point, the CoolRunner-II I²C Controller switches to Slave mode and resets the MSTA bit.

The CoolRunner-II I²C design will not generate a START condition while the bus is busy, however, the MAL bit will be set if the μC requests a START or repeated START while the bus is busy. The MAL bit is also set if a STOP condition is detected when this Master did not generate it.

If arbitration is lost during a byte transfer, SCL continues to be generated until the byte transfer is complete.

**START/STOP Detection**

This process monitors the SDA and SCL signals on the I²C bus for START and STOP conditions. When a START condition is detected, the Bus Busy bit is set. This bit stays set until a STOP condition is detected. The signals, DETECT_START and DETECT_STOP are generated by this process for use by other processes in the logic. Note that this logic detects the START and STOP conditions even when the CoolRunner-II I²C Controller is the generator of these conditions.

**Generation of SCL, SDA, START and STOP Conditions**

This process generates the SCL and SDA signals output on the I²C bus when in Master mode. The clock frequency of the SCL signal is ~100 KHz and is determined by dividing down the input clock. The number of input clock cycles required for generation of a 100 KHz SCL signal is set by the constant CNT_100 KHZ and is currently calculated for a system clock of 1.832 MHz. This constant can easily be modified by a designer based on the clock available in the target system. Likewise, the constants START_HOLD and DATA_HOLD contain the number of system clock cycles required to meet the I²C requirements on hold time for the SDA lines after generating a START condition and after outputting data.
The state machine that generates SCL and SDA when in Master mode is shown in Figure 6. Note that SCL and SDA are held at the default levels if the bus is busy. This state machine generates the controls for the system clock counter.

The internal SDA signal output from this design is either the SDA signal generated by this state machine for START and STOP conditions or the data from the MBDR register when the CoolRunner-II I²C Controller is in transmit mode. Note that both SCL and SDA are open-collector outputs, therefore, they are only driven to a "0". When a "1" is to be output on these signals, the CoolRunner-II I²C Controller 3-states their output buffers. The logic in the design will set internal SDA and SCL signals to "1" or "0". These internal signals actually control the output enable of the 3-state buffer for these outputs.

In the IDLE state, SCL and SDA are 3-stated, allowing any Master to control the bus. Once a request has entered to generate a start condition, the CoolRunner-II I²C Controller is in Master mode, and the bus is not busy, the state machine transitions to the START state.

The START state holds SCL High, but drives SDA Low to generate a START condition. The system clock counter is started and the state machine stays in this state until the required hold time is met. At this point, the next state is SCL_LOW_EDGE.

The SCL_LOW_EDGE state simply creates a falling edge on SCL and resets the system clock counter. On the next clock edge, the state machine moves to state SCL_LOW. In this state, the SCL line is held Low and the system clock counter begins counting. If the REP_START signal

**Figure 6: SCL, SDA, START, and STOP Generation State Machine**

The internal SDA signal output from this design is either the SDA signal generated by this state machine for START and STOP conditions or the data from the MBDR register when the CoolRunner-II I²C Controller is in transmit mode. Note that both SCL and SDA are open-collector outputs, therefore, they are only driven to a "0". When a "1" is to be output on these signals, the CoolRunner-II I²C Controller 3-states their output buffers. The logic in the design will set internal SDA and SCL signals to "1" or "0". These internal signals actually control the output enable of the 3-state buffer for these outputs.

In the IDLE state, SCL and SDA are 3-stated, allowing any Master to control the bus. Once a request has entered to generate a start condition, the CoolRunner-II I²C Controller is in Master mode, and the bus is not busy, the state machine transitions to the START state.

The START state holds SCL High, but drives SDA Low to generate a START condition. The system clock counter is started and the state machine stays in this state until the required hold time is met. At this point, the next state is SCL_LOW_EDGE.

The SCL_LOW_EDGE state simply creates a falling edge on SCL and resets the system clock counter. On the next clock edge, the state machine moves to state SCL_LOW. In this state, the SCL line is held Low and the system clock counter begins counting. If the REP_START signal
is asserted then the SDA signal will be set High, if the GEN_STOP signal is asserted, SDA will be set Low.

When the SCL low time has been reached, the state machine will transition to the IDLE state if arbitration has been lost and the byte transfer is complete to insure that SCL continues until the end of the transfer. Otherwise the next state is the SCL_HI_EDGE state.

The SCL_HI_EDGE state generates a rising edge on SCL by setting SCL to "1". Note, however, that the state machine will not transition to the SCL_HI state until the sampled SCL signal is also High to obey the clock synchronization protocol of the I²C specification. Clock synchronization is performed using the wired-AND connection of the SCL line. The SCL line will be held Low by the device with the longest low period. Devices with shorter low periods enter a high wait state until all devices have released the SCL line and it goes High. Therefore the SCL_HI_EDGE state operates as the high wait state as the SCL clock is synchronized.

The SCL_HI state then starts the system clock counter to count the high time for the SCL signal. If a repeated START or a STOP condition has been requested, the state machine will transition to the appropriate state after half of the SCL high time so that the SDA line can transition as required. If neither of these conditions has been requested, then the state machine transitions to the SCL_LOW_EDGE state when the SCL high time has been achieved.

The STOP_WAIT state is used to insure that the hold time requirement after a STOP condition is met.

**I²C Interface Main State Machine**

The main state machine for the I²C Interface logic is shown in Figure 7. This state machine is the same for both Slave and Master modes. In each state, the mode is checked to determine the proper output values and next state conditions. This allows for immediate switching from
Master to Slave mode if arbitration is lost or if the CoolRunner-II I²C Controller is addressed as a Slave.

This state machine utilizes and controls a counter that counts the I²C bits that have been received. This count is stored in the signal BIT_CNT. This state machine also controls two shift registers, one that stores the I²C header that has been received and another that stores the I²C data that has been received or is to be transmitted.

Note:
This state machine and the associated counters and shift registers are clocked on the falling edge of the incoming SCL clock. If the load is heavy on the SCL line, the rise time of the SCL signal may be very slow which can cause susceptibility to noise for some systems. This can be particularly dangerous on a clock signal. The designer is strongly encouraged to investigate the signal integrity of the SCL line and if necessary, use external buffers for the SCL signal.

When a START signal has been detected, the state machine transitions from the IDLE state to the HEADER state. The START signal detection circuit monitors the incoming SDA and SCL lines to detect the START condition. The START condition can be generated by the CoolRunner-II I²C controller or another Master—either source will transition the state machine to the HEADER state.

The HEADER state is the state where the I²C header is transmitted on the I²C bus from the MBDR register if in Master mode. In this state, the incoming I²C data is captured in the I²C Header shift register. In Master mode, the I²C Header shift register will contain the data that was just transmitted by this design. When all eight bits of the I²C header have been shifted in, the state machine transitions to the ACK_HEADER state.

Figure 7: I²C Interface Main State Machine

This state machine utilizes and controls a counter that counts the I²C bits that have been received. This count is stored in the signal BIT_CNT. This state machine also controls two shift registers, one that stores the I²C header that has been received and another that stores the I²C data that has been received or is to be transmitted.

Note:
This state machine and the associated counters and shift registers are clocked on the falling edge of the incoming SCL clock. If the load is heavy on the SCL line, the rise time of the SCL signal may be very slow which can cause susceptibility to noise for some systems. This can be particularly dangerous on a clock signal. The designer is strongly encouraged to investigate the signal integrity of the SCL line and if necessary, use external buffers for the SCL signal.
In the ACK_HEADER state, the CoolRunner-II I2C design samples the SDA line if in Master mode to determine whether the addressed I2C Slave acknowledged the header. If the addressed Slave does not acknowledge the header, the state machine will transition to the STOP state which signals the SCL/START/STOP generator to generate a STOP. If the addressed Slave has acknowledged the address, then the LSB of the I2C header is used to determine if this is a transmit or receive operation and the state machine transitions to the appropriate state to either receive data, RCV_DATA, or to transmit data, XMIT_DATA.

The I2C Header shift register is constantly compared with the I2C address set in the MADR register. If these values match in the ACK_HEADER state, the CoolRunner-II I2C Controller has been addressed as a Slave and the mode immediately switches to Slave mode. The MAAS bit is then set in the MBSR status register. The SDA line will be driven as set in the TXAK register to acknowledge the header to the current I2C bus Master. Again, the LSB of the I2C header is used to determine the direction of the data transfer and the appropriate state is chosen.

The RCV_DATA state shifts the incoming I2C data into the I2C shift register for transfer to the µC. When the whole data byte has been received, the state machine transitions to the ACK_DATA state and the value of the TXAK register is output on the SDA line to acknowledge the data transfer. Note that in Master mode, the indication that the Slave has transmitted the required number of data bytes is to not acknowledge the last byte of data. The µC must negate the TXAK bit to prohibit the ACK of the last data byte. The state machine exits this pair of states when a STOP condition has been detected, otherwise, the transition between these two states continues. In Master mode, the µC requests a STOP condition by negating the MSTA bit.

The XMIT_DATA state shifts the data from the I2C data register to the SDA line. When the entire byte has been output, the state machine transitions to the WAIT_ACK state. If an acknowledge is received, the state machine goes back to the XMIT_DATA to transmit the next byte of data. This pattern continues until either a STOP condition is detected, or an acknowledge is not received for a data byte.

Note that the data transfer states of this state machine assume that the µC can keep up with the rate at which data is received or transmitted. If interrupts are enabled, an interrupt is generated at the completion of each byte transfer. The MCF bit is set as well providing the same indication. Data is transferred to/from the I2C data register to/from the µC data register during the acknowledge cycle of the data transfer. The state machine does not wait for an indication that the µC has read the received data or that new data has been written for transmission. The designer should be aware of the effective data rate of the µC to insure that this is not an issue.

The STOP state signals the SCL/START/STOP generator to generate a STOP condition if the CoolRunner-II I2C design is in Master mode. The next state is always the IDLE state and the I2C activity is completed.
Operational Flow Diagrams

The flow of the interface between the μC and the CoolRunner-II I²C Controller is detailed in the following flow charts. These flow charts are meant to be a guide for utilizing the CoolRunner-II I²C Controller in a μC system.

Initialization

Before the CoolRunner-II I²C Controller can be utilized, certain bits and registers must be initialized as shown in Figure 8.

```
BEGIN

Enable I²C Interface Logic by Setting MEN

Define I²C Slave Address to Respond to in MADR

Modify MBCR to Enable Interrupts

END
```

*Figure 8: CoolRunner-II I²C Controller Initialization Flow Chart*

Master Transmit/Receive

The flow charts for transmitting data and receiving data while I²C bus Master are shown in Figure 9 and Figure 10. The major difference between transmitting and receiving is the additional step in the Master Receive flow chart of turning off the acknowledge bit on the second to last data word.
Figure 9: Master Transmit Flow Chart
Figure 10: Master Receive Flow Chart
Slave Flow Chart

The flow chart for receiving or transmitting data in Slave mode is shown in Figure 11. If in receive mode, the first read from the MBDR register is a dummy read because data has not yet been received. Since the CoolRunner-II I²C Controller is in Slave mode, the only way to know that the transaction is complete is to check that the bus is busy and that the Addressed as Slave bit is still set.

![Slave/Transmitter Flow Chart](image)

Figure 11: Slave/Transmitter Flow Chart

The design of the CoolRunner-II I²C Controller was implemented in VHDL and targeted to a 256 macrocell CoolRunner-II CPLD in a 144-pin TQFP package (XC2C256-5TQ144) using Xilinx Project Navigator. (Xilinx Project Navigator software is available free-of-charge from the Xilinx website: [www.xilinx.com/products/software/webpowered.htm](http://www.xilinx.com/products/software/webpowered.htm).

**Note:**
Since the system clock frequency was 1.832 MHz, the speed of the design was not critical and any speed grade part could have been used.

**Note:**
The I²C SCL line is used as a clock input into the CoolRunner-II I²C Controller. If there are many loads on the I²C bus, the rise time of the SCL line can be quite slow. The CoolRunner-II CPLD for this design requires a rise time no greater than 20 ns, therefore, the designer is strongly encouraged to examine the characteristics of the SCL signal in the I²C system. If the rise time of the I²C signals are greater than 20 ns, the inputs can be configured as Schmitt Triggers providing for input hysteresis and noise immunity. Schmitt Trigger inputs will prevent adverse effects of slow rising/falling input signals.

The I²C design utilization in a CoolRunner-II 256-macrocell device is shown in Table 7. This utilization was achieved using certain fitter parameters, actual results may vary. As shown,
there is plenty of room remaining in the device for the implementation of other logic in the system.

**Table 7: CoolRunner-II CPLD 256-Macrocell Utilization**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Available</th>
<th>Used</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macrocells</td>
<td>256</td>
<td>127</td>
<td>50%</td>
</tr>
<tr>
<td>P-terms</td>
<td>896</td>
<td>352</td>
<td>39%</td>
</tr>
<tr>
<td>Registers Used</td>
<td>256</td>
<td>110</td>
<td>43%</td>
</tr>
<tr>
<td>I/O Pins</td>
<td>118</td>
<td>42</td>
<td>36%</td>
</tr>
<tr>
<td>Function Block</td>
<td>640</td>
<td>305</td>
<td>48%</td>
</tr>
</tbody>
</table>

**Design Verification**

The Xilinx Project Navigator software package outputs a timing VHDL model of the fitted design. This post-fit VHDL was simulated with the original VHDL test benches to insure design functionality. Also, the CoolRunner-II I²C Controller design was simulated with an independently generated VHDL model of an I²C Slave design to verify that the interface specifications were implemented correctly. Please note that all verification of this design has been done through simulations.

---

**VHDL Code Download**

VHDL source code and test benches are available for this design. THE DESIGN IS PROVIDED TO YOU "AS IS". XILINX MAKES AND YOU RECEIVE NO WARRANTIES OR CONDITIONS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, AND XILINX SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. This design has not been verified on hardware (as opposed to simulations), and it should be used only as an example design, not as a fully functional core. XILINX does not warrant the performance, functionality, or operation of this Design will meet your requirements, or that the operation of the Design will be uninterrupted or error free, or that defects in the Design will be corrected. Furthermore, XILINX does not warrant or make any representations regarding use or the results of the use of the Design in terms of correctness, accuracy, reliability or otherwise.


**Disclaimer**

I²C is a registered trademark of Philips Electronics N.V. Xilinx provides this reference design as one possible implementation of this standard and claims no rights to this interface. To use this reference design, you must contact Philips to obtain any necessary rights associated with this interface.

**Conclusion**

This document has detailed the design of an I²C Controller design for a CoolRunner-II CPLD. Though the design has been extensively verified in simulations, Xilinx assumes no responsibility for the accuracy or the functionality of this design.
Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/24/02</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
<tr>
<td>12/30/03</td>
<td>1.1</td>
<td>Change to control register address.</td>
</tr>
</tbody>
</table>