XILINX®

XAPP147 (v1.0) January 25, 2001

Low Power Handspring Springboard Module Design with CoolRunner CPLDs

Summary

This application note presents development aids to help designers successfully and easily create Handspring™ Springboard™ Module designs. It includes a general discussion of the overall process, a summary of available software tools and an introduction to Xilinx CoolRunner CPLDs as used on the Insight Electronics Springboard Development Card. The appendices include additional details that developers will find helpful for both code creation and hardware development. Examples of hardware code (VHDL or Verilog) as well as "C" code are provided to augment the development of Handspring Springboard cards.

Introduction

Portable Digital Assistants (PDA) represent a large portion of the handheld electronics market. The ability to enhance the functionality of PDAs has grown in popularity in recent years. Handspring has led the pack in creating a PDA – the Visor – that specifically targets add-on specialization cards called Springboards. Handspring uses the industry standard Palm OS as its software platform and has created a “plug and play” format for developers to quickly create new applications that seamlessly fit into its software and hardware environment.

Application development is typically slowed by an apparent restriction to ASIC or low voltage CMOS logic technology. ASICs typically require more than one revision to correctly create an IC solution and require large initial capital commitments. The ASIC choice can extend product development beyond target market windows and consume precious capital. Low voltage CMOS logic is restrictive from a density viewpoint. Frequently, either choice incurs printed circuit board modifications that are difficult due to the small physical form factor of the Springboard card profile—enter Programmable Logic.

Xilinx has developed a low voltage Programmable Logic Device (PLD) family that is low cost, easy to use, and ideal for rapid product development — the XPLA3™ CoolRunner™ family. XPLA3 devices range from 32 to 512 macrocells and operate over a comfortable battery range from 2.7 to 3.6 volts. These devices consume only microamps from the Visor battery, so many Springboards can use the native PDA batteries. The package offering includes today’s 1.0mm and 0.8mm fine pitch BGA packages.

To make designs even easier to prototype, a development board has been created by Insight Electronics that combines the most commonly used add on functions onto a small PCB with remaining space to add ASSP ICs, transducers or additional logic functions (see Figure 1). This board is part of a design kit available directly from Insight Electronics over their website at:

www.insight-electronics.com

Figure 1 illustrates a simplified block diagram of the Insight board, and Figure 2 shows a photo.

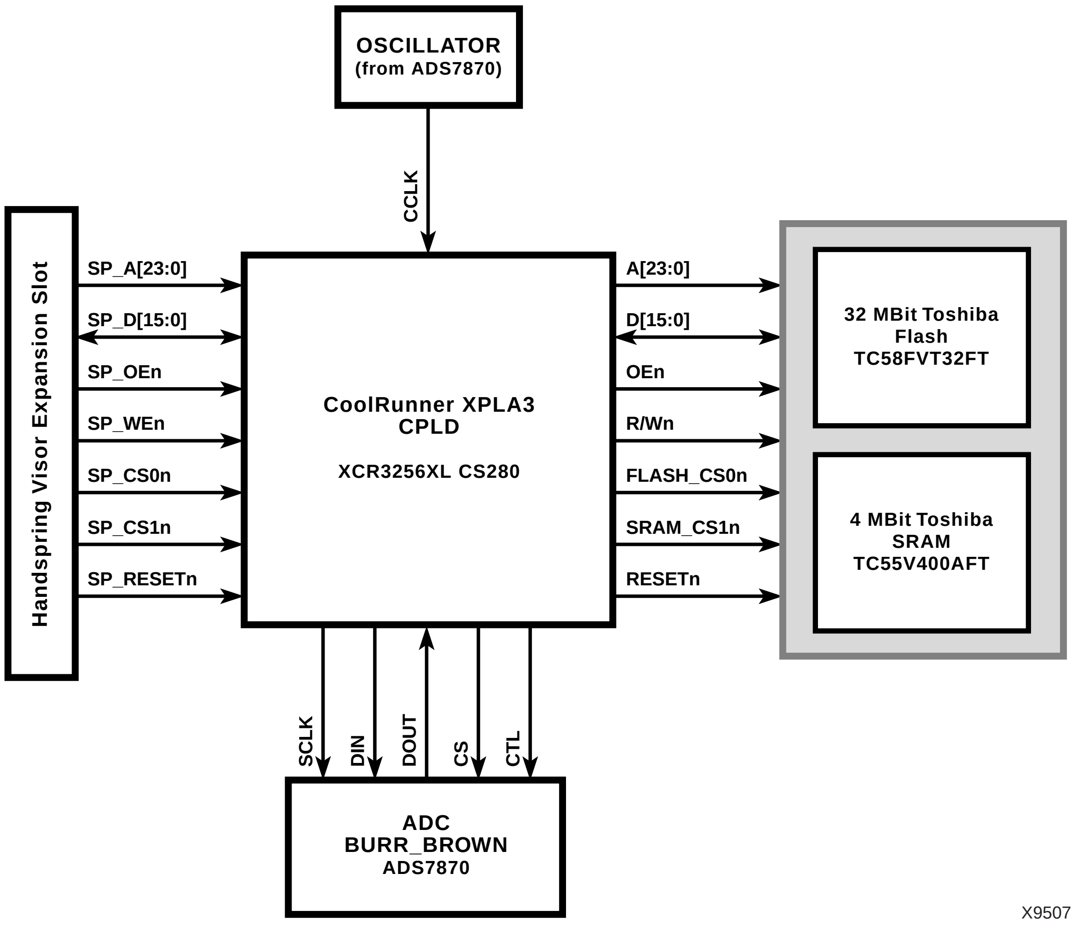


Figure 1: **Insight Electronics Springboard Block Diagram**

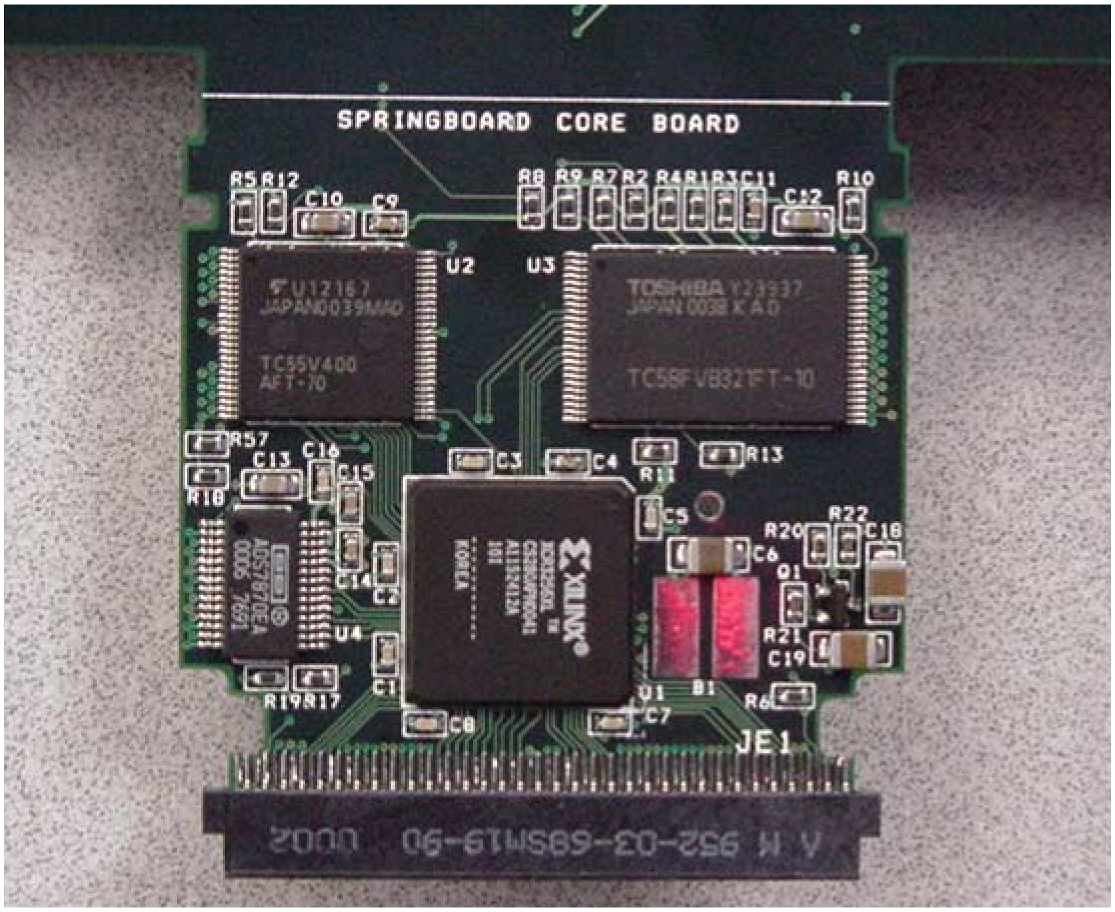


Figure 2: **Insight Electronics Springboard Circuit Card (partial view)**

Basic Development Flow

Here, we will simply outline some of the steps involved in developing an application. It is assumed that the Insight board will be used and will interface directly to the Visor bus.

First, a partition of the design must be done to identify which tasks will be accomplished in software and which in hardware. Fast transactions — faster than the Visor bus — are done in hardware. Slower operations are assigned to software. To reduce expense and increase flexibility, software operations are preferred.

Software Steps

1. Learn the development software.
2. Create the source code with “C” or Assembler.
3. Create the graphic Icon.
4. Create the “.prc” resource file.
5. Hot synch the .prc file and execute it without hardware.
6. As bugs are discovered, make edits and repeat steps 2-5.
7. Go to Hardware steps.

Hardware Steps

1. Create hardware design using VHDL, Verilog, ABEL or schematic
2. Compile it using Xilinx software — WebPACK is a free toolfor download.

3. Compile until an error free design occurs.
4. Simulate, if you wish.
5. If steps 1-4 give an acceptable initial design, attach a JTAG Download cable to the XPLA3 part and download the bitstream.
6. Go to Integration.

Integration Steps

1. At this point, the software and hardware compile.
2. Plug the programmed part into the Visor.
3. Launch the target application and exercise it.
4. Else, see the Debug section in Appendix E.

Both hardware and software errors will arise and edits can occur as needed. Luckily, software and programmable logic permits in-circuit modification. Nonetheless, it is advisable to develop diagnostic codes, test patterns, and other circuit checks..

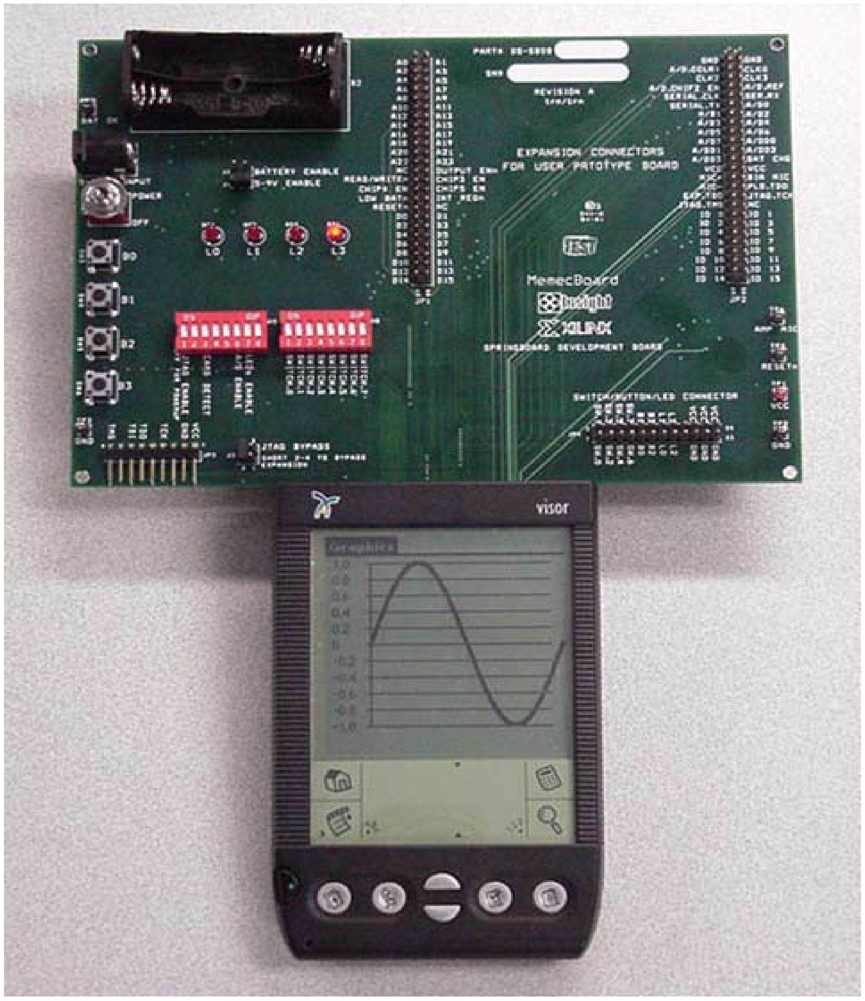


Figure 3: **Insight Circuit Card Inserted into Visor**

Springboard Basics

Handspring (www.handspring.com) has created a set of development tools to simplify the development of Springboards. Their website contains a developers' page that links to the software tools, various manufacturers of support products – like debuggers and generic packages – and technical notes on building and coding applications. Key documents designers can obtain there include the electrical specifications, physical dimensions and debug tools for the software. The page also contains links to the Palm Developers Zone, which has become the standard for Palm OS developers. Browsing the Handspring developers' site to discover the wealth of information available there is strongly recommended.

CoolRunner Basics

XPLA3 CPLDs deliver the best low power operation of any programmable logic device. Without having to do any heroic tricks, designers get low power benefits not present in other programmable logic devices. The only critical practices are those typical of CMOS logic – drive inputs to standard logic levels, don't let signals float, etc. There is no need to interrupt the VCC on the chip to gain the best low power operation. XPLA3 CPLDs can directly attach to the Visor backplane and be powered from the available VCC. The Insight development board permits jumper choice on using the Visor VCC or external batteries.

Many add-on applications are restricted by the classic "Von Neumann" bottleneck. They attempt to do everything with the microprocessor and are limited by the instruction cycle and the bus speed. CoolRunner CPLDs permit very high speed operation – in excess of 100 MHz if needed – and can capture high-speed data into an external SRAM for later analysis by the slower processor. Besides being capable of fast sequential switching, XPLA3 pin to pin delays are as fast as 5 nsec for high-speed combinational designs. There are multiple technical notes explaining the architecture, power improvement techniques, and a wide range of bus and memory interface techniques at: www.xilinx.com. Search for CoolRunner Application Notes.

Insight Board Basics

Figure 4 shows a block diagram of the standard Insight board connection.

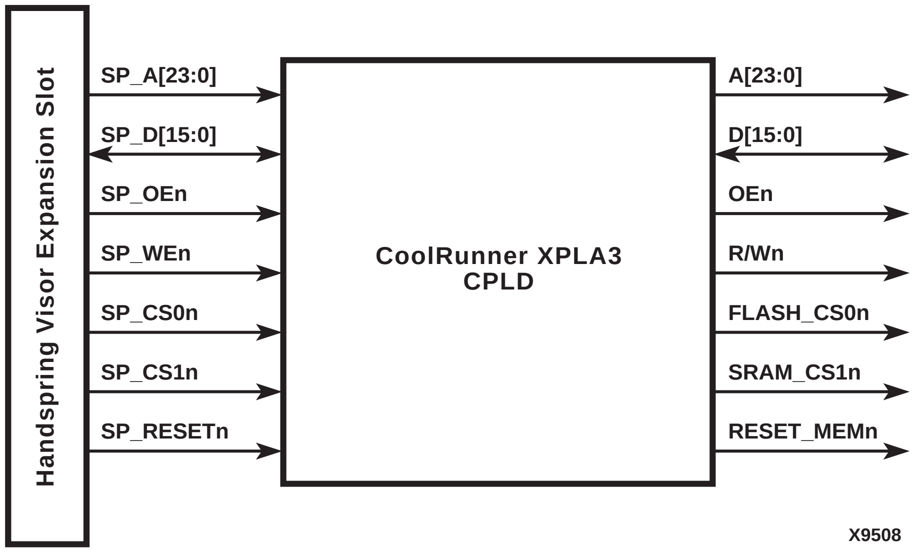


Figure 4: **Insight Board XPLA3 Signals**

All signals from the Visor are buffered through the CoolRunner CPLD.

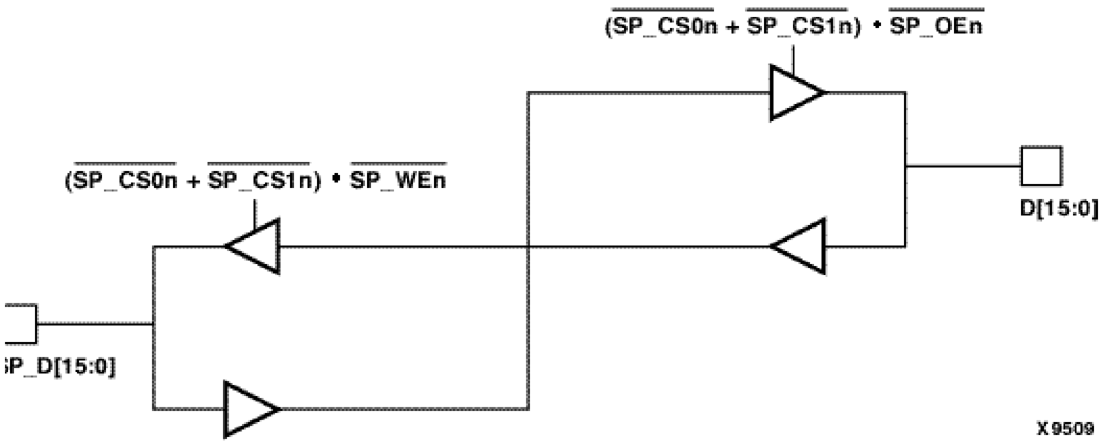
Note that in every case, signals originating from the springboard expansion slot are prefixed with "SP" (i.e. SP_A, SP_D, SP_OEn, SP_WEn, etc.). These signals are buffered through the CoolRunner CPLD. The buffered signals have similar names but have no "SP" prefix. These buffered signals are then routed to the external components, namely Flash, SRAM, and A/D. Table 1 lists the original signal names and their corresponding buffered signal names.

Table 1: **Signal Naming**

Original Signal Name	Buffered Signal Name
SP_A[23:0]	A[23:0]
SP_D[15:0]	D[15:0]
SP_OEn	OEn
SP_WEn	WEn
SP_CS0n	FLASH_CS0n
SP_CS1n	SRAM_CS1n
SP_RESETn	RESET_MEMn

Figure 5 shows the simple functionality that interfaces the chips through the CoolRunner CPLD. Other applications would take advantage of the CPLD resources and build address counters, serial data formatters and other intelligent control functions to accomplish their tasks. Figure 5 takes the simple approach of simply tying things together and relies on software to manage the external memories and the A/D converter.

SP_D


$(\overline{SP\_CS0n} + \overline{SP\_CS1n}) \cdot \overline{SP\_OEn}$
$(\overline{SP\_CS0n} + \overline{SP\_CS1n}) \cdot \overline{SP\_WEn}$
D[15:0]
SP_D[15:0]
X9509

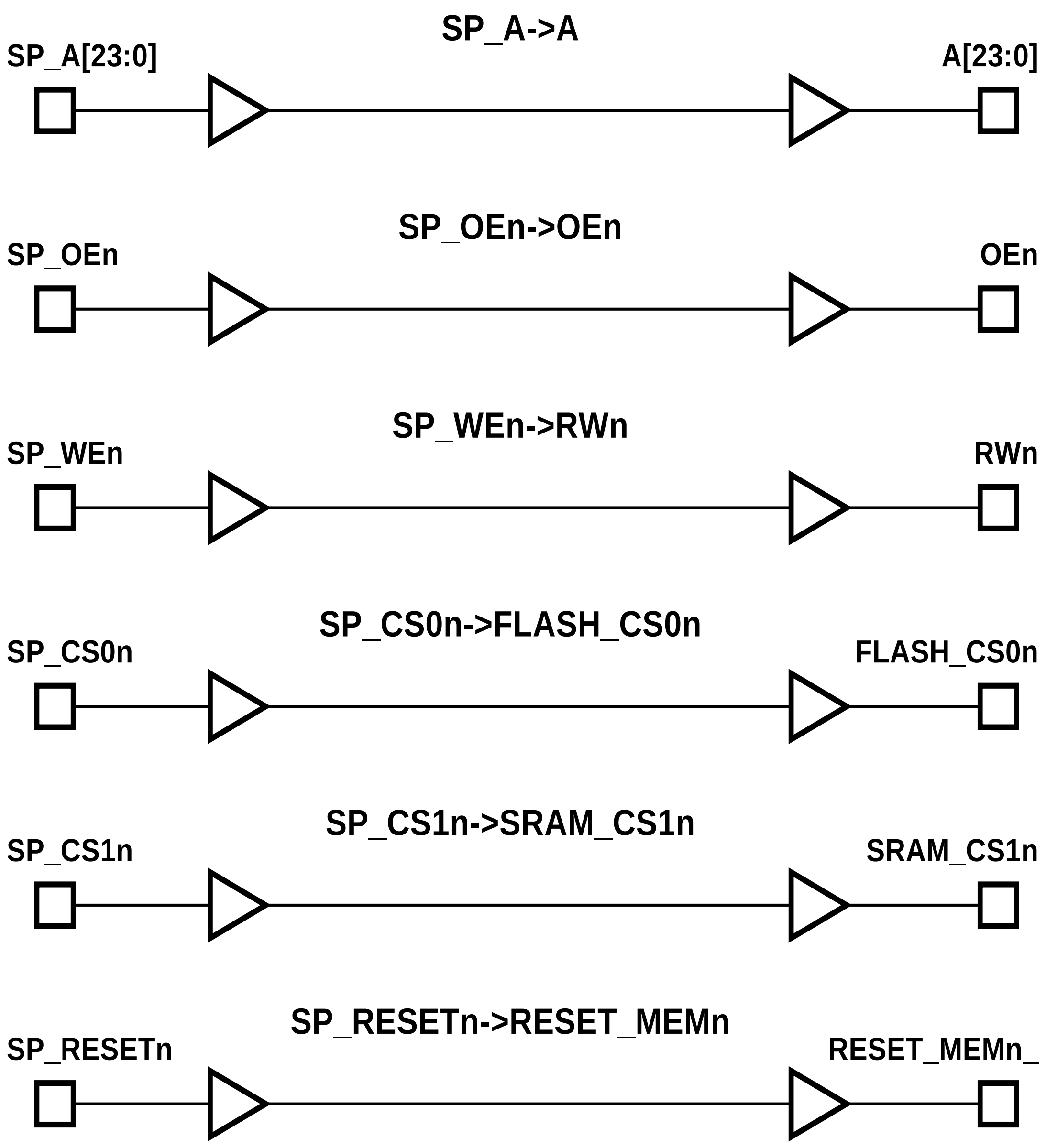


Figure 5: **Simplified CPLD Circuitry for Basic Connections**

Application Software Development

Most applications will interface through the Palm OS in the standard way. Specifically, they will launch from the LCD screen, begin execution, present the user with graphics, tables, menus, push buttons and text. Then, they will perform the target application and return control to the LCD application screen. Palm OS supports menu, table and push button resources. Most Palm OS applications will interact with the user through the push button, menu and table paradigm. The set of push buttons, menus and tables are called Palm OS “resources.”

It is important that software developers have the tools to:

a. create the launch icon graphics

b. create the required push buttons, menus and tables

c. create the source code for the application

d. compile a – c into a “Pilot resource” (.prc) file.

Metrowerks commercial CodeWarrior suite provides all the necessary tools to accomplish the tasks listed in a-d. As a free alternative, the Free Software Foundation provides a set of tools (GNU C++ ToolChain) that operate similar to the CodeWarrior capability. Although very complete, these tools can be daunting to configure and operate. Another alternative approach is the Pocket C program developed at Orbworks, which is an easy, inexpensive and extensible program. The appendices detail how to use Pocket C, the Ccontrols package and additional library extensions to tailor I/O commands to suit your needs.

Memory Spaces

Every add on card is required to come complete with the code that runs it. When a card is inserted into the application slot on the Visor, Palm OS automatically recognizes it, extracts its resource information (.prc) and adds the appropriate icon into the application display GUI. Hence, a minimum Springboard must have a Flash memory module. To make things more interesting for a broader range of applications, the Insight Electronics development module adds a 256 macrocell CoolRunner XPLA3 CPLD, an SRAM and a Texas Instruments A/D converter. The CPLD integrates everything together and onto the Visor Bus. Hence, it is important that developers recognize the options for mapping the memory and I/O spaces onto the bus. We emphasize options here because the CPLD’s reprogrammable capability allows designers to remap the other chips (if needed) to anywhere within the memory range they support.

The figure below shows how memory within a Springboard module is mapped into the memory space of the handheld. Note that the base address and size may change in future Handspring products. For now, the following figure is applicable to all current versions of the Visor.

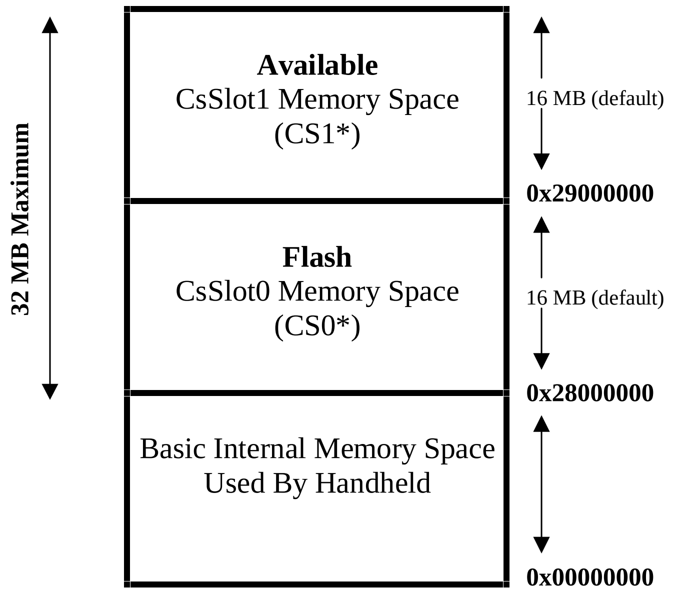


Figure 6: **Module Memory Map**

- As shown above, two chip select lines, CS0* and CS1* (* denotes active low), are output to the springboard.
- The Flash's memory space lies between 0x28000000 and 0x28FFFFFF.
- Any read or write to the flash will be automatically qualified by CS0*
- All other components are accessed between 0x29000000 and 0x29FFFFFF.
- Any read or write operation will be automatically qualified by CS1* and OE* (for reading) or by CS1* and WE* (for writing)

Basic Handshakes

Visor PDAs are currently built with the Motorola DragonBall microprocessor. This family of processors has been constructed with PDA-like applications in mind. They include LCD I/O support circuitry, power monitoring, IRDA circuitry and a host of necessary functions which make PDA development simple. Details of the DragonBall processor can be found at:

www.motorola.com

Conclusions

This application note has presented some of the tools that are available from Insight Electronics, Handspring and Xilinx for fast development of Springboard application cards. It should be viewed as a toolbox of easily accessible information to jump start fast product development with the best technologies available today. The reader is encouraged to seek out additional materials at the hyperlinked websites as well as to download the various free software packages available to him.

The appendices included below contain several useful techniques to aid in quickly creating solutions for today's fast moving needs.

Acknowledge-ments

We would like to thank the following individuals that contributed pieces to this compendium:

Jim Beneke, Insight Electronics

Miles Brown, Handspring

Bill Klein,Texas Instruments

Jeremy Dewey, Orbworks

Mark Ng, Xilinx

Tom Nixon, MemecBoard

Appendix A Springboard Pinout

The Springboard Expansion Slot is a 68-pin Connector that provides Springboard Designers direct access to the Handspring Visor's CPU. Among these accessible pins are 24 address lines, 16 data lines, and a variety of control signals.

The table below is a brief summary of the signals that will be used in this Xilinx demo board. It summarizes the signal names with their signal direction and their respective function. Note that the signal direction (I/O/P/PU) is viewed with reference to the module. For example, CS1* is driven by the CPU and is an input (I) to the module.

A more detailed table can be found on page 46 of the Springboard Development Guide for Handspring Handheld Computers (Release 1.11)

Table 2: **Demo Board Signals**

Signal Name	I/O/P/PU	Function
A[23:0]	I	24-bit Address Bus NOTE: A0 is not used per the Handspring spec
D[16:0]	I/O	16-bit Data Bus
OE*	I	Output Enable
WE*	I	Write Enable
CS0*	I	Chip Select 0
CS1*	I	Chip Select 1
IRQ*	O/PU	Interrupt Request
MIC-	I	Microphone
MIC+	I	Microphone
RESET*	I	Module Reset
LOWBAT*	I	Low Battery
VCC	P	Module VCC
GND	P	Module GND

I=input, O=output, and P=power, with respect to the module. For example, the IRQ signal is driven by the module and is an output to the handheld. PU indicates the signal is internally pulled up within the handheld

* indicates an active low signal

Appendix B Software Development with Pocket C

PocketC is an easy-to-learn program that can be used to develop Springboard applications. 'C' code can be written on the Visor Memopad and can be compiled on the Visor using the PocketC program.

While GNU and MetroWerks offer true development environments, PocketC can do similar things. For the beginner, PocketC is a more intuitive and user-friendly environment. The user interface is straightforward and the language syntax has been simplified. The primary drawback with PocketC is that applications created within that environment must have the PocketC application resident on the handheld.

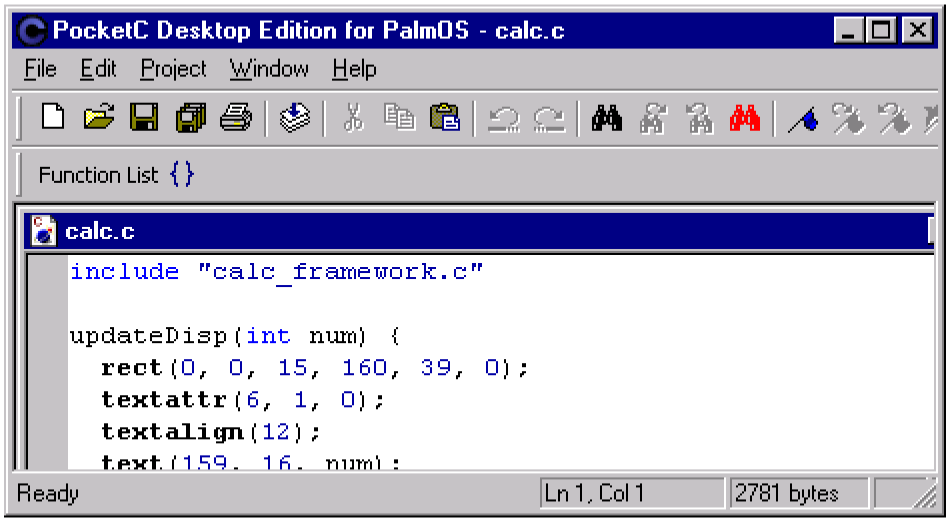
An evaluation version of PocketC can be downloaded from:

http://www.orbworks.com.

This software is shareware, which means that the compile functionality will be disabled after 45 days. The full version, PocketC Desktop Edition can be purchased for a nominal fee from:

http://www.viaweb.com/pilotgearsw/orbworks.html

PocketC Desktop Edition is run on a PC and contains many more advanced features that are described on the website.


PocketC Desktop Edition for PalmOS - calc.c
File Edit Project Window Help
Function List {}
calc.c
Ready
Ln 1, Col 1
2781 bytes

```
include "calc_framework.c"

updateDisp(int num) {
  rect(0, 0, 15, 160, 39, 0);
  textattr(6, 1, 0);
  textalign(12);
  text(159, 16, num);
```

Figure 7: **PocketC GUI**

Appendix C Ccontrols

Ccontrols is an excellent PalmOS program that automatically generates skeleton 'C' code for PocketC to create graphical tasks (i.e. buttons, images, etc.). It, too, is easy to use. Once the skeleton code is created, the designer inserts their own functions and the application is complete.

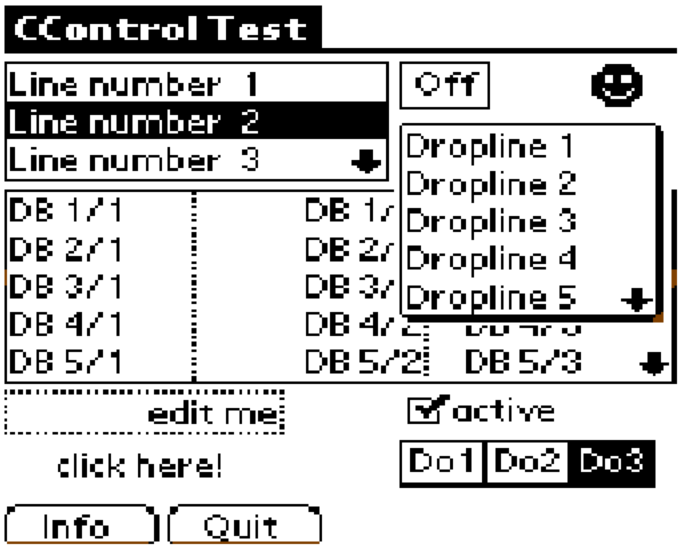


Figure 8: **Ccontrols GUI**

CControls features:

- supports all common palm controls
- supports tables and database-linked tables
- supports real menus
- supports dialog frames
- all controls have palm pilot's standard outfit and behavior customizable outfit for certain controls like edit fields

- all control's contents are created dynamically (the number of added items is only limited by your pilot's memory)
- easy to use: no need to initialize special environments
- a small footprint (controls library uses about 12k / three memos)

Ccontrols can be downloaded from the Orbworks web site, under the Palm OS Resources section:

http://www.orbworks.com/palmres.html

Using PocketC and Ccontrols

Simple Example: Hello World

The following example illustrates how to create a simple PocketC program that contains a single button.

1. On the Visor, launch the Ceditor application:
   a. Tap ‘New’
   b. Select ‘Button’
   c. Cross hairs will now appear. Position the Cross hairs using your stylus in order to choose the final position of the button.

   (When the Cbutton form shows up, notice how the attributes of the button, such as Left, Top, Width, Height, Line, and Edge have already been defined. Should you want to modify these options some more, please refer to the Ccontrols documentation.)

   d. Rename the Button name to something else, i.e. ‘Click Me’. Tap OK
2. Once a button is created, skeleton code must be generated.
   a. Tap on the Menu button.
   b. Choose Form -> Generate code…
   c. Enter the title of your program, i.e. “TestButton”
   d. Ceditor will now generate the ‘C’ code. Tap on ‘Yes’ on each of the 3 upcoming alert boxes.

   Three source files have now been generated:

   “TestButton main” -- top level ‘C’ source file

   “TestButton controls” -- defines the attributes of created symbols

   “TestButton methods” -- Event Loop. Add your code here.
3. To make this skeleton code useful, it must respond when ‘ClickMe’ is tapped. Simply edit the “TestButton methods” memo:

   Add the line, alert(“Hello World”);, to the on_h1() function.

   e.g. change the TestButton methods file:

```
  /$ TestButton methods (Ceditor)
  on_h1(){
    alert(“Hello World”);
  }
messageloop(){
      int e;
        while(1){
        e=event(1);
        if(Cevent(h1,e))
          on_h1();
      }
```

4. Compile this memo in PocketC and execute.

Appendix D Native Libraries

What are Native Libraries?

Native Libraries are used when custom functions are needed. Custom functions are defined as those functions which are not part of the PocketC function list. For example, IORead and IOWrite are 2 custom functions used to drive the Springboard I/O on the Visor. In order to use IORead and IOWrite, the FractalLib Native Library had to be created.

There are many Native Libraries that have already been created by the PocketC community and are free to use. Think of these libraries as extensions which add additional features to the standard PocketC program. Perhaps the most notable library would be Joe Stadolnik's PtoolBoxLib Native Library, which allows for PocketC to do many advanced graphical functions. These Native Libraries can be downloaded from the Orbworks website (www.orbworks.com).

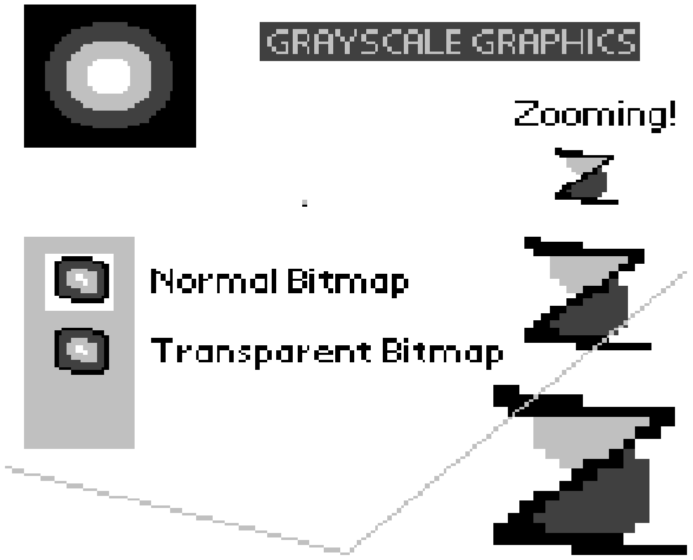


Figure 9: **The PtoolBoxLib**

Springboard I/O using PocketC: IOWrite and IORead

Two functions, IOWrite and IORead, have been created in a native library entitled IOLib.

IORead allows for PocketC to read data at a certain address from the Springboard module.

IOWrite allows for PocketC to write data to the Springboard.

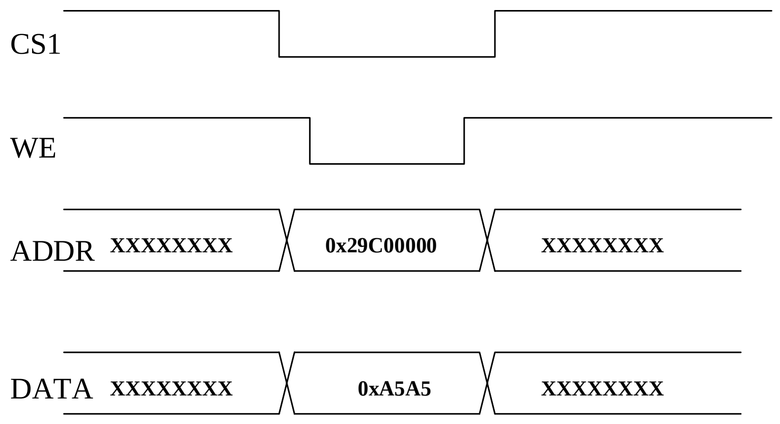
IOWrite(int Addr, int Data)

To use IOWrite(), an address and data must be passed to the function. The address values must be between 0x29000000 and 0x29FFFFFF.

USAGE EXAMPLE:

IOWrite(0x29C00000, 0xA5A5)	Writes 0xA5A5 to 0x29C00000

RESULTING TIMING DIAGRAM:



Notes:

- If no Springboard module is inserted, and the IOWrite function is executed by PocketC, a soft reset may occur on the Visor. This is normal. To avoid this, insert a valid module
- Always check to make sure IOWrite is writing to a proper address. Writing to an invalid address will always cause a reset.

Table 3: **Invalid Addresses**

Do Not Write	Reason
`IOWrite(0x27000000, 0xFFFF)`	Address 0x27000000 is invalid
`IOWrite(0x2800, 0xFFFF)`	Address 0x2800 is invalid
`IOWrite(0x29000001, 0xFF55`	AO is not used. AO must never be 'I'. Otherwise, Visor will reset.

IORead(int Addr)

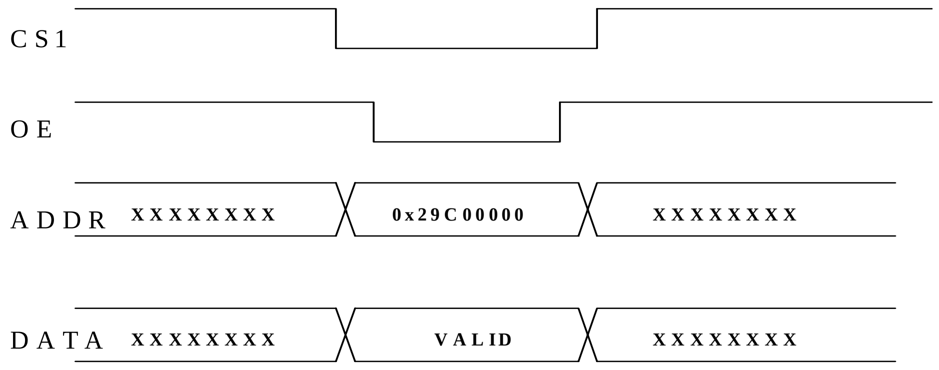
To use IORead, simply pass an address to the function and a 16 bit value will be returned.

The address values must be between 0x29000000 and 0x29FFFFFF.

USAGE EXAMPLE

IORead(0x29C00000)	Reads and returns data at 0x29C00000

RESULTING TIMING DIAGRAM:



Notes:

- If no Springboard module is inserted, and the IOWrite function is executed by PocketC, a soft reset may occur on the Visor. This is normal. To avoid this, insert a valid module
- Always check to make sure IORead is reading from a proper address. Reading from an invalid address will always cause a reset.

Table 4: **Invalid Addresses**

Do Not Read	Reason
`IORead(0x27000000)`	Address 0x27000000 is invalid
`IORead(0x2800)`	Address 0x2800 is invalid
`IORead(0x29000001)`	A0 is not used. A0 must never be 'l'. Otherwise, Visor will reset.

Using Native Libraries

To use a native library, you must first tell the compiler to load the library by using the library keyword. The functions are then used as if they were normal functions.

Example:

```
// My Applet which uses IORead and IOWrite
// IOLib defines IORead and IOWrite

library "IOLib"

main() {

int result;

IOWrite(0x29C00000, 0xFFFF);//write 0xFFFF to address location 0x29C00000
result = IORead(0x29C00000); //read data from 0x29C00000 and store in result

}
```

Creating Native Libraries

Should you want to create your own custom Native Library, please take a look at the "How To" document written by Jeremy Rixon located at:

http://rixon.org/FractalLib/index.html

These instructions should not be followed line for line. They are a bit outdated but they do provide a baseline for creating a library.

NOTE: creating a Native Library requires using the GCC ToolChain. It also requires some knowledge of C++.

Another example of a native library can be found at:

one-each.iwarp.com

The easiest way to create a Native Library would be to work off of Mark Ng's library, IOLib. Take a look at the three source files, and use the makefile that is included. This would only require that you add your own function(s) to the code.

Appendix E Debugging with POSE

Palm OS Emulator, also known as POSE, is a program that runs on Windows. A giant Palm Pilot will appear on the screen, and it will emulate the exact behavior that a Handspring Visor will. POSE is a great debugging tool and is excellent because designers will not have to continually HotSync their Visors.

Installation

1. POSE can be downloaded from:

   http://www.palmos.com/dev/tech/tools/emulator/

2. Run POSE by double clicking on Emulator.exe
3. You will now have to grab the ROM image from the Visor.

   NOTE: Currently the only way to grab a ROM image on a Visor is to use the Serial Port. Follow the instructions on the POSE program to obtain a ROM Image
4. After the ROM Image has been obtained, installation is complete

Usage

POSE is relatively simple to use. Treat POSE as though you were using your mouse as a stylus pen.

To Install a .prc file:

1. Right click on POSE, and select 'Install Application/Database' -> Others.
2. Point to the location of the .prc executable.

   NOTE: After installing a .prc on POSE, the application icon will not be immediately available. Refresh the screen by clicking on the Calendar, Phone Book, To Do, or Memo button and then go back to the Applications screen.

Appendix F Simple Memory Checker C code

The following 'C' code will compile and run using PocketC. It is meant to be used in conjunction with the VHDL code shown in Appendix G. The code shown below will write 0xA5A5 to consecutive addresses in the Toshiba SRAM. In this case, 0xA5A5 is written to SRAM starting from SRAM address location 0x00000 to 0x007FF.

Note that the 'for' loop is incrementing the address values so that they are always even. This is done because A0 (address 0 on the Handspring Visor Springboard Expansion slot) is not used. Writing to this address will reset the Visor unit.

The Address and Data values can easily be modified within each of the two 'for' loops.

```
//SRAM Tester
library "IOLib"

main(){
int i, k, result,error;
error=0;
graph_on();
clearg();
rect(0,0,0,165,165,1);

text(10,10,"writing 0xA5A5 to");

for(i=0x29000000;i<=0x29000FFF;i=i+2){
    text(10, 20, i);
    IOWrite(i, 0xa5a5);
    }

text(10, 30, "Done Writing!");

text(10,50,"Reading.....");

for(i=0x29000000;i<=0x29000FFF;i=i+2){
  text(10, 60, i);
  result=IORead(i);

  if (result == 0xa5a5){
   text(10, 70, "A5A5");
    }
  else{
  text(40, 70, "BAD");
  error=1;
    }
}

if(error==1){
  text(10, 90, "ERROR");
  }
else{

  while(1){
    for(k=0;k<50;k=k+1){
      text(10, 90, "Read/Write Succussful!");
      }

    for(k=0;k<50;k=k+1){
      text(10, 90, "                                              ");
      }
    }
  }
}
```

Appendix G Simple Memory Checker VHDL code

The following VHDL code implements the buffering scheme shown in Figure 5 of this Application Note. This code should be compiled with Xilinx WebPACK software to create a JEDEC file. This JEDEC should then be downloaded to the Xilinx CoolRunner XPLA3 part located on the Insight Springboard Development Card.

```
*******************************************************************
-- File:        SRAMTest.vhd
--
-- Purpose:     This creates the buffering scheme in the XPLA3 as
--              described in Application Note, XAPP147, "Low Power
--              Springboard Design with CoolRunner CPLDs.
--
--              This buffering scheme allows the user to write and
--              read to/from SRAM, FLASH, or any other external component
*******************************************************************

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity coolrunner is
  port(

  -- Original Springboard Signals:

  SP_A :      in        STD_LOGIC_VECTOR(23 DOWNTO 1);
  SP_A_0 :    out       STD_LOGIC;
  SP_D :      inout     STD_LOGIC_VECTOR(15 DOWNTO 0);
  SP_WEn   :  in        STD_LOGIC;
  SP_OEn :    in        STD_LOGIC;
  SP_CS0n:    in        STD_LOGIC;
  SP_CS1n:    in        STD_LOGIC;
  SP_RESETn:  in        STD_LOGIC;


  -- Buffered Signals:

  A     :     out       STD_LOGIC_VECTOR(22 DOWNTO 0);
  D     :     inout     STD_LOGIC_VECTOR(15 DOWNTO 0);
  RWn   :     inout     STD_LOGIC;
  OEn   :     out       STD_LOGIC;
  FLASH_CS0n: out       STD_LOGIC;
  SRAM_CS1n:  inout     STD_LOGIC;
  RESETn :    out       STD_LOGIC;

  --  SRAM Signals:

  FLASH_WR_PROTECT:  out  STD_LOGIC;
  SRAM_UPPER_BYTEn:  out  STD_LOGIC;
  SRAM_LOW_BYTEn:    out  STD_LOGIC;

  --  Expansion Connectors:

  CHIP3_EN: out       STD_LOGIC

   );

end coolrunner;

architecture BEHAVE of coolrunner is
```

```
signal SRAM_Write_En:  std_logic;-- Output Buffer Enable Signal
signal SRAM_Read_En:   std_logic;-- Output Buffer Enable Signal
signal SRAM_Read_Data: std_logic_vector(15 DOWNTO 0);-- 16-bit Output Data
signal SRAM_Write_Data:std_logic_vector(15 DOWNTO 0);-- 16-bit Input Data



begin


--******************************** SP_D -> D *****************************************
--
--                                                         SRAM_Write_En
--                                                                |
--                                          ---------------------|>---
--                                         |                           |
--               SRAM_Read_En              |                            ---[]  D
--                     |                   |    SRAM_Read_Data         |
--                ----<|-------------------|---------------------<|----
--               |                         |
-- SP_D[] --------                         |
--               |       SRAM_Write_Data   |
--                ----|>-------------------
--
--*************************************************************************************



SRAM_Write_En <= not SP_WEn and (not SP_CS0n or not SP_CS1n); -- Enable
Signal


  -- Bidirectional Bus, D (Goes TO/FROM the springboard):

D(15 DOWNTO 0) <=  SP_D(15 DOWNTO 0) when (SRAM_Write_En = ’1’) else (others
=> ’Z’);

  SRAM_Read_Data(15 DOWNTO 0) <= D(15 DOWNTO 0);


SRAM_Read_En   <= not SP_OEn and (not SP_CS0n or not SP_CS1n);  --  Enable
Signal

  -- Bidirectional Bus, DATA (Goes TO/FROM Handspring Unit):

SP_D(15 DOWNTO 0) <= SRAM_Read_Data(15 DOWNTO 0) when (SRAM_Read_En = ’1’)
else (others => ’Z’);

  SRAM_Write_Data(15 DOWNTO 0) <= SP_D(15 DOWNTO 0);


******************************** SP_A -> A ********************************
--
--SP_A[] ------|>-----------------------------|>------ A[]
--
***************************************************************************



A(22 DOWNTO 0) <= SP_A(23 DOWNTO 1); --  Since A0 is not used, level shift
SP_A_0         <= ’0’;               --  SP_A(0) is not used in this version
                                          of Handspring Visor
```

```
*************************** SP_CS0n -> FLASH_CS0n *************************
--
--   SP_CS0n [] ------|>-----------------------------|>------ CS0n[]
--
***************************************************************************


FLASH_CS0n     <= SP_CS0n;


*************************** SP_CS1n -> SRAM_CS1n *************************** 
--
--   SP_CS1n [] ------|>-----------------------------|>------ SRAM_CS1n[]
--
***************************************************************************

SRAM_CS1n      <= SP_CS1n;


*************************** SP_CS1n -> SRAM_CS1n***************************
--
--   SP_WEn [] ------|>-----------------------------|>------ RWn[]
--
***************************************************************************


RWn            <= SP_WEn;

***************************** SP_OEn -> OEn*******************************
--
--   SP_OEn [] ------|>-----------------------------|>------ OEn[]
--
***************************************************************************


OEn            <= SP_OEn;

************************** RESETn -> SP_RESETn ****************************
--
--SP_RESETn [] ------|>-----------------------------|>------ RESET[]
--
***************************************************************************

RESETn         <= SP_RESETn;



--SRAM SIGNALS:
SRAM_UPPER_BYTEn <= ’0’;  -- Enable Upper Byte
SRAM_LOW_BYTEn   <= ’0’;  -- Enable Lower Byte
FLASH_WR_PROTECT <= ’1’;


--Expansion Connector Signals:

CHIP3_EN       <= SP_CS1n;  -- Bring SP_CS1n signal to a pin that we can
                             probe.

END BEHAVE;
```

Appendix H LED Test C Code

The 'LED Test' PocketC source code is shown below. This code creates 4 buttons that, when used in conjunction with the LED Test VHDL code shown in Appendix I, controls each of the 4 LED's on the Insight Springboard Development Card. Each button in this application can be turned on or off. For example, by looking at the 'LED test methods' file, the button named 'h1' will output data 0xFFFF to address 0x00003E when turned on. When turned off, the button will output 0x0000 to the same address.

These address values are then decoded by the CoolRunner so that it can blink the appropriate LED.

Note: 3 separate files are shown below, namely 'Led test main', 'Led test ccontrols', and 'Led test methods'. These files have been created using Ccontrols (see Appendix C).

```
// Led test main (CEditor)
library "IOLib"
include "Led test controls (CEditor)"
include "Led test methods (CEditor)"


main(){
graph_on();
title("Led test");
text(60,50,"LED1");
text(60,75,"LED2");
text(60,100,"LED3");
text(60,125,"LED1");
initcontrols();
initcontents();
inititems();
drawcontrols();
messageloop();
}


/$ Led test controls (CEditor)
include "Ccontrols.c"
Chandle  h1, h2, h3, h4;
initcontrols(){
h1=Cswitch(20,50,30,12,1,0);
h2=Cswitch(20,75,30,12,1,0);
h3=Cswitch(20,100,30,12,1,0);
h4=Cswitch(20,125,30,12,1,0);
}
initcontents(){
Csetcontent(h1,"OFF");
Csetcontent(h2,"OFF");
Csetcontent(h3,"OFF");
Csetcontent(h4,"OFF");
}
inititems(){
}
drawcontrols(){
Cdraw(h1);
Cdraw(h2);
Cdraw(h3);
Cdraw(h4);
}


/$ Led test methods (CEditor)
on_h1(){
```

```
if(Cgetstate(h1)){
     Csetcontent(h1, "ON");
     IOWrite(0x2900003e,0xffff); //write 0xFFFF to address 0x2900003e
      }
else{
     Csetcontent(h1,"OFF");
     IOWrite(0x2900003e,0x0000); //write 0x0000 to address 0x2900003e
     }

Cdraw(h1);

}
on_h2(){
if(Cgetstate(h2)){
     Csetcontent(h2, "ON");
     IOWrite(0x29000fc0,0xffff); //write 0xFFFF to address 0x29000fc0
      }
else{
     Csetcontent(h2,"OFF");
     IOWrite(0x29000fc0,0x0000); //write 0x0000 to address 0x29000fc0
      }
Cdraw(h2);
}

on_h3(){
if(Cgetstate(h3)){
     Csetcontent(h3, "ON");
      IOWrite(0x2903f000,0xffff); //write 0xFFFF to address 0x2903f000

      }
else{
     Csetcontent(h3,"OFF");
     IOWrite(0x2903f000,0x0000); //write 0x0000 to address 0x2903f000

      }
Cdraw(h3);
}

on_h4(){
if(Cgetstate(h4)){
     Csetcontent(h4, "ON");
     IOWrite(0x29fc0000,0xffff); //write 0xFFFF to address 0x29fc0000

      }
else{
     Csetcontent(h4,"OFF");
    IOWrite(0x29fc0000,0x0000); //write 0x0000 to address 0x29fc00000

      }

Cdraw(h4);
}
messageloop(){
int e;
while(1){
e=event(1);
if(Cevent(h1,e)) on_h1();
else if(Cevent(h2,e)) on_h2();
else if(Cevent(h3,e)) on_h3();
else if(Cevent(h4,e)) on_h4();
}
}
```

Appendix I: LED Test VHDL Code

This VHDL code is meant to be used in conjunction with the PocketC code shown in Appendix H. It decodes address and data lines from Handspring Visor and blinks the appropriate LED. The 2.5 MHz CCLK output (generated by Texas Instruments ADS7870 A/D) is divided to generate a 4.76 Hz signal. This 4.76 Hz clock is used to toggle the selected LED

```
***************************************************************************
-- File:   led_test.vhd
--
-- Purpose: LED test code for Insight Springboard Development Card.
--          When used in conjunction with the LED test 'C' code, this VHDL
--          code decodes address and data lines from Handspring Visor and
--          Blinks appropriate LED.  The 2.5 MHz CCLK output (generated
--          by Texas Instruments ADS7870 A/D) is divided to generate a 4.76
--          Hz signal. This 4.76 Hz clock is used to toggle the selected LED
--
***************************************************************************


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity LED_TEST is
    port(

      -- Handspring port interface
     SP_A     : in STD_LOGIC_VECTOR(23 downto 0);  -- 24-bit address bus
     SP_D     : in STD_LOGIC_VECTOR(15 downto 0);  -- 16-bit data bus
     SP_CS1n  : in STD_LOGIC;-- CS1                -- Active low chip select
     SP_WEn   : in STD_LOGIC;-- WE


     -- LED interface
    led1_sel  : out STD_LOGIC;    --LED1 pin
    led2_sel  : out STD_LOGIC;    --LED2 pin
    led3_sel  : out STD_LOGIC;    --LED3 pin
    led4_sel  : out STD_LOGIC;    --LED4 pin

    -- ADC interface
    cclk          :  in STD_LOGIC;      -- Output CCLK
    ad_osc_en     :  out STD_LOGIC;     -- Oscillator Enable
    ad_convert    :  out STD_LOGIC;     -- Convert Pin
    ad_chip2en_n  :  out STD_LOGIC;     -- A/D Chip Select
    ad_rise_fall  :  out STD_LOGIC;     -- Rise/Fall pin
    serial_tx     :  out STD_LOGIC;     -- Din
    serial_clk    :  out STD_LOGIC;     -- SClk

          -- Active low write enable

    -- SRAM Interface

    SRAM_CS1n         : out STD_LOGIC; --SRAM CS
    SRAM_UPPER_BYTEn  : out STD_LOGIC; --SRAM Upper Byte Enable
    SRAM_LOW_BYTEn    : out STD_LOGIC; --SRAM Lower Byte Enable

    --Flash Interface

    FLASH_CS0n        : out STD_LOGIC; --Flash CS
    FLASH_WR_PROTECT  : out STD_LOGIC; --Flash /WP/ACC pin
    --SRAM and Flash Interface (shared)

    RWn     : out STD_LOGIC; --WE for SRAM and Flash
    OEn     : out STD_LOGIC; --OE for SRAM and Flash
    RESETn  : out STD_LOGIC;

    CHIP3_EN : out STD_LOGIC --To bring CS out to Expansion Connector

           );

end LED_TEST;



architecture BEHAVIOUR of LED_TEST is

-- ******************** CONSTANT DECLARATIONS ***********************
constant LED1_ADDR : STD_LOGIC_VECTOR(23 downto 0) :=
"000000000000000000111110";  -- A1 to A5 asserted
constant LED2_ADDR : STD_LOGIC_VECTOR(23 downto 0) :=
"000000000000111111000000";  -- A6 to A11 asserted
constant LED3_ADDR : STD_LOGIC_VECTOR(23 downto 0) :=
"000000111111000000000000";  -- A12 to A17 asserted
constant LED4_ADDR : STD_LOGIC_VECTOR(23 downto 0) :=
"111111000000000000000000";  -- A18 to A23 asserted


-- ********************* SIGNAL DECLARATIONS ************************
signal toggle_clk    : STD_LOGIC;-- Divided clk for toggling LEDs

signal led1_flag     : STD_LOGIC;-- LED Flag set by one ASSERT_LED1_FLAG
process,
                      -- and read by READ_LED1_FLAG process
signal led2_flag     : STD_LOGIC;-- LED Flag set by one ASSERT_LED2_FLAG
process,
                      -- and read by READ_LED2_FLAG process
signal led3_flag     : STD_LOGIC;-- LED Flag set by one ASSERT_LED3_FLAG
process,
                      -- and read by READ_LED3_FLAG process
signal led4_flag     : STD_LOGIC;-- LED Flag set by one ASSERT_LED4_FLAG
process,
                      -- and read by READ_LED4_FLAG process


-- ******************** COMPONENT DECLARATION ***********************
-- Clock Divider Function
component CLK_DIVIDER
  port(
    input_clk    : in STD_LOGIC;
    output_clk   : out STD_LOGIC );

end component;


begin

  -- ***************** SIGNAL ASSIGNMENTS **********************


  -- **************** COMPONENT ASSIGNMENT *********************
  -- Clock Divider Function
  CLK_DIV: CLK_DIVIDER
```

```
    port map(
      input_clk=> cclk,
        output_clk=> toggle_clk );



  -- ***************** Process: LED1_FLAG ************************
  -- Purpose:   Read address and data lines.  Assert 'led1_flag'
  --            when address and data are correct
  --            LED_FLAG is then read by the corresponding READ_LED_FLAG
  --            Process
  -- Components: none

ASSERT_LED1_FLAG: process(toggle_clk, SP_A, SP_D, SP_WEn, SP_CS1n)
begin
  if(SP_CS1n = '0') then
    if(SP_WEn'event and SP_WEn='1') then
      if (SP_D = "1111111111111111" and SP_A = LED1_ADDR) then
      led1_flag <= '1';
      elsif (SP_D = "0000000000000000" and SP_A = LED1_ADDR) then
      led1_flag<= '0';
      end if;
    end if;
   end if;
end process ASSERT_LED1_FLAG;



  -- ***************** Process: ASSERT_LED2_FLAG ************************
  -- Purpose:   Read address and data lines.  Assert 'led2_flag'
  --            when address and data are correct
  --            LED_FLAG is then read by the corresponding READ_LED_FLAG
  --            Process
  -- Components: none

ASSERT_LED2_FLAG: process(toggle_clk, SP_A, SP_D, SP_WEn, SP_CS1n)
begin
  if(SP_CS1n = '0') then
    if(SP_WEn'event and SP_WEn='1') then
      if (SP_D = "1111111111111111" and SP_A = LED2_ADDR) then
      led2_flag <= '1';
      elsif (SP_D = "0000000000000000" and SP_A = LED2_ADDR) then
      led2_flag<= '0';
      end if;
    end if;
   end if;
end process ASSERT_LED2_FLAG;


  -- ***************** Process: ASSERT_LED3_FLAG ************************
  -- Purpose:   Read address and data lines.  Assert 'led3_flag'
  --            when address and data are correct
  --            LED_FLAG is then read by the corresponding READ_LED_FLAG
  --            Process
  -- Components: none

ASSERT_LED3_FLAG: process(toggle_clk, SP_A, SP_D, SP_WEn, SP_CS1n)
begin
  if(SP_CS1n = '0') then
    if(SP_WEn'event and SP_WEn='1') then
      if (SP_D = "1111111111111111" and SP_A = LED3_ADDR) then
      led3_flag <= '1';
      elsif (SP_D = "0000000000000000" and SP_A = LED3_ADDR) then
      led3_flag <= '0';
      end if;
    end if;
   end if;
end process ASSERT_LED3_FLAG;


  -- ***************** Process: ASSERT_LED4_FLAG ************************
  -- Purpose:   Read address and data lines.  Assert 'led4_flag'
  --            when address and data are correct
  --            LED_FLAG is then read by the corresponding READ_LED_FL
  --            Process
  -- Components: none

ASSERT_LED4_FLAG: process(toggle_clk, SP_A, SP_D, SP_WEn, SP_CS1n)
begin
  if(SP_CS1n = '0') then
    if(SP_WEn'event and SP_WEn='1') then
      if (SP_D = "1111111111111111" and SP_A = LED4_ADDR) then
      led4_flag <= '1';
      elsif (SP_D = "0000000000000000" and SP_A = LED4_ADDR) then
      led4_flag <= '0';
      end if;
    end if;
   end if;
end process ASSERT_LED4_FLAG;


  -- ***************** Process: READ_LED1_FLAG ************************
  -- Purpose:   Read the LED1_FLAG.  If LED1_FLAG is asserted (logic high),
  --            toggle the LED.  Otherwise, turn off the LED.
  -- Components: none


READ_LED1_FLAG: process(led1_flag, toggle_clk)
begin
  if (led1_flag = '1') then
    led1_sel <= toggle_clk; --Blink LED
  else
    led1_sel <= '1';     --Turn OFF LED
  end if;
end process READ_LED1_FLAG;

  -- ***************** Process: READ_LED2_FLAG ************************
  -- Purpose:   Read the LED2_FLAG.  If LED2_FLAG is asserted (logic high),
  --            toggle the LED.  Otherwise, turn off the LED.
  -- Components: none


READ_LED2_FLAG: process(led2_flag, toggle_clk)
begin
  if (led2_flag = '1') then
    led2_sel <= toggle_clk; --Blink LED
  else
    led2_sel <= '1';     --Turn OFF LED
  end if;
end process READ_LED2_FLAG;

  -- ***************** Process: READ_LED3_FLAG ************************
  -- Purpose:   Read the LED3_FLAG.  If LED3_FLAG is asserted (logic high),
  --            toggle the LED.  Otherwise, turn off the LED.
  -- Components: none

READ_LED3_FLAG: process(led3_flag, toggle_clk)
begin
  if (led3_flag = '1') then
    led3_sel <= toggle_clk; --Blink LED
  else
    led3_sel <= '1';     --Turn OFF LED
  end if;
end process READ_LED3_FLAG;

  -- ***************** Process: READ_LED4_FLAG ************************
  -- Purpose:  Read the LED4_FLAG.  If LED4_FLAG is asserted (logic high),
  --           toggle the LED.  Otherwise, turn off the LED.
  -- Components: none

READ_LED4_FLAG: process(led4_flag, toggle_clk)
begin
  if (led4_flag = '1') then
    led4_sel <= toggle_clk; --Blink LED
  else
    led4_sel <= '1';     --Turn OFF LED
  end if;
end process READ_LED4_FLAG;

--SRAM Signals  (Turn OFF the SRAM)
SRAM_UPPER_BYTEn   <= '1';
SRAM_LOW_BYTEn     <= '1';
SRAM_CS1n          <= '1';

--Flash Signals (Turn OFF the Flash)
FLASH_CS0n         <= '1';
FLASH_WR_PROTECT   <= '1';

--Shared Signals (Turn OFF SRAM and Flash)
RWn                <= '1';
OEn                <= '1';
RESETn             <= '1';

--A/D Signals    (Turn OFF A/D but enable CCLK output)
ad_osc_en          <= '1'; --Osc En pin = '1' to enable CCLK (2.5 MHz)
ad_convert         <= '0';
ad_chip2en_n       <= '1';
ad_rise_fall       <= '0';
serial_tx          <= '0';
serial_clk         <= '0';


--So that CS can be probed
CHIP3_EN           <= SP_CS1n;  --Bring CS pin from Springboard Slot to
                                  CHIP3_EN pin
           --Not necessary in this demo, but it gives the ability to
           --see what CS is doing.  Also nice to probe with a logic analyzer


end BEHAVIOUR;
```

```
******************************************************************
-- File:  clk_divider.vhd
--
-- Purpose: 20x input clock divider.
**************************************************************************

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;


entity CLK_DIVIDER is
  port(
       input_clk   : in STD_LOGIC;   -- Input Clock  (2.5 MHz Cclk from
                                        ADS7870 A/D)
       output_clk: out STD_LOGIC);   -- Clock divider output


end CLK_DIVIDER;



architecture DEFINITION of CLK_DIVIDER is


-- ******************** SIGNAL DECLARATIONS ***********************
signal div : UNSIGNED (19 downto 0) := (others => '0');

begin

  -- ******************** Process: CNT_PROCESS ************************
    -- Purpose: Defines the 20x counter
  --
    -- Components:none

  CNT_PROCESS:process(input_clk)
  begin

    -- On rising edge of clock
    if (input_clk'event and input_clk = '1') then
      div <= div + 1;
    end if;

      end process;

    output_clk <= div(19);

end DEFINITION;
```

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/25/01	1.0	Initial Xilinx release.