XAPP146 (v1.1) January 3, 2002

Designing an Eight Channel Digital Volt Meter with the Insight Springboard Development Kit

Summary

Personal Digital Assistants, such as the Handspring Visor™, are increasingly being used for data acquisition. One such application is the Digital Volt Meter, a device commonly used to measure voltage at a particular source. This Application Note will discuss the design of an eight channel Digital Volt Meter for the Handspring Visor. Specifically, it will illustrate how to use a Xilinx ultra-low power CoolRunner CPLD to interface a Texas Instruments ADS7870 Data Acquisition System to the Handspring Spingboard™ expansion slot.

This Application Note is intended for all Springboard designers. While it is true that the accompanying design files are intended for an Insight Springboard Development Kit, the concepts presented here are applicable to any other Springboard design.

This 8-channel Digital Volt Meter design builds upon the TI ADS7870 Data Aquisition System Interface described in application note **XAPP355**. This Application Note will not discuss the ADS7870 Interface in detail. Readers should familiarize themselves wiith XAPP355 before proceeding.

All related source code will also be provided for download. To obtain the VHDL code described in this document, go to section **VHDL Code Download**, page 9 for instructions.

Overview

Operation of the Digital Volt Meter is simple. Once the Handspring application is launched, the Visor's LCD screen will automatically display and update the voltages on all eight channels of the ADS7870. Therefore, all a user must do is connect any of the eight analog inputs of the ADS7870 to a voltage source. In the default configuration, all eight channels of the data aquisition system are configured for single ended operation, which means that single ended input can vary between 0V to 2.5V. If a user desires wider input voltage, external biasing can be introduced. Also, if the voltage source in question has differential outputs, the inputs to the ADS7870 can easily be configured as differential. However, these options will not be discussed here, as this reference design covers only the default configuration.

Operational Flow

Figure 1 shows the operational flow of the Digital Volt Meter design. The Handspring application enters an infinite loop in which it first issues a command to the Xilinx CoolRunner CPLD. Upon receiving this specific data and address value, the CoolRunner device commands the ADS7870 to begin converting. Immediately after the CoolRunner device orders the ADS7870 to convert, the first analog input channel is sampled eight times, with each result written to SRAM locations 1-8, respectively. Next, the second analog input channel is sampled eight times, with each result stored in SRAM locations 9-16. This continues until the eighth (and last) analog input channel is sampled and stored in SRAM locations 57-64. When all results have been written to SRAM, the CoolRunner device allows the Visor to read the contents, calculate the voltage, and display it to the screen.

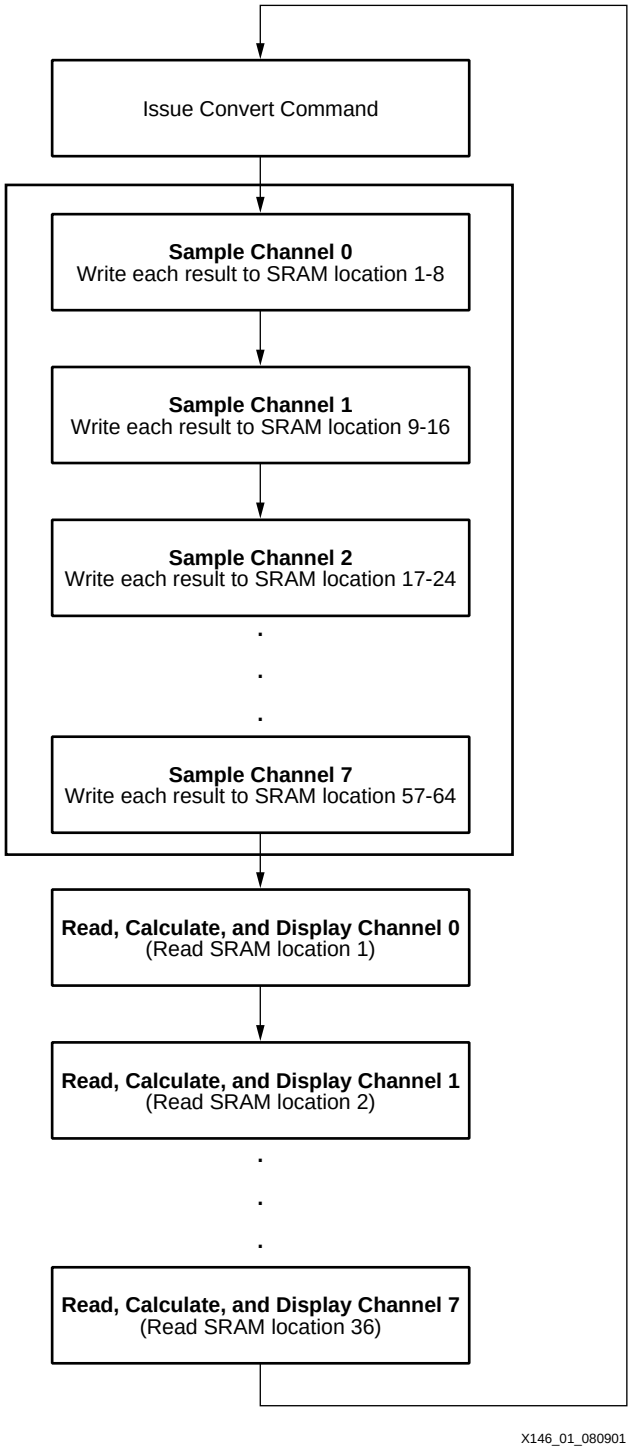

Figure 1: **Operational Flow of Digital Volt Meter Design**

Notice that the boxed area in Figure 1 represents tasks which hardware does. The boxes outside of the shaded area represent tasks software does (Handspring). In other words, the data aquisition process is done completely in hardware, while data gathering is done by the Handspring Visor.

This is a classic scheme, as many processors are not fast enough handle high speed data transfers. This is especially true in the case of the Handspring Visor, which utilizes a Motorola Dragonball processor running at 16 MHz (33 MHz on the premium models). For very fast

transactions, such as handling data from a high speed A/D converter, dedicated external hardware is often used to handle the bus transactions. The data can then be stored in memory (SRAM) for the slower processor to read.

In this case, the Xilinx CoolRunner CPLDs combination of high speed and low power make it an ideal candidate for high speed data manipulation.

VHDL Interface

XAPP355 provides and explains the Texas Instruments ADS7870 Data Aquisition System interface. The VHDL code presented in XAPP355 is intended to be a "building block" for future designs. A detailed understanding of the VHDL code is not needed. Rather, the designer needs only to focus on the details of the ADS7870. If certain aspects of the ADS7870 need to be adjusted, the "constants" section of the VHDL code can then be modified to accordingly.

This reference design shows how to customize the original code presented in XAPP355. Slight changes have been made to the "constants" section. Figure 2 below shows the portions of the "constants" section that have been modified.

As shown, all eight analog input channels of the ADS7870 have been enabled and have been configured for single-ended operation. The locations in SRAM that will store each channels' conversion results have also been defined. Figure 2 shows the conversion result address map.

```
*************** 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";

-- *************** 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 *********************
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";
```

Figure 2: **Modified "Constants" Section**

Table 1: **Conversion Result Address Map**

Channel	Sample #	SRAM Location (A17..A0)	SRAM Location (Decimal)
0	1	00000000000000001	1
	8	00000000000001000	8
1	1	00000000000001001	9
	8	00000000000010000	16
2	1	00000000000010001	17
	8	00000000000011000	24
3	1	00000000000011001	25
	8	00000000000100000	32
4	1	00000000000100001	33
	8	00000000000101000	40
5	1	00000000000101001	41
	8	00000000000110000	48
6	1	00000000000110001	49
	8	00000000000111000	56
7	1	00000000000111001	57
	8	00000000001000000	64

PocketC Code

The PocketC source code is shown in **Appendix A: PocketC Source Code**, page 7. As stated in **XAPP147**, the Xilinx Native Library, "IOLib.prc", which defines the functions "IORead" and "IOWrite", is needed in order for PocketC to access the Springboard IO. IOLib.prc must also be installed on the Handspring Visor in order for this DVM Application to work.

The PocketC code is simple because the ADS7870 is interfaced through hardware. The program is comprised of an infinite loop during which two major tasks take place:

1) Initiating a new conversion

2) Retrieving, computing and displaying results stored in SRAM

Initiating a New Conversion

The CoolRunner CPLD will only begin a new conversion process upon receiving a Springboard address of 0x2900003E and a Springboard data value of 0xFFFF followed by an address of 0x2900003E and a data value of 0x0000.

The following two PocketC commands accomplish this:

```
IOWrite(0x2900003e,0xffff);
IOWrite(0x2900003e,0x0000);
```

Retrieving, Computing and Displaying Results

Immediately after initiating a new conversion, the software retrieves and computes the conversion results for all 8 channels, one by one. Notice that a wait state is not needed between the time when a new conversion is initiated and the time when the results are retrieved. The hardware will have completed its entire chain of events well before the software executes its next line of code.

The PocketC code that is used to retrieve, compute and display the results for channel 1 is shown below:

```
//Channel 1
result = IORead(0x29000002);
result=result >> 4;
Channel1 = result;
Channel1 = (Channel1/2047) * 2.5;
text(70,10, format(Channel1,2));
```

Table 1 shows the address values of each conversion result. However, keep in mind that these address values are with respect to the SRAM. Because Springboard Address 0 (A0) is not used, Springboard A1 is connected to A0 of the SRAM. This means that the address values are slightly shifted. Table 2 summarizes this.

For Example, the results of Channel 1, Sample 1 are stored in SRAM location 1. Since A0 is not used, A1 is connected to A0 of the SRAM. Therefore, SRAM address 1 corresponds to Springboard Address 2. Since csSlot1 starts at a base of 0x29000000 by default, the function IORead(0x29000002) will read from SRAM location 1.

Also notice that the software only reads the first sample of each channel. For simplicity's sake, the remaining seven samples are never read. This illustrates the point that the hardware is always faster than the software.

Table 2: **Springboard Address Values**

Channel	Sample #	SRAM Address (Decimal)	Springboard Offset (Decimal)	Springboard Offset (Hex)	Springboard Address (Hex)
1	1	1	2	0x000002	0x29000002
	8	8	16		
2	1	9	18	0x000012	0x29000012
	8	16	32		
3	1	17	34	0x000022	0x29000022
	8	24	48		
4	1	25	50	0x000032	0x29000032
	8	32	64		
5	1	33	66	0x000042	0x29000042
	8	40	80		
6	1	41	82	0x000052	0x29000052
	8	48	96		
7	1	49	98	0x000062	0x29000062
	8	56	112		
8	1	57	114	0x000072	0x29000072
	8	64	128		

For any given conversion, the ADS7870 stores the 12-bit conversion result in two internal registers, ADDR0 and ADDR1. Both internal registers are each 8 bits wide. When combined, they are sixteen bits wide. Since the conversion result is always 12 bits, there are 4 bits of extra data. Of these four bits, three are unused, and one is used for an overflow flag. Table 3 and Table 4 show the arrangement of ADDR1 and ADDR1.

Table 3: **Contents of ADDR1, the MS Byte**

ADDR 1 (MS Byte)							
D7	D6	D5	D4	D3	D2	D1	D0
ADC11	**ADC10**	**ADC9**	**ADC8**	**ADC7**	**ADC6**	**ADC5**	**ADC4**

Table 4: **Contents of ADDR0, the LS Byte**

ADDR 0 (LS Byte)							
D7	D6	D5	D4	D3	D2	D1	D0
ADC3	**ADC2**	**ADC1**	**ADC0**	**0**	**0**	**0**	**OVR**

Since the SRAM on the Insight Springboard Development Board is 16 bits wide, the ADS7870 interface, as described in **XAPP355**, writes to each SRAM location the contents of ADDR1 followed by the contents of ADDR0. (ADDR1 is Most Significant Byte and ADDR0 is the Least Significant Byte. This is illustrated in Figure 3.)

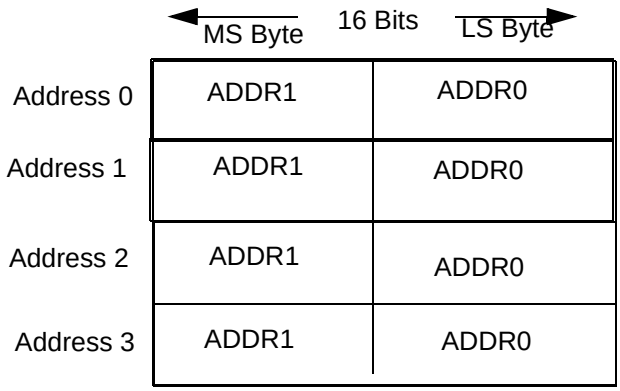

Figure 3: **Contents of SRAM Memory**

Therefore, when software retrieves the conversion result (by executing the 'IORead' function), sixteen bits are returned. To calculate voltage, the result must first be shifted four bits to the right, thereby eliminating the four extra bits and leaving only the 12-bit conversion result.

Once the four bit shift is complete, calculating voltage from the remaining 12-bit result is straightforward. Since the ADS7870 is set for single ended operation, the input voltage may vary from 0V-2.5V. The output codes of the ADS7870 will range from 0 to 2047.

The following formula may then be used to compute measured voltage:

[(12-bit result) / 2047] * 2.5V = Measured Voltage

The measured voltage will the be displayed to the screen and the software will do the same procedure for all other channels. After all channels have been updated, the loop starts again.

Notice that in this implementation, the OVR bit is not monitored. Again this is done for simplicity and software can be adjusted accordingly if you would like to monitor this bit.

Conclusion

The Xilinx CoolRunner CPLD is ideal for Springboard applications. Its re-programmability and its abundance of logic resources allow users to successfully prototype and debug Springboard modules without having to make board changes. In addition, the Xilinx CoolRunner CPLD's

exclusive combination of ultra low power and high speed make it the only feasible programmable logic solution for Springboard designs.

Appendix A: PocketC Source Code

//8 Channel digital volt meter

```
@cid "FFFF";
@ver "1.0";
@name "8 Channel DVM";
@dbname "8 Channel DVM";
@licon1 "xilinx.bmp";
@sicon1 "small1.bmp";

library "IOLib"

main(){

int result;
int i;
float Channel1, Channel2, Channel3, Channel4, Channel5, Channel6, Channel7,
Channel8;

graph_on();
clearg();
rect(0,0,0,165,165,1);
textattr(0,1,0);

text(10,10, "Channel 1");
text(10,20, "Channel 2");
text(10,30, "Channel 3");
text(10,40, "Channel 4");
text(10,50, "Channel 5");
text(10,60, "Channel 6");
text(10,70, "Channel 7");
text(10,80, "Channel 8");

text(90,10, "V");
text(90,20, "V");
text(90,30, "V");
text(90,40, "V");
text(90,50, "V");
text(90,60, "V");
text(90,70, "V");
text(90,80, "V");


while(1){

IOWrite(0x2900003e,0xffff);

IOWrite(0x2900003e,0x0000);

//Now, Clear the numbers:

text(70,10, "   ");
text(70,20, "   ");
text(70,30, "   ");
text(70,40, "   ");
text(70,50, "   ");
text(70,60, "   ");
text(70,70, "   ");
```

```
text(70,80, "   ");

//Channel 1
result = IORead(0x29000002);
result=result >> 4;
Channel1 = result;
Channel1 = (Channel1/2047) * 2.5;
text(70,10, format(Channel1,2));

//Channel 2
result = IORead(0x29000012);
result = result >> 4;
Channel2 = result;
Channel2 = (Channel2/2047) * 2.5;
text(70,20, format(Channel2,2));
//Channel 3
result = IORead(0x29000022);
result=result >> 4;
Channel3 = result;
Channel3 = (Channel3/2047) * 2.5;
text(70,30, format(Channel3,2));
//Channel 4
result = IORead(0x29000032);
result=result >> 4;
Channel4 = result;
Channel4 = (Channel4/2047) * 2.5;
text(70,40, format(Channel4,2));
//Channel 5
result = IORead(0x29000042);
result=result >> 4;
Channel5 = result;
Channel5 = (Channel5/2047) * 2.5;
text(70,50, format(Channel5,2));
//Channel 6
result = IORead(0x29000052);
result=result >> 4;
Channel6 = result;
Channel6 = (Channel6/2047) * 2.5;
text(70,60, format(Channel6,2));
//Channel 7
result = IORead(0x29000062);
result=result >> 4;
Channel7 = result;
Channel7 = (Channel7/2047) * 2.5;
text(70,70, format(Channel7,2));
//Channel 8
result = IORead(0x29000072);
result=result >> 4;
Channel8 = result;
Channel8 = (Channel8/2047) * 2.5;
text(70,80, format(Channel8,2));

for(i=0;i<800;i++);  //wait for a little bit before refreshing the screen

//volt2=result2;
//volt2=(volt2/2047)*2.5;
//text(50,110,format(volt2,2));
//text(70,110,"volts");
}

event(1);
}
```

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.

XAPP146 - http://www.xilinx.com/products/xaw/coolvhdlg.htm

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/24/01	1.0	Initial Xilinx release.
01/03/02	1.1	Minor revisions