Pico Computing Inc.

Guide To Samples and Wizards

Version 5.0.0.3. Apr 16th, 2010.

Pico Computing, Inc.
150 Nickerson, Suite 311
Seattle, WA, 98109-1634
(206) 283-2178
www.picocomputing.com

1 Overview

This document describes the basic sample projects and wizards distributed on the Installation CD or the Web installer.

Samples:
The samples are complete, ready to compile projects that you can use, or adapt to your own use. The samples are installed in **c:\pico\samples**.

Wizards:
This wizards are templates of projects that are used by PicoUtil to generate a customized version of a project. You may find this more convenient that copying and patching a sample project. The wizards are installed in in **c:\pico\wizards**.

The samples installed from the CD-rom or from the Web installer are a very limited set. More samples and wizards are published on the Pico Computing website(www.picoComputing.com)
Other manuals in this help library can be located at GuideToDocumentation.pdf

NOTE: This link will access the pdf from the PicoComputing.com Website .

2 Synthesizing a Project using Makefiles

Many of the sample projects use the Xilinx command line tools **xst, ngdbuild, map, par, bitgen,** and **data2mem** directly. These projects are controled by a file with the extension **.fwproj.** The fwproj files defines options for the make process. Fort example, in the directory **c:\pico\sampes\PicoBus_counter\firmware** the file **e14FX20_Picobus_counter.fwproj** comprises:

```
PICO_MODEL=E14FX20
EXPORT_BIT_FILE=E14FX20_PicoBus_counter.bit
USER_MODULE_NAME=PicoBus_counter
ENABLE_USER_PICOBUS=y
USER_VERILOG_FILES=PicoBus_counter.v
```

The command:

build e14FX20_Picobus_counter

Will build the firmware bit file for this example. The file build.bat invokes **build-pico-fw-project** e14FX20_Picobus_counter and captures the output in e14FX20_Picobus_counter.log. build-pico-fw-project in turn invokes the make utility in c:\pico\msys\1.0\bin to create the output bit file.

Refer to Load Using_the_Makefile.pdf.

3 Synthesizing a Project from Picoutil

The fwproj file mentioned in the previous section can also be managed from Picoutil. Right click the bit file and select build. Picoutil searches for an appropriate fwproj file and monitors the output of the build process. Refer to Picoutil.pdf.

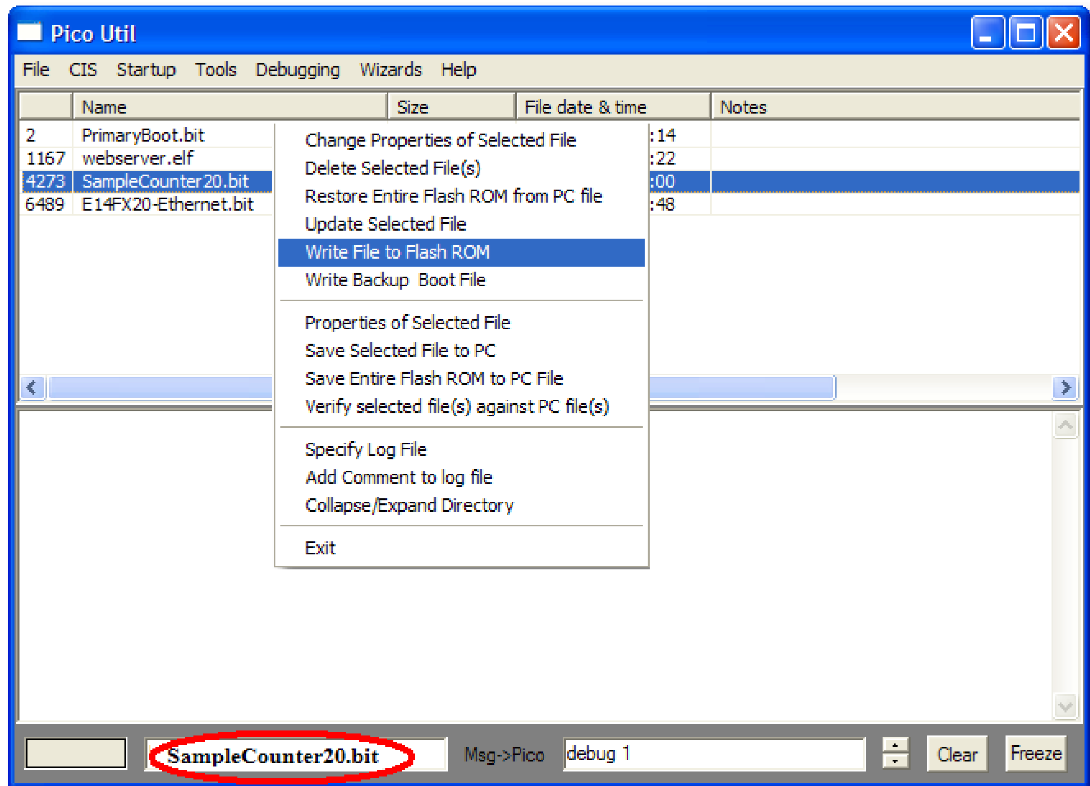
4 Loading the Bit file onto the Pico Card.

The new image **e14FX20_Picobus_counter.bit** can be loaded into the FPGA on the Pico Card using a number of strategies. We will describe here the strategy of writing the file to the flash ROM and then rebooting the FPGA with the new file. The tools PicoCommand.exe and PicoUtil.exe can be used for

©2010 Pico Computing, Inc.

this purpose. In PicoUtil:

- From the main menu press **File / Write File to Flash ROM**,
- Select the file **c:\pico\sample\E14_counter\firmware\e14FX20_Picobus_counter.bit**.
- Press **OK** on the file parameters screen.



To reboot the FPGA with **e14FX20_Picobus_counter.bit** select the file **e14FX20_Picobus_counter.bit** and press enter (or right click and reboot, or menu / startup / boot). The new image will sign on with a message similar to the following:

```
PicoUtil.exe: 4.0.2.1 Release: Aug  1 2006 15:19:10.
Pico.sys: 4.0.2.1 Release: Aug  1 2006 12:04:54.
Pico.dll: 4.0.2.1. Release: Aug  1 2006 15:17:47
Firmware: (4vfx20ff672) 4.0.2.1: Jul 14 2006 13:25
PPC: Pico Monitor 4.0.2.1 Jul 14 2006 13:55:25.
```

The lower dashboard will display **e14FX20_Picobus_counter.bit.** You are now running the new e14FX20_Picobus_counter.bit This image is essentially the same as PrimaryBoot.bit with the addition of the counters we have built in.

Refer to Picoutil.pdf.

5 Firmware Samples

A variety of sample projects are installed in the folder **c:\pico\samples**. These samples will often contain firmware projects and software projects to exercise the firmware. The following projects are typically installed:

- **E12_counter.** This project is pair of counters which are accessible from either the PC host (over

©2010 Pico Computing, Inc.

the PCMCIA bus) or from the PPC. Sample software programs can be used to read these counters. This project is described in more detail under File SampleCounter.
- **E14_counter.** This project is pair of counters which are accessible from either the PC host (over CardBus) or from the PPC. Sample software programs can be used to read these counters. This project is described in more detail under File SampleCounter.
- **E14_BMcounter.** This project is a single counter accessible using DMA or single word access from the PC host using the PicoBus (as abstraction of the CardBus. A sample software program illustrates how to access this counter. This project is described in more detail under Bus Mastering Sample.
- **E14_BMram.** This project illustrates DMA access to a BRAM buffer on the Pico Card. A sample software program illustrates how to access this counter. This project is described in more detail under Bus Mastering Sample.
- **stream_checksum.** This project uses the streams model to transfer data over DMA. A sample software project illustrates how to use the stream.

5.1 Counter Visible from PC and PPC

SampleCounter is an example of two counters attached to the PLB (Processor Local Bus) of the PPC and the bus of the PC host respectively. Each counter increments when it is read. The host counter is accessed from the host side of the interface using a *port address*, whereas the PPC counter is accessed from the PPC using *memory mapped IO*.

SampleCounter is a low speed interface. The cardbus interface will only support a maximum of 10 Mbytes /second using the access mode of SampleCounter. This is adequate for many applications in which the FPGA is doing any significant computations.

SampleCounter is installed in the folder **c:\pico\samples\E12_counter** or **c:\pico\samples\E14_counter** and contains the following subdirectories:

- **.\firmware**. This folder contains two source files: **SampleCounter.v** is a Verilog file containing the logic to support the two counters cited above; **ppcsystem_usermodule.includev** is an instantiation of the module which will be included by \src\ppcsystem.v during a system build.
- **.\readPPCcounter.** This is a PPC program which can be used to read the memory mapped counter on the PPC side.
- **.\readHostCounter.** This is a PC program which can be used to read the port addressed counter on the host side.

SampleCounter will also make reference to the following files and directories:

- **c:\pico\firmware\buildproject.bat.** This is the fundamental file used to build a FPGA file (.bit file).
- **c:\pico\firmware\pico14\src**. These are the fundamental files used to build the PicoE14 or PicoE12 firmware. They provide the interfaces to the card bus, the DDR-II ram on the Pico card, and several other peripherals.
- **c:\pico\firmware\picoxx\edk_ppc\hdl** and **c:\pico\firmware\pico14\edk_ppc\implementation.** These files constitute a black box generated by EDK and provide the PPC support. For most purposes, this black box need not be changed.

A build of the system can be started from the .bat file **build20.bat, build60.bat,** or **buildFX12.bat**. These .bat files call **c:\pico\firmware\buildproject.bat**. Refer to Synthesizing a Project for a

©2010 Pico Computing, Inc.

discussion of the steps invoked by buildproject.bat.

5.1.1 Sample Counter for Pico-E12

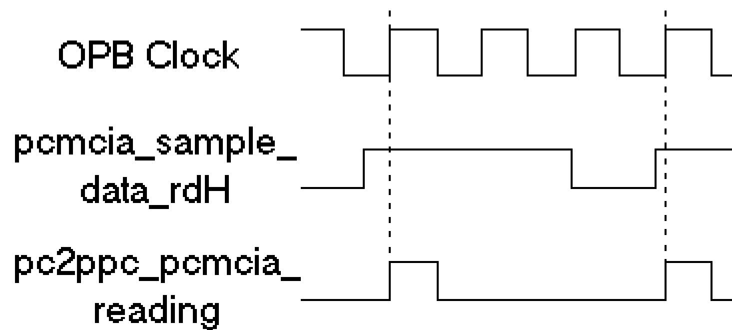
Pico E-12 sample counter:
There is no clock on the PCMCIA bus and the following logic uses the PPC clock to simulate such a clock.

```
86   //reading from pcmcia_counter.
87   always @(posedge OPB_Clk)
88      begin
89      if (pcmcia_sample_data_rdH & ~pc2ppc_pcmcia_reading)
90         begin
91         pc2ppc_pcmcia_reading    <= 1;
92         ppc2pc_fifo_read_inhibit <= 0;
93         pcmcia_step_counter      <= 1;
94         end
95      else
96      if (IORead & pc2ppc_pcmcia_reading)
97         ppc2pc_fifo_read_inhibit <= 1;
98      else
99      if (~IORead & pc2ppc_pcmcia_reading)
100        begin
101        pc2ppc_pcmcia_reading    <= 0;
102        ppc2pc_fifo_read_inhibit <= 0;
103        pcmcia_counter           <= pcmcia_counter + pcmcia_step_counter; //increment next cycle
104        pcmcia_step_counter      <= 0;
105        end
106     end
107
108  //PC read of pcmcia_counter
109  assign dout[C_PCMCIA_DWIDTH-1:0] =
110       (pcmcia_sample_data_rdL ? pcmcia_counter[C_PCMCIA_DWIDTH-1:0] : 0) |
111       (pcmcia_sample_data_rdH ? pcmcia_counter[31:C_PCMCIA_DWIDTH]  : 0);
112
113  //----------------- PPC computation ---------------------------------------------------
114  //ppc read of opb_counter. More conservative strategy for incrementing counter.
115  always @(posedge OPB_Clk)
116     begin
117     if (opb_sample_data_rd)
118        opb_step_counter <= 1;
119     else
120        begin
121        opb_counter      <= opb_counter + opb_step_counter; //increment next cycle.
122        opb_step_counter <= 0;
123        end
124     end
125
126  //ppc read of opb_counter.
127  assign Sl_DBus[C_OPB_DWIDTH-1:0] = opb_sample_data_rd ? opb_counter[C_OPB_DWIDTH-1:0] : 0;
```

The OPB counter is called **opb_counter** and the host counter is called **pcmcia_counter**.

- line 87: When the correct address is generated and **IORead** is high, **pcmcia_sample_data_rdH** is asserted.
- line 91: **pc2ppc_pcmcia_reading** is asserted in order to simulate a host clock that is in sync with the OPB clock.
- line 96: If **IORead** is asserted and the counter is already set in the read state, an inhibitor is asserted in order to prevent multiple reads. **pc2ppc_pcmcia_reading** can only be reasserted when **pcmcia_sample_data_rdH** is reset.

©2010 Pico Computing, Inc.



- line 103: At the end of the read cycle, the counter will be incremented.
- line 109: The value of the pcmcia_counter is ORed onto the PCMCIA bus. This bus is sampled well before the pcmcia_counter is incremented.
- line 115: When the proper address is generated on the OPB bus, **opb_sample_data_rd** is asserted. The counter is not incremented on this clock cycle but is incremented on the next leading edge of OPB_Clk.
- line 127: The value of opb_counter is ORed onto the OPB bus. This bus is sampled well before cb_counter is incremented. Logic in ppcsystem.v OR's all bus information onto the CF data bus.

5.1.2 Sample Counter for Pico-E14

Pico E-14 sample counter:
The Cardbus interface has two clocks: the CB clock and the OPB clock. This simplifies the Pico-E14 implementation substantially.

```
78
79  //----------------- PC computation ----------------------------------------------------
80  //reading from cb_counter.
81  always @(posedge CB_Clk)
82     begin
83     if (cb_sample_data_rd)
84        begin
85        CB_dataOut       <= cb_counter;
86        cb_counter       <= cb_counter + 1;
87        end
88     else
89        CB_dataOut       <= 0;
90     end
91
92
93  //----------------- PPC computation ---------------------------------------------------
94  //ppc read of opb_counter. More conservative strategy for incrementing counter.
95  always @(posedge OPB_Clk)
96     begin
97     if (opb_sample_data_rd)
98        opb_step_counter <= 1;
99     else
100       begin
101       opb_counter       <= opb_counter + opb_step_counter; //increment next cycle.
102       opb_step_counter <= 0;
103       end
104    end
105
106 //ppc read of opb_counter.
107 assign Sl_DBus[C_OPB_DWIDTH-1:0] = opb_sample_data_rd ? opb_counter[C_OPB_DWIDTH-1:0] : 0;
108
109 endmodule
```

The OPB counter is called **opb_counter** and the CB counter is called **cb_counter**.

©2010 Pico Computing, Inc.

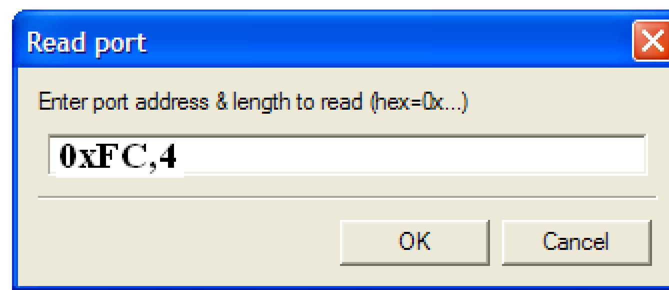
- line 83: When the proper address is generated on the CB bus, **cb_sample_data_rd** is asserted.
- line 85: The value of cb_counter is stored in the registered output
- line 86: On the next CB_clk the counter **cb_counter** is incremented and the register **cb_step_counter** cleared. This delayed incrementing strategy is designed to make sure the counter is not incremented while it is being sampled.
- line 89: The registered output is cleared under all other circumstances. Logic in ppcsystem.v OR's all cardbus peripherals onto the CB bus.
- line 97: When the proper address is generated on the OPB bus, **opb_sample_data_rd** is asserted. The counter not incremented on this clock cycle but is incremented on the next leading edge of OPB_Clk.
- Line 107: The value of opb_counter is ORed onto the OPB bus. This bus is sampled well before cb_counter is incremented.

5.1.3 Testing SampleCounter from PicoUtil

The new FPGA file will perform all the functions of PrimaryBoot.bit. In addition we can trigger either of the counters as follows:

Reading the counter from the host (CBcounter).

Press **Menu / Debugging / Input Port**, and the following dialog box will appear.



This will trigger a read of cb_counter by accessing port **0x74** (=CB_SAMPLE_DATA) from the PC host, and **,4** indicates a read of a 32bit word. Each time the read port command is executed the result will increase by one.

NOTE: The physical port address used by the PC hardware to access the counter will be offset by the base address of the Pico Card (say 0x7800). Pico.sys will add the base address (0x7800) to the port address (0x74) to obtain the physical address (0x7874). You can find the base address of the IO range assigned to the Pico card from device manager. Pico.sys will not allow you to access a port address outside the range assigned by the operating system.

Reading the PPC counter.

From the main screen of PicoUtil enter '**read 0x70000040,4**' in the Msg->Pico window and press enter. This will trigger a read of opb_counter. The address 0x70000040 is sent to monitor.exe running on the Pico-E14 card and four bytes are read from this address. Each time the message is sent to the Pico card the address will increment by 4! This is because the address is accessed in byte mode and the four byes surrounding address 0x70000040 are accessed for each byte.

©2010 Pico Computing, Inc.

This is not exactly what we want. In order to do a 32bit read, we will have to write a program and run it on the PPC. This is covered in the next section.

NOTE: The program monitor.exe runs in real mode and provides no protection for an invalid memory access. You can read and write address 0x70000040 or any other address within the address range of the PPC.

5.1.4 Reading cb_counter with a PC program

The project **c:\Pico\samples\E14_counter\ReadHostCounter** is a host side program to read the IO ports in the SampleCounterXX.bit. Within this project **.\Release\ReadHostCounter.exe** is ready to execute.

The relevant part of the source (readHostCounter.cpp) is quoted below:

```
if ((picoXfaceP=CreatePicoXface(paramsP, erBuf, sizeof(erBuf), &erC)) == NULL)
   {erParamP = erBuf; goto xit;}
//Get and print cb_counter
if ((erC=picoXfaceP->ReadPorts(0x74, (uint16_t*)&u32, 2)) < 0) goto xit;
printf("cb_counter=%08X\n", u32);
```

When this program is executed it will read and increment **cb_counter** (on the E14) or **pcmcia_counter** (on the E-12).

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\B>cd c:\pico\bin

C:\pico\bin>readHostCounter
(Aug 18 2006) cb_counter=00000000

C:\pico\bin>readHostCounter
(Aug 18 2006) cb_counter=00000001

C:\pico\bin>readHostCounter
(Aug 18 2006) cb_counter=00000002

C:\pico\bin>readHostCounter
(Aug 18 2006) cb_counter=00000003

C:\pico\bin>readHostCounter
(Aug 18 2006) cb_counter=00000004

C:\pico\bin>_
```

5.1.5 Reading opb_counter with a PPC program

The project **c:\pico\samples\E14_counter\ReadPPCcounter** is a PPC program to read the memory mapped counter port**.** Within this project **.\bin\ReadPPCcounter.elf** is ready to load and execute.

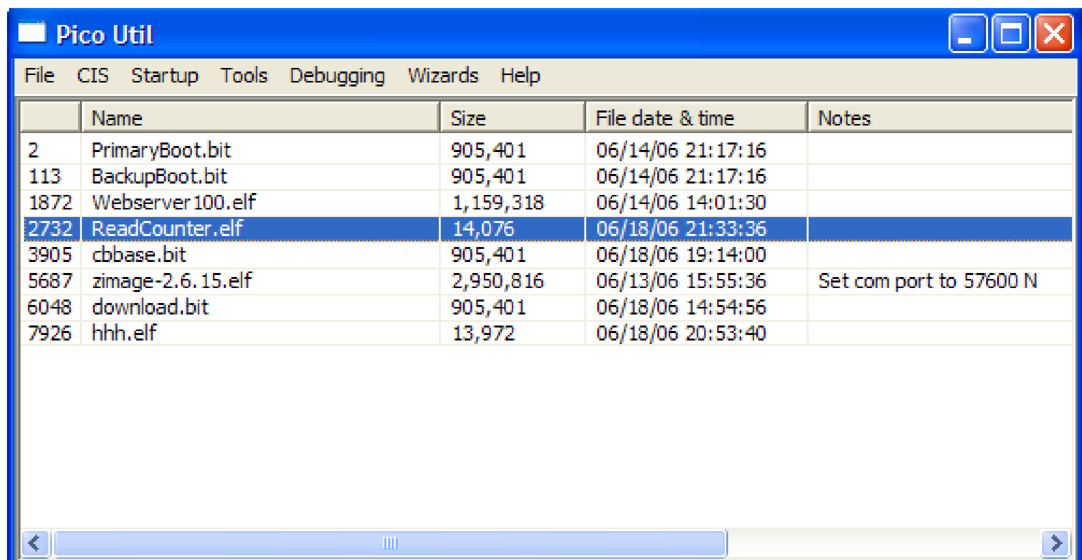
The relevant .C source is quoted below:

```
#include <stdarg.h>
#include <picoalways.h>      //relevant includes for Pico Card.

int main(void)
   {volatile uint32_t *counterP=(uint32_t*)0x70000040;
    Kprintf("Hello from ReadCounter, port=%08X\r\n", *counterP);
    return 0;
   } //main...

//End of file
```

The elf file must be written to the flash ROM of the Pico Card. Using PicoUtil and File / Write File, add **ReadPPCcounter.elf** to the flash ROM. The screen in PicoUtil will now appear as follows:



You can execute ReadCounter.elf by selecting it and pressing enter. Each time the program is executed it will report the value of the opb_counter and increment the counter (this time it will correctly increment by one!):

```
Hello from ReadCounter, port=00000001
Hello from ReadCounter, port=00000002
Hello from ReadCounter, port=00000003
```

5.2 Bus Mastering Examples

The Pico E-14 uses the cardbus interface which is capable of operating at PCI speeds – namely 33 Mwords/sec or 1Gbit/sec. In order to achieve speeds approaching this theoretical limit, the device must use 'burst mode DMA' mode. The Pico-E14 supports this mode and provides a file level interface which can be used in streaming mode. The actual speed depends upon the design of the PCI back plane and will vary significantly from machine to machine. DMA is an acronym for direct memory access. This technology is also called Bus Mastering.

The user can create a device and attach it to the core firmware in the module MemoryPeripherals.v. Detailed discussion of the Bus Mastering technology can be found in the manual Picodll.pdf.

There are two examples in this section:

1. BMcounter creates a device with a single counter register. Reads of this device increment the counter, writes to this device add the content of the CB bus to the register.
2. BMram create a device with block rams used to buffer the data to/from the PC. A simple state machine transforms the data between the input and output buffers.

In both of these examples the interface to the Pico Card uses the 'channel interface' defined in the file **Pico_channel.h**.

5.2.1 Single Counter

Firmware module.

BMcounter implements a single register and the necessary Bus Mastering interface to access this register. The kernel device of BMcounter is extremely trivial - comprising little more that a 32bit counter. This counter is incremented each time it is read and the value of the cardbus data is added to the counter on a write. The BMcounter device is implemented using the Pico Channel architecture. BMcounter uses a single channel to read or write the counter.

BMcounter is installed in the folder **Pico\samples\E14_BMcounter** and contains the following subdirectories and files:

- **.\firmware**. This folder contains two source files:
  - **BMCounter.v** is a Verilog file containing the logic to support the counter cited above;
  - **MemoryPeripherals_usermodule.includev** is an instantiation of the module which is included from \src\ files during a system build.
- **.\software.** This contains a C++ program BMcounter.cpp which can be used to read & write the counter from the PC host.

```
`include "E14Defines.v"

module BMSampleCounter
  (input         PicoRst,
   output [31:0] PicoDataOut,       //data returned
   input  [31:0] PicoAddr,          //address from BM/PCMCIA bus
   input  [31:0] PicoDataIn,        //data    from BM/PCMCIA
   input         PicoRd,            //IO Read from BM/PCMCIA bus
   input         PicoWr,            //IO Write to  BM/PCMCIA bus
   input         PicoClk            //CardBus clock
  );
```

PicoAddr is specified when the application program accesses the Pico card. picoAddr may be any address in the range [0x1000,0000 0x7FFF,FFFF], however, BMcounter uses channel 11 and will therefore be confined to the range [0x10B0,0000 - 0x10BF,FFFF]. The fundamental addresses used by BMcounter are:

```
`define BMSAMPLE_CHANNEL          11
`define BMSAMPLE_DATA_ADR         `BM_ADDR_FROM_CHANNEL(`BMSAMPLE_CHANNEL)
//adr of data for BMSAMPLE = 0x10Bxxxxx
`define BMSAMPLE_READ_STATUS_ADR  (`BM_STATUS_FROM_CHANNEL(`BMSAMPLE_CHANNEL))
//adr of read status  for BMSAMPLE = 0x100000B0
`define BMSAMPLE_WRITE_STATUS_ADR (`BM_STATUS_FROM_CHANNEL(`BMSAMPLE_CHANNEL)+4)
//adr of write status for BMSAMPLE = 0x100000B4
```

The bottom 20 bits of picoAddr can be ignored. The following logic develops the signal bmRead and bmWrite:

```
wire   bmSelected, bmRead, bmWrite;
assign bmSelected = {PicoAddr[31:20], 20'b0} == `BMSAMPLE_DATA_ADR;
assign bmRead     = bmSelected & PicoRd;           //CardBus read  data
assign bmWrite    = bmSelected & PicoWr;           //CardBus write data
```

The logic to read or write data might be something like:

```
reg [31:0] deviceData;
always @(posedge PicoClk)
  begin
  if (bmRead) deviceData  <= deviceData + 1;        else       //perform action required when data is read
  if (bmWrite) deviceData  <= deviceData + PicoDataIn;        //perform actios required when data is written
  end
```

The device must implement status registers, the following addition logic will be required:

```
wire   bmStatRead, bmStatSelectR, bmStatWrite, bmStatSelectW;
assign bmStatSelectR  = ({picoAddr[31:2], 2'b0} == `BMSAMPLE_READ_STATUS_ADR);
assign bmStatSelectW  = ({picoAddr[31:2], 2'b0} == `BMSAMPLE_WRITE_STATUS_ADR);
assign bmStatRead     = (bmStatSelectR & PicoRd);  //device read status
assign bmStatWrite    = (bmStatSelectW & PicoRd);  //device write status
reg    [11:0]readWordsAvailable;  //number of 32bit words available to read from device
reg    [11:0]writeWordsAvailable; //number of 32bit words that may be written to the device
```

The output to the cardbus would then be:

```
assign PicoDataOut =
    (bmRead      ? deviceData                                          : 0) |
    (bmStatRead  ? {`BM_STATUS_SIGNATURE, 6'h0, readWordsAvailable}  : 0) |
    (bmStatWrite ? {`BM_STATUS_SIGNATURE, 6'h0, writeWordsAvailable} : 0);
```

Software module.

The module BMcounter.cpp reads and writes the device created by BMcounter.v. The following statement:

```
#include <pico_channel.h>
cPicoChannel  channel(11);       //create channel 11
```

creates the channel to access the device. The Read function reads data from the Pico Card to the PC.

```
  //read 128 words from channel
  if ((erC=channel.Read(u32, sizeof(u32))) < 0)
    {printf("Error %u reading Pico channel\n%s\n", -erC,  InterpretError(erC, erBuf, sizeof(erBuf)));
return erC;}
```

based upon the login implemented in the device we would expect the values of u32[] to be n, n+1, n+2,...

The write function writes data to the Pico Card:

```
  //write 128 words to channel
  if ((erC=channel.Write(u32, sizeof(u32))) < 0)
    {printf("Error %u writing Pico channel\n%s\n", -erC,  InterpretError(erC, erBuf, sizeof(erBuf)));
return erC;}
```

based upon the login implemented in the device this should add the values of u32[] to the counter

©2010 Pico Computing, Inc.

register inthe firmware.

The value of the counter register can be retrieved using REadDevice():

```
if ((erC=channel.ReadDevice(0, &u32, sizeof(uint32_t)) < 0) goto err;
```

The first parameter (zero) is relative to the base address of the device (0x10B0,0000) and will therefore read the counter.

5.2.2 Device with RAM buffers

BMram implements a single register and the necessary Bus Mastering interface to access this register. The kernel device of BMram is extremely trivial - comprising little more that a 32bit counter. This counter is incremented each time it is read and the value of the cardbus data is added to the counter on a write. The BMram device is implemented using the Pico Channel architecture. BMram uses a single channel to read or write the counter.

BMram is installed in the folder **Pico\samples\E14_BMram** and contains the following subdirectories and files:

- **.\firmware**. This folder contains two source files:
  - **BMram.v** is a Verilog file containing the logic to support the counter cited above;
  - **MemoryPeripherals_usermodule.includev** is an instantiation of the module which is included from \src\ files during a system build.
- **.\software.** This contains a C++ program BMramS.cpp which can be used to read & write the counter from the PC host.

```
`include "E14Defines.v"

module BMRamDevice
  (input         PicoRst,
   output [31:0] PicoDataOut,       //data returned
   input  [31:0] PicoAddr,          //address from BM/PCMCIA bus
   input  [31:0] PicoDataIn,        //data    from BM/PCMCIA
   input         PicoRd,            //IO Read from BM/PCMCIA bus
   input         PicoWr,            //IO Write to  BM/PCMCIA bus
   input         PicoClk            //CardBus clock
  );
```

PicoAddr is specified when the application program accesses the Pico card. picoAddr may be any address in the range [0x1000,0000 0x7FFF,FFFF], however, BMram uses channel 16 and will therefore be confined to the range [0x1100,0000 - 0x110F,FFFF]. The fundamental addresses used by BMram are:

```
`define BMRAM_CHANNEL          16
`define BMRAM_DATA_ADR           `BM_ADDR_FROM_CHANNEL(`BMRAM_CHANNEL)
//adr of data for BMRAM = 0x11xxxxx
`define BMRAM_READ_STATUS_ADR    (`BM_STATUS_FROM_CHANNEL(`BMRAM_CHANNEL))
//adr of read status  for BMRAM = 0x10000100
`define BMRAM_WRITE_STATUS_ADR   (`BM_STATUS_FROM_CHANNEL(`BMRAM_CHANNEL)+4)
//adr of write status for BMRAM = 0x10000104
`define BMRAM_COMPUTE_STATUS_ADR (`BM_STATUS_FROM_CHANNEL(`BMRAM_CHANNEL)+8)
//adr of compute status for BMRAM (read only)
`define BMRAM_COMPUTE_CTRL_ADR   (`BM_STATUS_FROM_CHANNEL(`BMRAM_CHANNEL)+8)
//adr of compute control (write only) =0x10000108
```

The bottom 20 bits of PcoAddr can be ignored. The following logic develops the signal bmRead and bmWrite:

```
assign picobus_en       = ({PicoAddr[31:20], 20'b0}          ==
`BMRAM_DATA_ADR);          //ignore the bottom 20 bits.
assign pbus2Buf_wen     = picobus_en & PicoWr;
         //write from picobus to inputBuf
assign buf2Pbus_ren     = picobus_en & PicoRd;
         //read from outputBuf to picoBus
assign readStatus_ren   = PicoRd && ({PicoAddr[31:2], 2'b0}  ==
`BMRAM_READ_STATUS_ADR);   //read of outputBuf status
assign writeStatus_ren  = PicoRd && ({PicoAddr[31:2], 2'b0}  ==
`BMRAM_WRITE_STATUS_ADR);  //read of inputBuf  status
assign computeStatus_ren= PicoRd && ({PicoAddr[31:2], 2'b0}  ==
`BMRAM_COMPUTE_STATUS_ADR);//status of computational engine (for debugging)
assign computeCtrl_wen  = PicoWr && ({PicoAddr[31:2], 2'b0}  ==
`BMRAM_COMPUTE_CTRL_ADR);  //write to set sizeof buffer
```

Handling the receipt of data from the PC.

The logic to handle the input from the PC host follows:

```
//Use PicoAddr to trigger statemachine -------------------
always @ (posedge PicoClk)
  if (computeCtrl_wen) endofBuf[8:0] <= PicoDataIn[8:0];          //load
endofBuf count
```

This logic merely capture the endofBuf register sent from the PC-host.

```
always @ (posedge PicoClk)
  if (picobus_en)                                                 //access to
data aread of device
     begin
       if (PicoWr)
           begin
             hostInputDone  <= PicoAddr[10:2] == endofBuf[8:0];   //finished
when whole buffer is read (PicoAddr is a byte address)
             hostOutputDone <= 0;
           end
```

Handling the transfer of data from the Pico Card to the PC.

When the PC writes to the Pico Card, the buffer will be full when the PicoAddr gets to the endofBuf value. At this point the trigger hostInputDone will be asserted and the logic in the state machine will start executing.

```
always @ (posedge PicoClk)
  if (picobus_en)                                                 //access to
data aread of device
     begin
       if (PicoRd)
           begin
             hostOutputDone <= PicoAddr[10:2] == endofBuf[8:0];   //finished
when whole buffer is written
             hostInputDone  <= 0;
           end
```

When the PC is reading from the Pico Card, the buffer will be exhaused when the PicoAddr gets to endofBuf. At this point the state machine can reset back to its idle state.

```
always @ (posedge PicoClk)
  if (picobus_en)                                                 //access to
data aread of device
     begin
       if (currentState == `WORKING)
           begin
             hostInputDone  <= 0;
             hostOutputDone <= 0;
```

©2010 Pico Computing, Inc.

```
      end   end
```

When the state machine is working neither the input nor output flag will be asserted.

Processing the data on the FPGA.

The device is implemented as a state machine. When the host has transferred a buffer to process the state will shift from IDLE to WORKING. In the WORKING state the state machine will sequence through the input buffer, perform a trivial calculation (add 0x10101010), and store the results in outputBuf. When the state machine has finished processing the data it will shift to the state of FINISHED. THe state machine will stay in this state until the PC has read the data, at which point it will switch to state FINAL for one clock cycle and finally back to IDLE.

```
//state machine logic -------------------------------------
always @ (posedge stateMachineClk)
  begin
    case (currentState)
      `IDLE:
         begin
           inputBufAddr   <= 0;
           outputBufAddr  <= -1;
           outputBuf_wen  <= 0;                                 //no writing
to outputBuf until finished Pico input
           currentState   <= hostInputDone ? `WORKING : `IDLE;    //goto
WORKKING to process buffer
         end
      `WORKING:
         begin
           inputBufAddr   <= inputBufAddr  + 1;
           outputBufAddr  <= outputBufAddr + 1;
           computeDataOut <= computeDataIn + 32'h01010101;        //do
computation
           outputBuf_wen  <= 1;                                 //write to
outputBuf on next stateMachineClk
           if (inputBufAddr[8:0] == endofBuf[8:0])currentState <=
`FINISHED;//finished computation, goto finished
         end
      `FINISHED:
         begin //finished, waiting for host to pickup results.
           outputBuf_wen  <= 0;                                 //no more
writing to outputBuf.
           if (hostOutputDone) currentState <= `FINAL_STATE;      //shift to
idle state
         end
      `FINAL_STATE:
         currentState <= `IDLE;
    endcase
  end
```

Interacting with Pico.sys - status registers

The device must implement status registers to indicate to Pico.sys when the data is available for reading and when it the device may receive the next buffer full of data for processing on the FPGA. The data is marshalled into PicoDataOut using OR logic (see MemoryPeripherals.v)

```
assign PicoDataOut =
  buf2Pbus_ren     ?  buf2PbusDataOut
                        : //output from bram
  readStatus_ren   ? {`BM_READ_STATUS_SIGNATURE, 6'h0, (currentState ==
`FINISHED ? `BUFSIZE : 20'h0)}   : //output read status
  writeStatus_ren  ? {`BM_WRITE_STATUS_SIGNATURE,6'h0, (currentState == `IDLE
? `BUFSIZE : 20'h0)}   : //output write status
  computeStatus_ren? {LastPicoAddr,hostInputDone,hostOutputDone,
inputBufAddr,outputBufAddr,currentState}: //output current state
                      32'h0;
```

```
//nothing selected
```

- **buf2Pbus_ren** provides data directly from outputBuf when the PicoAddr is in the range [0x1100,0000 - 0x110F,FFFF] and PicoRD is asserted.
- **readStatus_ren** returns the status of the outputBuf. For our purposes it is sufficient to return BUFSIZE when the output buffer is ready (state == FINISHED) and zero otherwise
- **writeStatus_ren** returns the stat of the inputBuf. For our purposes it is sufficient to return BUFSIZE when the output buffer is ready (state == IDLE) and zero otherwise
- **computeStatus_ren** returns a set of registers used for debugging (see ShowDebugRegister in BMramS.cpp)

Software module.

The module BMramS.cpp reads and writes the device created by BMram.v. The following statement:

```
#include <pico_channel.h>
cPicoChannel  channel(16);       //create channel 16
```

creates the channel to access the device.

When the channel is successfully opened, the following statement specifies the size of the buffer. THe size is specified in units of 32bit words (hence the divide by four) :

```
 //Specify size of input buffer
channel.WriteDevice(8, &(ii=sizeof(u32)/4 - 1), sizeof(ii));
```

The Write function sends data to the Pico Card to the PC for processing:

```
channel.SetPicoAddr(0);
if ((erC=channel.Write(u32, sizeof(u32))) < 0) goto err;
```

The call to SetPicoAddr(0) is required so tht the logic in the device can properly sense when the end of buffer condition occurs. As soon as the buffer full condition is met the state machine will start procesing the input buffer. The Read function can be issued immediately, however, it will not return until the outputBuf full condition is asserted by the firmware device.

```
channel.SetPicoAddr(0);
if ((erC=channel.Read(u32, sizeof(u32))) < 0)
```

The function void ShowDebugRegister() uses the function cPicoChannel::ReadDevice()to retrieve the value of the computeStatus register. This can be called at any time to inspect the internal state of the firmware device.

©2010 Pico Computing, Inc.