XAPP672 (1.0) September 2, 2003

The UltraController Solution: A Lightweight PowerPC Microcontroller

Author: Glenn C. Steiner

Summary

The UltraController™ embedded processor solution is available as a complete reference design, with documentation, to be utilized as a lightweight PowerPC™ microcontroller. The 32-bit input / 32-bit output design created as a simple block, ready to integrate into larger designs, requires only a reset and a clock input. The UltraController solution utilizes the available PowerPC processor(s) in the Virtex-II Pro™ device and several block RAMs. The UltraController design is available for a variety of applications including logic and data control, device configuration, system monitoring, and simple data manipulation. A reference design, created both in fabric and on the UltraController processor, clearly demonstrates substantial fabric savings by moving slow logic into the UltraController processor. This allows users to reduce cost by utilizing smaller devices. A block diagram of the UltraController solution is shown in Figure 1.

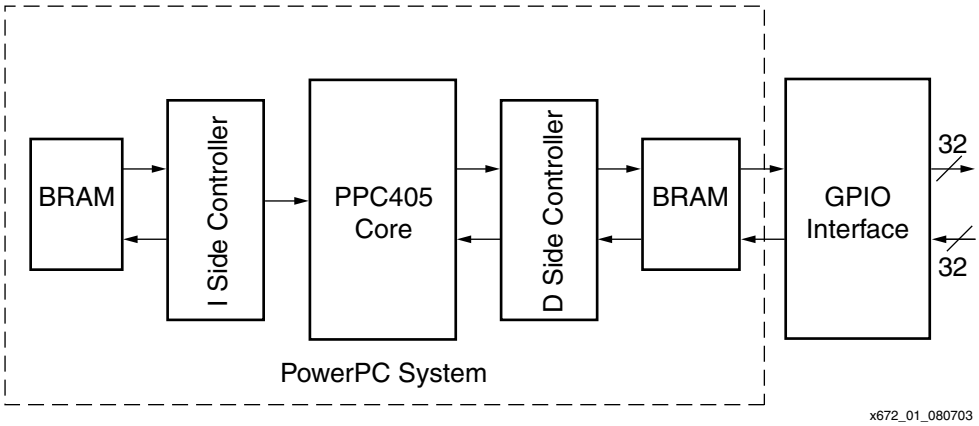

Figure 1: **UltraController Block Diagram**

Introduction

Technical Advantages

The embedded PowerPC (PPC405) processor in Virtex-II Pro FPGAs gives the UltraController solution all of the advantages of a system-on-a-chip processor.

- Tight connectivity to fabric reducing latency
- Reduced board area
- Elimination of pins typically required when connecting to an external processor
- Associated power reduction both for the processor, and the switching current required with an external processor interface.

The simple I/O interface in the UltraController solution eliminates the need for fabric instantiated busses and thus utilizes less than 50 logic cells. This makes a PowerPC solution nearly zero cost for simple microcontroller applications.

The UltraController solution allows designers to make a trade-off between slow logic implemented in fabric versus implementing the same control function in the PowerPC processor. If unused block RAM is available, the logic functions implemented by the PowerPC processor are free. A reference design created in both fabric and on the PowerPC processor showed that 2,400 logic cells equated to 16k Bytes of code/data storage. Trade-offs of this type can save designers from moving to a larger part by transferring slower fabric functions into the PowerPC processor.

Technical Characteristics and I/O Interface

An UltraController solution utilizing the Power PC Processor includes:

- 32, 32-bit general purpose registers
- 64-bit arithmetic
- 64-bit instruction-side on-chip memory (OCM) block RAM interface
- 32-bit data-side on-chip memory block RAM interface
- 64-bit time-base
- Support for debug through a JTAG port, with available graphical debuggers

There are several advantages to placing code and data in an OCM. The OCM provides a direct connection to the PowerPC execution unit eliminating the need for an interface bus. Additionally, using the OCM memory interface guarantees a fixed latency of execution for a higher level of determinism.

As shown in Figure 2, the UltraController interface includes:

- 32-bit input port
- 32-bit output port
- Reset input
- Clock input

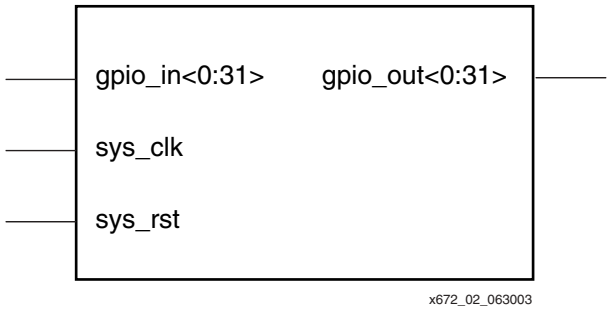

Figure 2: **UltraController Interface Block Diagram**

Using external multiplexers, a user can create larger input and output configurations.

The UltraController interface function is implemented via I/O operations utilizing port B of the data-side block RAM. The UltraController design includes a two-state controller to sequentially read a dedicated RAM location and latch the information for fabric access in a 32-bit wide output register; or to read system input data from a 32-bit wide bus into a dedicated RAM location. User C code is created to write or read from the dedicated I/O locations. A side benefit of the RAM interface is the ability to read back written data. This allows stateless output operations to be created via a read, modify selected bits, and write-back operation. As demonstrated in the reference design, this is easily done in a few lines of C code.

The UltraController reference solution is delivered in two configurations with source examples and top-level modules in both VHDL and Verilog. Separate downloadable files accomodate a single processor in a dual processor Virtex-II Pro device. The implementations are shown in Table 1.

Table 1: **UltraController Module Names and Memory Characteristics**

Module Name	Instruction-Side Memory	Data-Side Memory
uc_4i_4d	8 KB, four block RAMs	8 KB, four block RAMs
uc_8i_8d	16 KB, eight block RAMs	16 KB, eight block RAMs

Implementing an UltraController Design

The UltraController reference design consists of a completed and tested EDK PowerPC design. The HDL reference implementation provided with this application note includes code examples in Verilog and VHDL. The PowerPC software reference design example is in C code. The reference code implements a simple melody repeat game. This implementation provides the opportunity to demonstrate simple, but common I/O functions necessary in a wide variety of designs. Contained in the *simon.c* reference design are module examples of LED and LCD display drivers, square-wave sound generation, push-button reading, and a software state machine. The block diagram in Figure 3 shows device signal connections to the UltraController interface for the reference design.

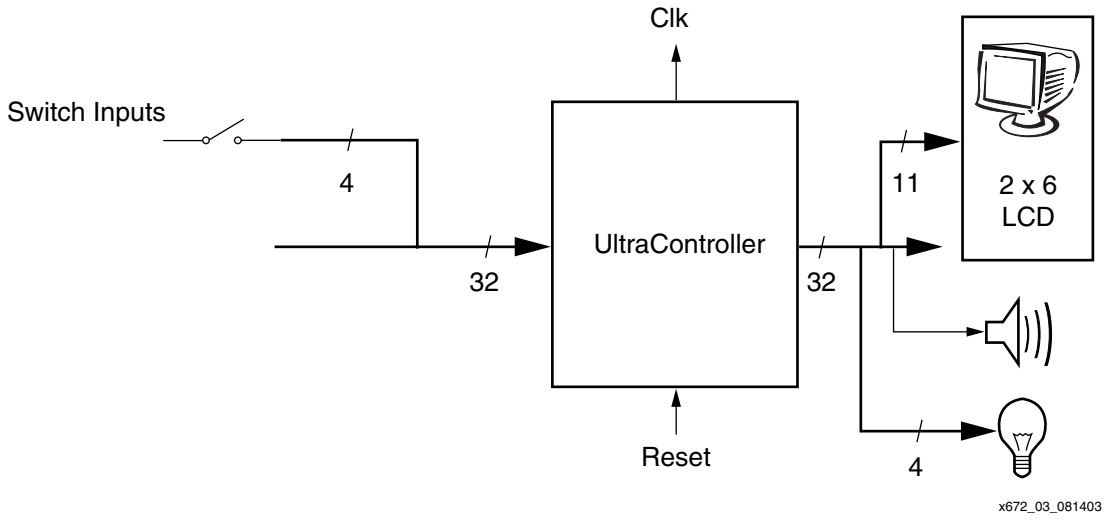

Figure 3: **UltraController Reference Design**

To demonstrate the flexibility of the UltraController and its interfaces, a frequency counter was created utilizing the UltraController reference design. The frequency-counter implementation is not covered in ths application note. The design accepts a 1 Hz to 200 MHz signal input, simultaneously measures its frequency and period, computes the frequency, and then automatically scales and formats the output for display on an LCD. This design requires reading two 32-bit counters and was accomplished by multiplexing the two 32-bit inputs into the 32-bit input port of the UltraController interface. The diagram in Figure 4 shows the UltraController frequency counter implementation. Design and debug required less than two days. The implementation time breakdown follows:

- ½ day for event and time base counter design, simulation, and debug.
- 1 day for software design, implementation, and debug
- ½ day for system integration and debug

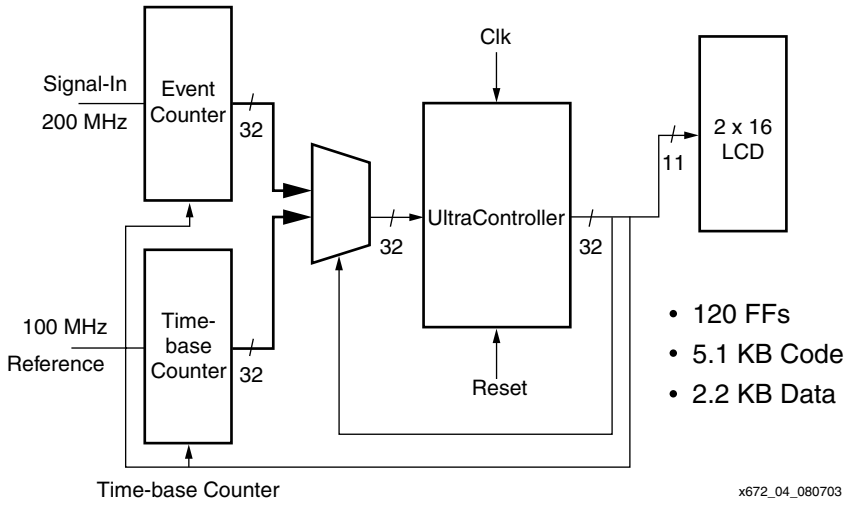

Figure 4: **UltraController Frequency Counter**

Using the UltraController Reference Design

Reference Design Environment

The UltraController reference design requires the following Xilinx software products to build a system:

- ISE - Version 5.2.03i (Service Pack 3)
- EDK - Version 5.2i / EDK 3.2.2 Build EDK_Cm.22 (service pack 2)

The reference design can be demonstrated utilizing the following hardware:

- Memec Design (Insight) Virtex-II Pro P4 Development Board
- Xilinx Parallel Cable IV for bit-stream download and code debug

Other information on building the UltraController reference design using the above products, any update information, and board configuration data is found in the ReadMe/QuickStart file included with the reference design.

HDL Implementation and FPGA Fabric Connection

The UltraController reference design is delivered with a sample top level module: *UltraController_Demo*. There is only one module instantiated by the *UltraController_Demo* module: uc_4i_4d (or uc_8i_8d).

The uc_4i_4d (or uc_8i_8d) module contains the PowerPC core, JTAG, reset controller, block RAM interfaces, block RAM, and the GPIO interface module. The UltraController_Demo module contains the following example Verilog code. Both Verilog and VHDL are available in the reference design file. The Verilog example follows:

```
module UltraController_Demo(
          sys_clk,
          nsys_rst,
          gpio_in,
          gpio_out
          );
   input         sys_clk;
   input         nsys_rst;
   input [29:31] gpio_in;   // 3 inputs for Push Buttons
   output [16:31] gpio_out; // 15 outputs for LCD and LEDs

   // Instantiate BUFGP on Input Clock
   wire uc_sys_clk
   BUFGP U1 (.I(sys_clk), .O(uc_sys_clk));

   // Instantiate the UltraController Core
   system uc_4i_4d (
          .sys_rst    (~nsys_rst | RST),  // I - Active High reset
          .sys_clk    (uc_sys_clk),       // I
          .gpio_out   (gpio_out),         // O
          .gpio_in    (gpio_in)           // I
          );
endmodule
```

In this example, the external clock after buffering, and GPIO input and outputs are passed directly through to the UltraController module. The external negative reset (active Low) is inverted and passed through to the UltraController module.

With new designs, connect a reset line and clock to enable the UltraController. Then connect gpio_out and gpio_in to the appropriate pins in the target design.

UltraController Port Connections

Table 2 has the definitions for the UltraController module port connections.

Table 2: **UltraController Module Port connections**

Port	I/O	Description
sys_rst	I	UltraController module reset (active High). Resets all logic within the UltraController module, including the PowerPC core and the GPIO interface. Does NOT reset the PowerPC memory.
sys_clk	I	CPU clock – Clock signal for the PowerPC core. See Table 5 for maximum allowable frequencies
gpio_out[0:31]	O – 32 bit bus	UltraController 32 bit output port. Data written by the PowerPC processor to the corresponding block RAM memory address is latched in an output register on every-other rising edge of sys_clk. (gpio_in reads alternate with gpio_out writes)
gpio_in[0:31]	I – 32 bit bus	UltraController 32 bit input port. Data is read by the GPIO state machine and written to block RAM on every-other rising edge of sys_clk. (gpio_in reads alternate with gpio_out writes)

GPIO Interface Timing

The UltraController GPIO interface utilizes a two-state state-machine to transfer data to and from a block RAM interface on alternating rising clock edges. First, on a rising clock edge, fabric supplied data to gpio_in is written to a fixed block RAM location. Subsequently this data can be read by a user defined software routine. On the next rising clock edge, data located in a fixed block RAM location is latched into a register and made available to the FPGA fabric. The timing diagram in Figure 5 summarizes the GPIO operation.

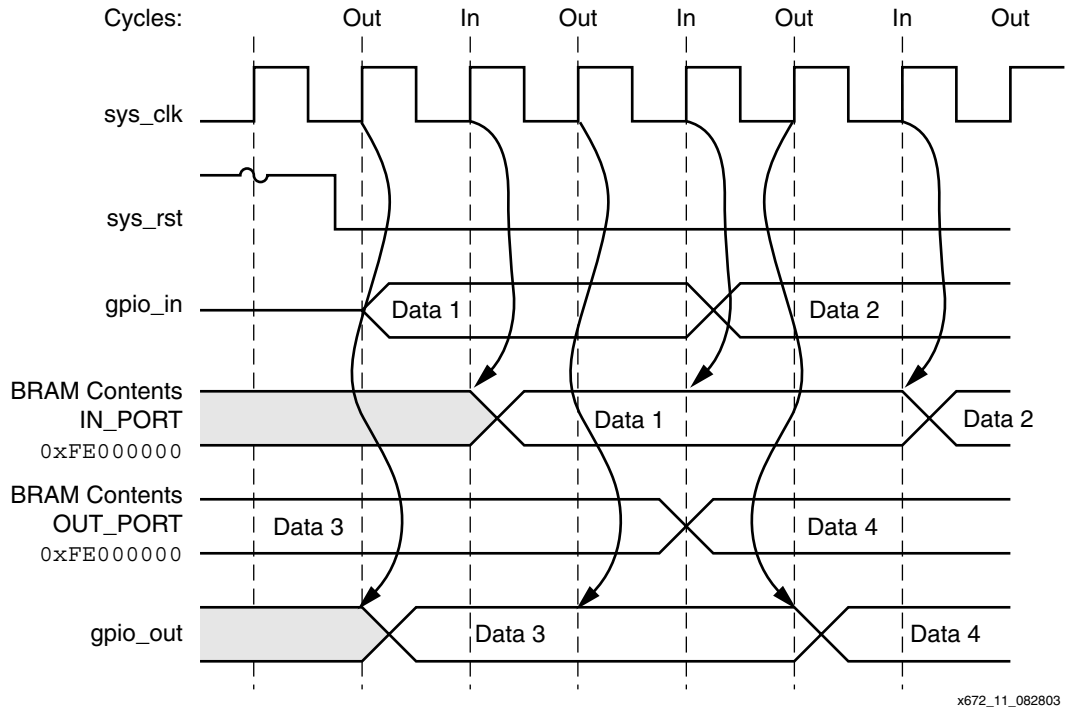

Figure 5: **GPIO Timing**

ISE Design Files Provided

Table 3 shows the source modules for the ISE project:

Table 3: **ISE Source Modules Included in UltraController Reference Solution**

File Name	Description
..\projvav\uc_4i_4d.v or ..\projvav\uc_4i_4d.vhd ..\projvav\uc_8i_8d.v or ..\projvav\uc_8i_8d.vhd	Top-level UltraController reference example module. A "4" indicates four block RAMs and an "8" indicates eight block RAMs.
..\hdl\system.v or ..\hdl\system.vhd	Platform Studio generated UltraController module including the PowerPC core. Must be added as a sub-module of the top-level module.
..\projvav\uc_4i_4d.ucf ..\projvav\uc_8i_8d.ucf	User constraint file for the UltraController implementation.
..\ppc405_1\code\executable.elf	The compiled object file data to be loaded into block RAM. Must be added as a sub-module of the top-level module.
..\implementation\system_stub.bmm	Block RAM definitions created by Platform Studio. Must be added as a sub-module of the top-level module.
..\projvav\tf_uc.tf	Reference design test fixture to be used for simulation. See the UltraController tutorial on the UltraController web site (www.xilinx.com/processor) for test fixture usage.

The files listed in Table 4 are generated by Platform Studio in building the UltraController system. They are delivered as part of the .zip file to enable an ISE build without regenerating netlists in Platform Studio.

Table 4: **Additional ISE Modules Included in the UltraController Reference Solution**

File Name	Description
..\implementation*.ngc	Generated netlists for the PowerPC block and associated blocks
..\hdl*.v ..\hdl*.vhd	Generated Verilog or VHDL wrappers for the PowerPC block and associated blocks

UltraController System Performance

For design implementation simplicity and reduced power consumption an UltraController implementation using the PowerPC core and OCM running at the same speed (1:1) yields the maximum system frequency and performance (Table 5).

Table 5: **UltraController Performance**

Speed Grade	-5		-6		-7	
UltraController Version	MHz	DMIPs	MHz	DMIPs	MHz	DMIPs
4 x 4	175	193	195	215	200	220
8 x 8	160	176	180	198	195	215

While highly dependent on coding style, raw input and output performance provide a starting point for determining if a microprocessor implementation will meet system requirements. Several tight input and output loops were tested to demonstrate system performance. The Code Example reads the previously output value, and then enters a tight loop to output the shifted and inverted values 1, 2, 4, and 8. The benchmark of this output loop is 20 million cycles-per-second (MCPS) or 1/10th of the processor clock speed.

Code Example

```
Port_value = *OUT_PORT &0xFFFF0FFF;
For (I=1; I < 16; I += i) // count 1, 2, 4, 8
{
  *OUT_PORT = port_value | (~I << 12);
  }
```

Input/Output Performance

The next three tables list the output benchmark and input/output performance. Table 6 was generated using the Code Example. Table 7 uses the Xio macro described in the Reading and Writing to the GPIO Interface using C Coding section. Table 8 is the results from a "processing loop" where data is read, a decision is made based upon the input, and a value is output dependent upon the input.

Table 6: **Benchmark -- Output Loop (Read Outside, Modify and Write Inside)**

Speed Grade	-5		-6		-7	
UltraController Version	MHz	I/O MCPS	MHz	I/O MCPS	MHz	I/O MCPS
4 x 4	175	17.5	195	19.5	200	20
8 x 8	160	16	180	18	195	19.5

Table 7: **Benchmark -- Output Loop with Xio_x Macro (Read Outside, Modify and Write Inside)**

Speed Grade	-5		-6		-7	
UltraController Version	MHz	I/O MCPS	MHz	I/O MCPS	MHz	I/O MCPS
4 x 4	175	12.5	195	13.9	200	14.2
8 x 8	160	11.4	180	12.8	195	13.9

Table 8: **Benchmark -- Processing Loop (Read Input Inside, Decision, Modify and Write Inside)**

Speed Grade	-5		-6		-7	
UltraController Version	MHz	I/O MCPS	MHz	I/O MCPS	MHz	I/O MCPS
4 x 4	175	7	195	7.8	200	8
8 x 8	160	6.4	180	7.2	195	7.8

Mapping Between Fabric GPIO Pins and Software Ports

The reference design is targeted for the Insight (Memec) Virtex-II Pro (P4) development board. The following mapping exists between the calling C code, the hardware interface, and the Virtex-II Pro (XC2VP4 - FG456) device pins:

```
Description  Code Bit    GPIO Output Bit       Device Pin
LCD D0         0                  31              D7
LCD D1         1                  30              F9
LCD D2         2                  29              D5
LCD D3         3                  28              D6
LCD D4         4                  27              C7
LCD D5         5                  26              D8
LCD D6         6                  25              C8
LCD D7         7                  24              E8
LCD RS         8                  23              E6
LCD EN         9                  22              E7
--            10                  21
Sound         11                  20              U5
LED 1         12                  19              V8
LED 2         13                  18              W6
LED 3         14                  17              U10
LED 4         15                  16              V10

Description   Code Bit    GPIO Input Bit        Device Pin
SW 1           0                  31              V7
SW 2           1                  30              W5
SW 3           2                  29              AA12
```

For the C code, bit 0 is considered to be the least significant bit. However, for the PowerPC core and its connected devices, bit 31 is the least significant bit. The C code interface routines shown in the next section manipulate the mapping between the two systems of numbering bits.

Software Implementation

The port, memory, and processor speed values for the delivered software reference solution are listed in Table 9.

Table 9: **Port, Memory, and Processor Speed Characteristics**

Description	uc_4i_4d	uc_8i_8d	How To Change
32-Bit output port address (gpio_out)	0xFE000000	0xFE000000	not recommended — change linker script
32-Bit input port address (gpio_in)	0xFE000004	0xFE000004	not recommended — change linker script
Instruction Memory Start Address	0xFFFFE000	0xFFFFC000	See Appendix A for changing memory size
Instruction Memory End Address (including boot vector)	0xFFFFFFFF	0xFFFFFFFF	not recommended — last address contains boot vector
Data Memory Start Address	0xFE000008	0xFE000008	Not recommended — requires changing I/O port addresses. Change memory end address to change size.
Data Memory End Address	0xFE001FFF	0xFE003FFF	See Appendix A for changing memory size
Stack Size	1024 bytes	2048 bytes	Platform Studio Menu: Options/Compiler Options Tab: Details Parameter: Stack Size
Heap Size	4 bytes	4 bytes	Platform Studio Menu: Options/Compiler Options Tab: Details Parameter: Heap Size
Processor Speed (used for usleep)	100 MHz	100 MHz	Speed should match implemented PowerPC frequency. Platform Studio Menu: Options/Compiler Options Tab: Environment Parameter: Core Clk Freq

Typically the GPIO addresses are not used since the addresses are passed via the linker script to the GPIO routine. Simply reference the memory locations pointed to by OUT_PORT and IN_PORT (see examples in Reading and Writing to the GPIO Interface using C Coding).

To increase or decrease the memory sizes, use the information in Appendix A.

The stacks are set at 1024 and 2084 bytes for the uc_4i_4d and uc_8i_8d designs respectively. Actual implementation dictates the amount of stack needed. It is always better to start with too much stack space and then adjust downwards upon completion of the design.

A Heap is utilized when a C program dynamically allocates memory via a specific function called malloc. For small microcontroller applications this is not done. The UltraController heap is set to four bytes.

When using a timing function called "usleep", the software system must be notified of the processor speed. This is done via the Core Clk Freq (aka: processor speed) setting as noted in Table 9.

Reading and Writing to the GPIO Interface using C Coding

Using the supplied GPIO software interface module: *gpio.c*, the GPIO input port appears as the variable IN_PORT. The GPIO output port appears as the variable OUT_PORT. Using C code programming conventions, these memory locations are referenced as pointers as in the following examples:

```
*OUT_PORT = data_to_be_output;
data_to_be_read = *IN_PORT;
```

However, to assure proper synchronization of data read and written to the I/O interfaces via the GPIO interface utilizing block RAM, it is strongly suggested that the user utilize the Xilinx in-line macros to force memory synchronization. Implement the previous examples as:

```
XIo_Out32( (XIo_Address)OUT_PORT, data_to_be_output);
data_to_be_read = XIo_In32( (XIo_Address)IN_PORT );
```

The module utilizing the Xilinx in-line I/O macros must contain the following statements to force inclusion of the Xilinx I/O library and to force instantiation of the function calls as in-line macros:

```
#define USE_IO_MACROS
#include "xio.h"
```

Since output writes are written through the block RAM, all data written is also saved in the associated RAM location. This enables the ability to read data previously written to the output port.

The next example reads the block RAM location containing the data previously written, masks out the data area to be changed (LCD MASK), ORs in the "data_to_be_output", and then writes the data to the output port.

```
tmp32 = XIo_In32( (XIo_Address)OUT_PORT );
XIo_Out32( (XIo_Address)OUT_PORT, tmp32 & LCD_MASK | data) );
```

Finally, the data being accessed can not reside in the low order bits of the port. Thus, the desired output data must be shifted left by an appropriate number of bits to be properly positioned (LCD_OFFSET in the following example). Utilizing this technique, the LCD output routine is:

```
void GPIO_writeLCD( Xuint32 data )
{
  Xuint32 tmp32;
  tmp32 = XIo_In32( (XIo_Address)OUT_PORT ); // Read  Output Port
  XIo_Out32( (XIo_Address)OUT_PORT, tmp32 & LCD_MASK | (data << LCD_OFFSET)
  );
  }
```

Where the data read from the output port is temporarily held in the unsigned 32-bit integer (Xunit) variable tmp32.

In a similar manner, a read of three push buttons will consist of calling the following function:

```
Xuint32 GPIO_readSwitch(void)
{
  Xuint32 tmp32;
  tmp32 = XIo_In32( (XIo_Address)IN_PORT ); // Read Input Port
  return((~tmp32 & SWITCH_MASK) >> SWITCH_OFFSET ); // Return switch bits
  }
```

In this example, the input port is read and assigned to the temporary variable tmp32, the three bits of interest are masked out by SWITCH_MASK, the result is right shifted by SWITCH_OFFSET to align the least significant bit with bit zero, and the data is inverted to show a High (1) bit interpreted as a button push.

The following code implements a complete I/O module for the data word output to a two line LCD and reading three push buttons:

```
#define USE_IO_MACROS
#include "xio.h"

Xuint32 OUT_PORT[1] __attribute__ ((section(".io_reg")))={ 0 };//Address 0
Xuint32 IN_PORT[1]  __attribute__ ((section(".io_reg")))={ 0 };//Address 4

#define LCD_MASK   0xFFFFFC00
#define LCD_OFFSET 0
#define LCD_MASK   0xFFFFFC00
#define LCD_OFFSET 0
#define SWITCH_MASK   0x00000007
#define SWITCH_OFFSET 0

// Write LCD
void GPIO_writeLCD( Xuint32 data )
{
  Xuint32 tmp32;
  tmp32 = XIo_In32( (XIo_Address)OUT_PORT ); // Read Output Port
  XIo_Out32( (XIo_Address)OUT_PORT, tmp32 & LCD_MASK | (data << LCD_OFFSET)
  );
  }
// Read Switches -- invert value (Return a 1 = pressed)
Xuint32 GPIO_readSwitch(void)
{
  Xuint32 tmp32;
  tmp32 = XIo_In32( (XIo_Address)IN_PORT ); // Read Input Port
  return((~tmp32 & SWITCH_MASK) >> SWITCH_OFFSET ); // Return switch bits
  }
```

As shown in shown in the previous code, the following statements should be copied to any new user I/O routine interfacing with the GPIO interface:

```
Xuint32 OUT_PORT[1] __attribute__ ((section(".io_reg")))={ 0 };//Address 0
Xuint32 IN_PORT[1]  __attribute__ ((section(".io_reg")))={ 0 };//Address 4
```

These statements create a mapping between the Linker-Script declaration of the input and output memory locations and the memory pointers (OUT_PORT and IN_PORT) referenced by the C code interface.

Software Design Files Provided

The sources in Table 10 are in the Platform Studio project directory.

Table 10: **Platform Source Modules Included in UltraController Reference Solution**

File Name	Description
simon.c	Reference design example. Contains the main() and various sub-functions. Primary body of code implements a simple memory melody repeat game.
lcd.c	LCD driver. Contains LCD initialization function, low level LCD output function, and function to write two strings; one each to the upper and lower lines of the LCD display.
gpio.c	32-bit input and 32-bit output functions. Collection of modules to output to and read from target devices on the reference design board. The LCD output and switch input functions as described in the Reading and Writing to the GPIO Interface using C Coding section. Write to LEDs and write to speaker pin functions are also included.
linker_script	Text file describing how to link compiled sources together for a target system. See Appendix A for changing the linker script to accommodate memory size changes.

Reference Design

The reference design, in VHDL and Verilog is located on the Xilinx web site at:

www.xilinx.com/ultracontroller

For additional information, there are tutorials for the UltraController solution. These tutorials cover:

- Changing software code
- Simulation in ModelSim SE or PE
- Debugging fabric logic with ChipScope Pro tools
- Debugging software code with a GNU project debugger (GDB).

Conclusion

For simple control, system monitoring, or I/O functions, the UltraController reference design is an ideal solution. By using OCM interfaces, and a GPIO connected to the second port on the data-side OCM block RAM, the need for processor busses is eliminated. Thus, the design utilizes less than 50 logic cells.

UltraController solutions are especially attractive when the design can be partitioned into a fast portion ("nanosecond logic") implemented in the FPGA fabric, and a slow portion ("millisecond logic") implemented in the PowerPC processor. In these situations the UltraController solution saves valuable FPGA fabric resources by trading logic cells, for code and data RAMs. These savings enable utilization of smaller devices and help to avoid design creep into larger devices. This ultimately leads to overall cost savings.

By utilizing the predefined and tested UltraController solution, several design process challenges are removed, and the design time for an embedded controller solution is shortened.

Appendix A

Changing Memory Size

The UltraController reference design is delivered as a series of predefined configurations. This section outlines the steps to customize the amount of instruction or data memory. The minimum memory size for instruction memory is 4 Kb or two block RAMs. The minimum memory size for data memory is 8 Kb or four block RAMs. Memory sizes must be multiples of two: 4 Kb, 8 Kb, 16 Kb, and so on.

Changing Instruction-Side or Data-Side Block Memory Size

For maximum performance, placement of the instruction and data block memories is critical. Xilinx recommends constraining the location of the memory in the .ucf file or through the use of the Xilinx Floorplanner tool for placement of the memory blocks. Both of the provided examples can be used for placement of the memory blocks (Figure 6). For designs utilizing less than eight instruction or eight data memory blocks, start with the uc_8i_8d design module and eliminate memory blocks as appropriate. Similarly, for larger designs, start with the uc_8i_8d design module and add memory blocks as appropriate.

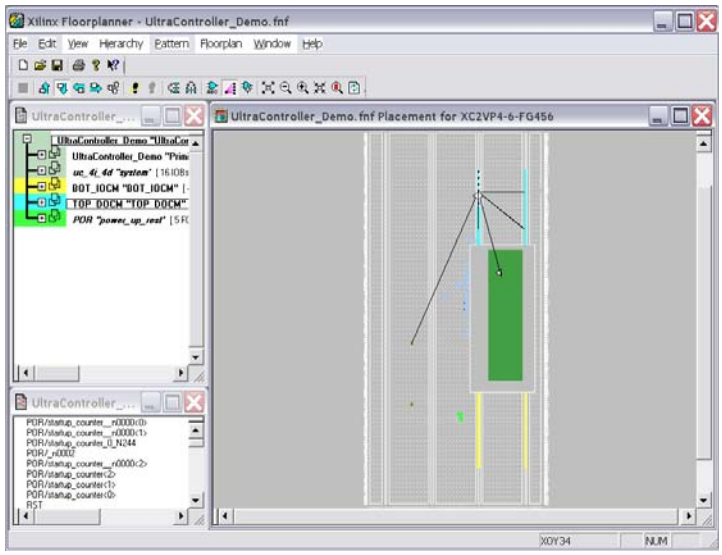

Figure 6: **Floorplanner Examples Showing Placement of Block Memory for uc_4i_4d**

Changing Instruction-Side Block Memory Size

Inside the Platform Studio software:

1. Change the program start address under Options / Compiler Options - Details Tab. The screen capture in Figure 7 illustrates the dialog. The current address is `0xFFFFE000`.

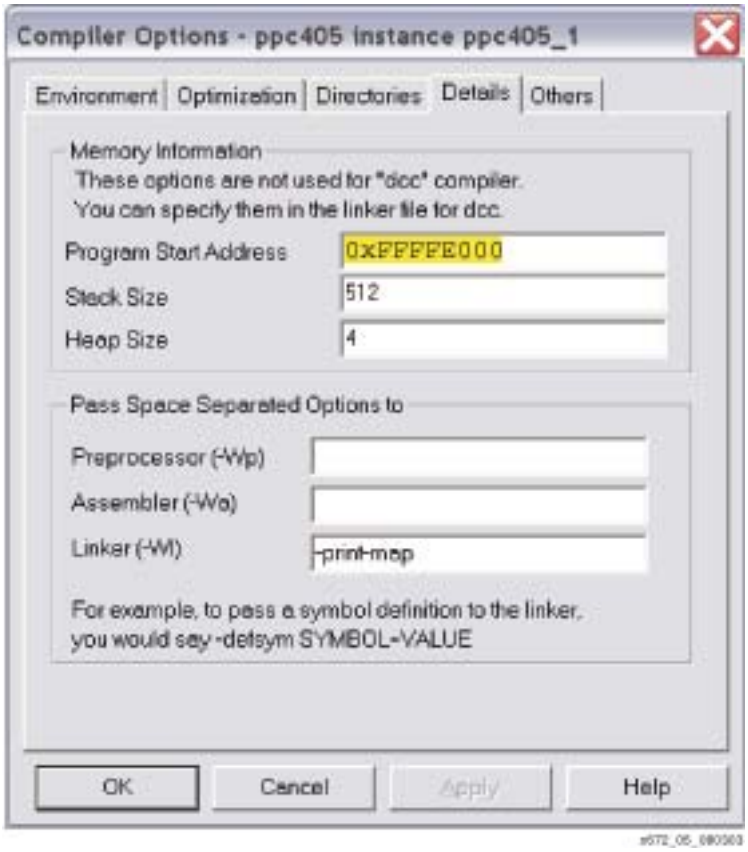

Figure 7: **Program Start Address**

2. Change the hardware memory starting address (*isbram*) under Project / Add/ Edit Cores. The screen capture in Figure 8 illustrates the dialog. The current address is `0xFFFFE000`.

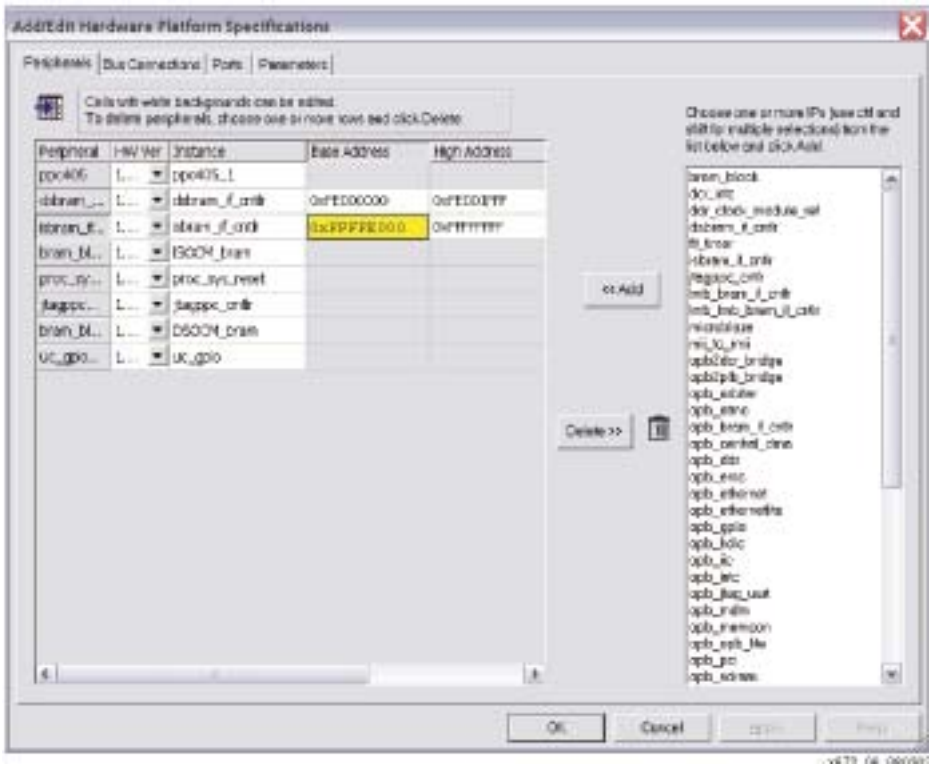

Figure 8: **Instruction-Side Memory Address**

3. Under the System Tab, double-click on Linker Script to edit the linker script. In the defines section change IOCM_SIZE to the new memory size. In the MEMORY section change the *isocm* ORIGIN to the new memory starting address and change the LENGTH appropriately. The current _IOCM_SIZE is 8k. The current address is `0xFFFFE000`. The current LENGTH is 8k (bytes) less four bytes for the boot vector.

```
/* Define sizes of memory and I/O space */
_IOCM_SIZE = 8k;
_DOCM_SIZE = 8k;
_IO_REG_SIZE = 8; /* 8 bytes for GPIO */
MEMORY {
isocm          : ORIGIN = 0xFFFFE000,   LENGTH = 8k - 4
bootm          : ORIGIN = 0xFFFFFFFC,   LENGTH = 4
io_reg_mem     : ORIGIN = 0xFE000000,   LENGTH = 8
dsocm          : ORIGIN = 0xFE000008,   LENGTH = 8k - 8
 }
```

4. Edit the file xmd.ini located in the Platform Studio project directory. Change the start address (*isocmstartadr*) and memory size respectively. In the following example, the current address is `0xFFFFE000` and the current size is `0x2000` or 8k. This command line notifies XMD where the instruction-side memory is located.

```
ppcconnect -debugdevice isocmstartadr 0xFFFFE000 isocmsize 8192
     isocmdcrstartadr 0x18 dcrstartadr 0x78002000
```

Changing Data-Side Block Memory Size

The GPIO interface maps to the lowest eight bytes of the data-side block memory space. To minimize effort do not modify the data-side starting address since it is easiest to change the memory size by changing end address. The following changes are made from inside of the Platform Studio:

1. Change the hardware memory ending address (*dsbram*) under Project / Add/ Edit Cores. The screen capture in Figure 9 illustrates the dialog. The current address is `0xFE001FFF`.

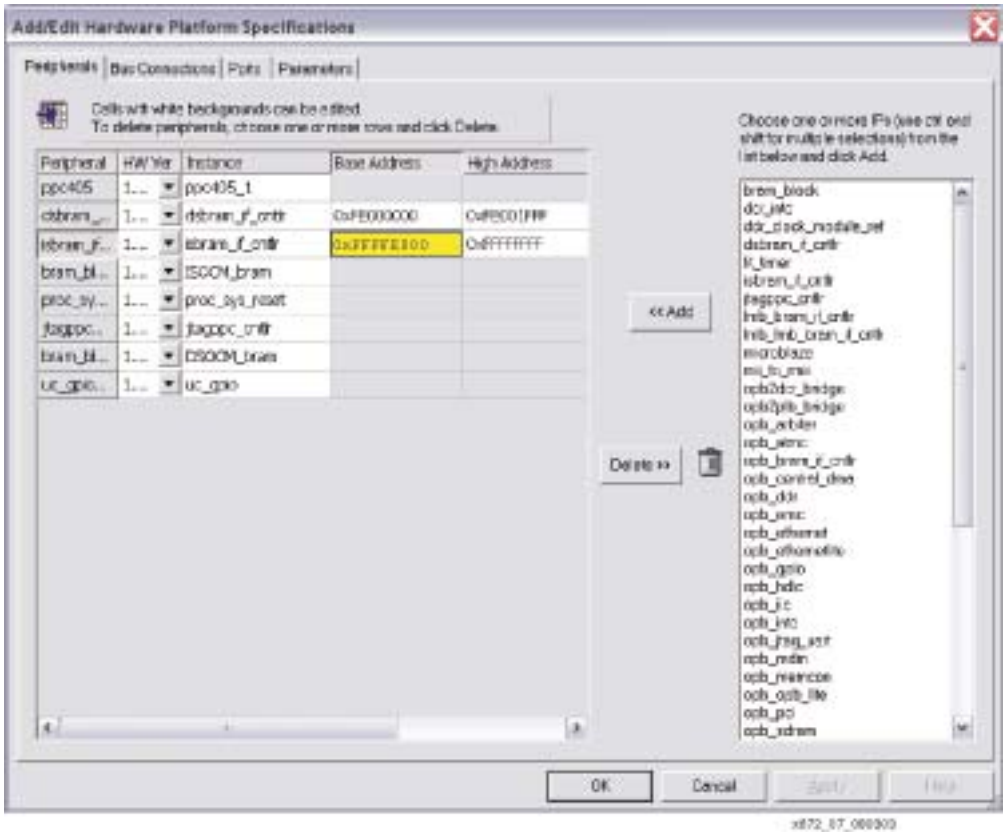

Figure 9: **Data-Side Memory Address**

2. Under the System Tab, double-click on Linker Script to edit the linker script. In the defines section change _DOCM_SIZE to the new memory size. In the MEMORY section change the *dsocm* LENGTH to the new memory size. The current _DOCM_SIZE is 8k. The current LENGTH is 8k (bytes) less eight bytes for the GPIO interface.

```
/* Define sizes of memory and I/O space */
_IOCM_SIZE = 8k;
_DOCM_SIZE = 8k;
_IO_REG_SIZE = 8; /* 8 bytes for GPIO */
MEMORY {

MEMORY {
   isocm          : ORIGIN = 0xFFFFE000,    LENGTH = 8k - 4
   bootm          : ORIGIN = 0xFFFFFFFC,    LENGTH = 4
   io_reg_mem     : ORIGIN = 0xFE000000,    LENGTH = 8
   dsocm          : ORIGIN = 0xFE000008,    LENGTH = 8k - 8
    }
```

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/02/03	1.0	Initial Xilinx release.