

PicoBus32 Hello World Documentation

February 6, 2013

Contents

1	Introduction	1
2	Firmware	1
2.1	Using the PicoBus	1
2.2	Application Logic	2
2.3	Simulation	4
3	Purty	6
4	Software	6

1 Introduction

The PicoBus is a fairly standard 32-bit wide synchronous bus with read and write signals. It uses a shared address for reads and writes, so reads and writes are exclusive. It is driven by the host computer that the Pico card is in, so all peripherals on the PicoBus are accessible from software running on the computer.

For a further explanation of the PicoBus, including signal descriptions and how to perform a read or write, please consult the PicoAPI documentation.

This sample maintains 4 32-bit registers that hang off the PicoBus. One register stores the inverted data from the PicoDataIn bus. Another register XORs the input data from PicoDataIn with its current value. A third register sums the data from PicoDataIn when it is written. The last register tracks the number of times that any of the first three registers have been written.

2 Firmware

This project only uses the PicoBus. We use the PicoDefines.v file to tell the Pico framework that we need the PicoBus. The framework then takes care of impelementing the support logic to connect the PicoBus to the PCIe interface, which gives it access to the host computer.

2.1 Using the PicoBus

Here's what our PicoDefines.v file looks like:

```
'define USER_MODULE_NAME PicoBus32_HelloWorld

'define PICOBUS_WIDTH 32

// We define the type of FPGA and card we're using.
'define PICO_MODEL_M501
'define PICO_FPGA_LX240T

'include "BasePicoDefines.v"
```

You can see the relevant PICOBUS_WIDTH define that specifies the PicoBus as 32b wide. The Pico framework sees this define and connects the PicoBus to our module. Here's what the port list of our module looks like:

```
module PicoBus32_HelloWorld(
    input                PicoClk,
    input                PicoRst,
    input  [31:0]        PicoAddr,
    input  [31:0]        PicoDataIn,
    input                PicoRd,
    input                PicoWr,
    output reg [31:0]    PicoDataOut
);
```

Note that PicoAddr is a shared address for the read and write datapaths.

When PicoWr is high, the user logic that controls the address space designated by PicoAddr will perform its operation based upon the PicoDataIn bus. When PicoRd is asserted, the user logic that controls the address space designated by PicoAddr should respond by placing the appropriate data on the PicoDataOut bus the following PicoClk cycle.

2.2 Application Logic

Here we show how to create the 4 registers that hang off the PicoBus.

```
reg [31:0] TheReg0;    // This is a simple flip of the bits
reg [31:0] TheReg1;    // This is an XOR with the value currently in it
reg [31:0] TheReg2;    // This is adder starting at 0, and incrementing by
                        // each write
reg [31:0] TheReg3;    // This is a counter which increments by 1 on each
                        // write
```

Now that we have instantiated our registers, we need to allow them to be written from the PicoBus. First, we check the status of PicoWr to determine when the host wants to write data. Second, we check the address to make sure we're the intended peripheral on the bus, and which of our registers are being written. For example, here is the statement to check if we should write a register sitting at address 0 on the PicoBus:

```
if ( PicoWr && PicoAddr[31:0] == 32'h0 )
```

When performing a PicoBus write, the PicoWr signal is raised with the data to be written to the register. The target register should grab the data from PicoDataIn in the same PicoClk cycle as PicoWr is asserted.

```
always @(posedge PicoClk) begin
    if (PicoRst) begin
        // we'll start off with known reset values when the system asserts
        // PicoRst at startup.
        TheReg0 <= 32'h0;
        TheReg1 <= 32'hdeadbeef;
        TheReg2 <= 32'h0;
        TheReg3 <= 32'h0;
    end else begin
        // Here we process writes to the FPGA.
        // The PicoBus will assert PicoWr on the same cycle that PicoAddr is
        // valid and the data in on PicoDataIn.
        if ( PicoWr && PicoAddr[31:0] == 32'h0 )
            // simply invert the bits of the input data, and store it.
            TheReg0 <= ~PicoDataIn;
        if ( PicoWr && PicoAddr[31:0] == 32'h4 )
            // XOR the input data with our previous value.
            TheReg1 <= TheReg1 ^ PicoDataIn;
        if ( PicoWr && PicoAddr[31:0] == 32'h8 )
            // add the value written to our current value.
            TheReg2 <= TheReg2 + PicoDataIn;
        if ( PicoWr && ( PicoAddr[31:0] == 32'h0 || PicoAddr[31:0] == 32'h4
            || PicoAddr[31:0] == 32'h8 || PicoAddr[31:0] == 32'hc ) )
            // increment the value of TheReg3 by 1 each time a write is done
            // to any of the registers we have.
            // note that this statement is processed in parallel with the
            // statements that handle writes to each
            // of the specific registers.
            TheReg3 <= TheReg3 + 1;
    end
end
```

Note that the registers in this sample are initialized to different values when PicoRst is asserted. Also note that the registers are only written when PicoAddr corresponds to the valid address range for a given register. Lastly, note that multiple registers can be written in a single clock cycle.

Once the registers can be written via the PicoBus, we want to be able to read them via the PicoBus. Similar to the write, we must check the PicoRd and PicoAddr signal to see if we are the target register being read. When performing a PicoBus read, the PicoRd signal is raised a cycle before the data must be presented. PicoDataOut[31:0] must be assigned zero at all other times because it connects to a shared OR bus in its parent module. A simple if/else will accomplish this:

```
always @(posedge PicoClk) begin
    // Here we answer read requests on the PicoBus.
    // We place the requested data on PicoDataOut on the next cycle after
    // PicoRd and PicoAddr are valid.
    if (PicoRd && (PicoAddr[31:0] == 32'h0) )
        PicoDataOut <= TheReg0;
    else if (PicoRd && (PicoAddr[31:0] == 32'h4) )
        PicoDataOut <= TheReg1;
    else if (PicoRd && (PicoAddr[31:0] == 32'h8) )
        PicoDataOut <= TheReg2;
```

```

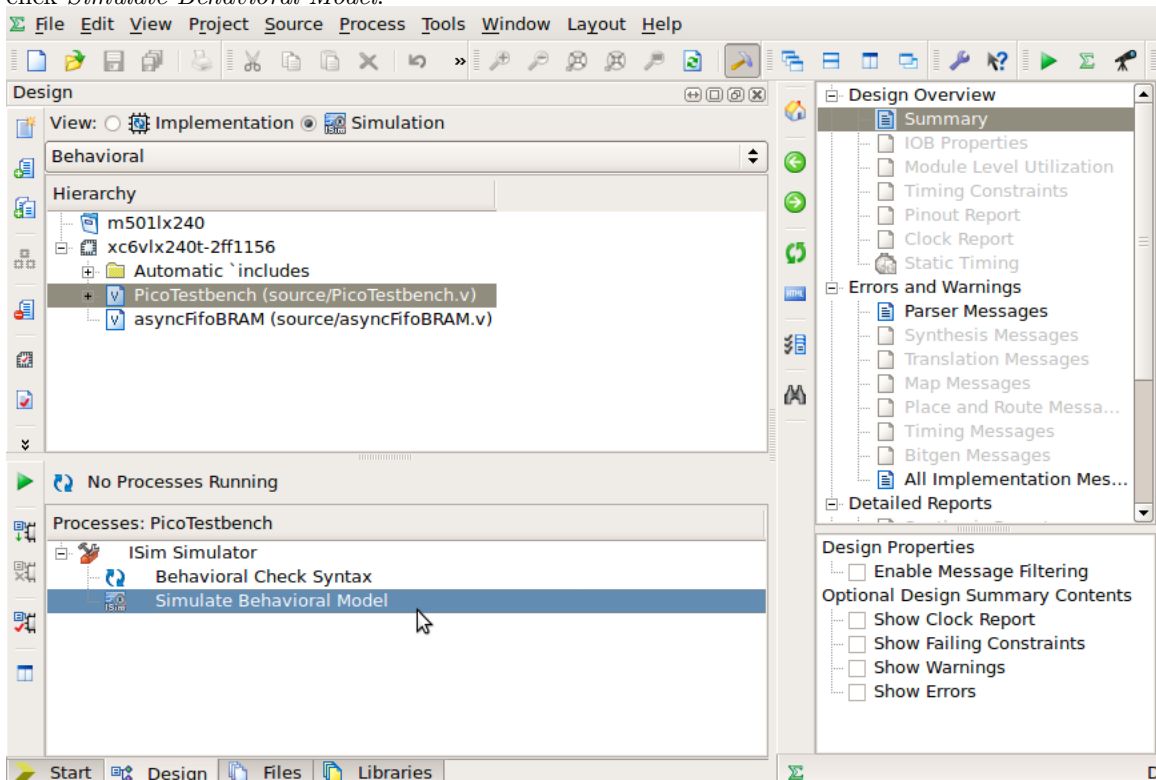
else if (PicoRd && (PicoAddr[31:0] == 32'hc) )
    PicoDataOut <= TheReg3;
else
    // since the PicoBus is shared, we need to drive our output to 0
    // when we're not being read from.
    PicoDataOut <= 32'h0;
end

```

Note that the user logic will only respond with data on the PicoBus if the PicoAddr corresponds to the address space for this user logic. Also note that the PicoBus is a shared resource, so if the user logic is not being read, it should drive the PicoDataOut bus to 0.

2.3 Simulation

We can now simulate the design using Isim to demonstrate the power of the streaming system. To launch Isim, select the *Simulation* radio button in ISE, select the *PicoTestbench*, and then double-click *Simulate Behavioral Model*.



The simulation should launch, and you can see the output in the console. On reads, the simulation will print out what is the data that is read back. This console output is the easiest way to see the results of a read from your module.

Console

ISim P.28xd (signature 0x54af6ca1)

This is a Full version of ISim.

Time resolution is 1 ps

Simulator is doing circuit initialization process.

0: PicoSimInit

Finished circuit initialization process.

Enable Stream 126 as in

Enable Stream 126 as out

Enable Stream 125 as in

Enable Stream 125 as out

8302000: Host received 32b

[Addr : 000]0x21524110

Writing 0x60 to registers at 0x10 and 0x20

Reading registers at 0x10 and 0x20

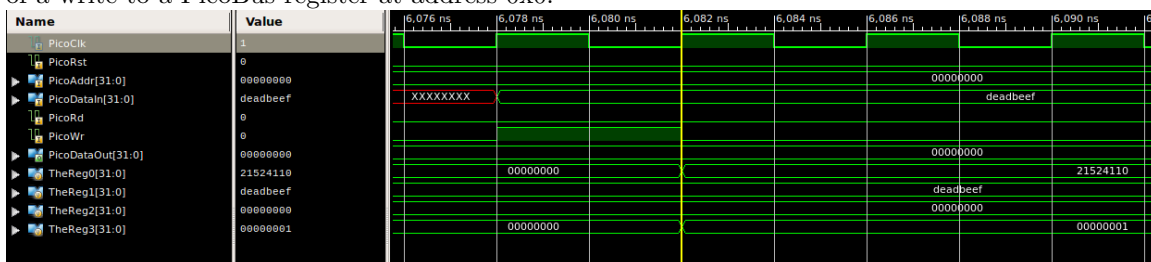
13166000: Host received 32b

[Addr : 000]0xdeadbe8f

15470000: Host received 32b

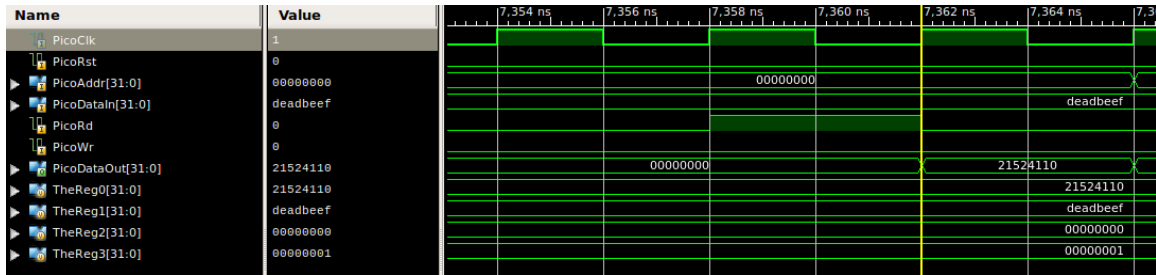
[Addr : 000]0x00000060

If you want more detail, you can also delve into the waveform viewer and look at the transaction there. We add the UserModule signals to the waveform and run the simulation. Here is a screenshot of a write to a PicoBus register at address 0x0.



Note that TheReg0 and TheReg3 are written on the cycle after a PicoWr is asserted with PicoAddr set to 0.

Here is a screenshot of a read from the PicoBus register at address 0x0.



Note that PicoDataOut is driven by TheReg0 1 cycle after the assertion of PicoRd.

3 Purty

After testing the user logic in simulation, we build a bitfile using the Xilinx ISE tools. To load an FPGA module, launch purty and right-click on the desired FPGA. Click on “Load FPGA” and navigate to the built bitfile. Now that the bitfile is loaded on the FPGA, we can perform a PicoBus read and/or write on the card. To do a read, right-click on the card we just loaded, then select “Read from PicoBus.” We then enter the desired read address. To do a write, we must also specify the desired write data.

For example, similar to the testbench, we now write 0xdeadbeef to the PicoBus register at address 0x0. After the write, we then read back that address.



Note that the information printed in the least-significant 32 bits in the Purty console matches that in the Isim console.

4 Software

Reading and writing via the PicoBus using Purty is mainly for debugging purposes. To actually use this function in a real application, we need to read and write the PicoBus in software.

First, we load the bit file into the FPGA and obtain a handle using the RunBitFile function, taking the file name from the command line:

```
// specify the .bit file name on the command line
if (argc < 2) {
    fprintf(stderr, "Please specify the .bit file on the command line.\n"
                  "For example: helloworld ../firmware/"
                  "M501_PicoBus32_HelloWorld.bit\n");
    exit(1);
}
bitFileName = argv[1];
```

```

printf("Loading FPGA with '%s' ...\n", bitFileName);
err = RunBitFile(bitFileName, &pico);
if (err < 0) {
    fprintf(stderr, "RunBitFile error: %s\n", PicoErrors_FullError(err, ibuf
        , sizeof(ibuf)));
    exit(1);
}

```

For this sample, we write the following test pattern via the PicoBus:

```

uint32_t    buf[] = {0x76543210, 0xfedcba98, 0x76543210, 0xfedcba98};
uint32_t    buf2[4];

```

Next we write to multiple registers on the PicoBus using the WriteDeviceAbsolute function. Each call to WriteDeviceAbsolute in the following code snippet writes 16 bytes of data from buf to addresses 0x0, 0x4, and 0x8 respectively.

```

// now we write to each of the registers we created in the firmware. each
will respond differently to a write,
// and we'll have to read back from them to see what each one did.
if((err = pico->WriteDeviceAbsolute(0, buf, 16)) < 0) {
    fprintf(stderr, "Error Writing: %d\n", err);
    exit(1);
}
if((err = pico->WriteDeviceAbsolute(4, buf, 16)) < 0) {
    fprintf(stderr, "Error Writing: %d\n", err);
    exit(1);
}
if((err = pico->WriteDeviceAbsolute(8, buf, 16)) < 0) {
    fprintf(stderr, "Error Writing: %d\n", err);
    exit(1);
}

```

Since we write 16B of data with these calls, the first 32 bits of data in the first call will be written to address 0, the second 32 bits to address 0x4, the third to address 0x8, and the last to address 0xC. With the second call, the data is written to addresses 0x4 through 0x10. The third WriteDeviceAbsolute call writes data to addresses 0x8 through 0x14.

Reading data is very similar to writing and can be done by using the ReadDeviceAbsolute function. For this demo we'll read back from address 0 and 0xC to see if the result is what we expect:

```

// read the register at address 0.
// this register should have inverted the bits of the 128-bit word that we
wrote to it.
if((err = pico->ReadDeviceAbsolute(0, buf2, 16)) < 0) {
    fprintf(stderr, "Error Reading: %d\n", err);
    exit(1);
}

// read the register at address 0xC.
// this register keeps a count of the total number of writes to any of the
registers.
if((err = pico->ReadDeviceAbsolute(0xC, buf2, 16)) < 0) {
    fprintf(stderr, "Error Reading: %d\n", err);
}

```

```
    exit(1);  
}
```

Note that each call to `ReadDeviceAbsolute` in the preceding code snippet reads 16 bytes of data into `buf2`. The data is read from PicoBus addresses `0x0` and `0xC` respectively. Similar to the write, since we are reading 16B on a 32-bit PicoBus, the first `ReadDeviceAbsolute` call reads from addresses `0x0` through `0xC`. The second `ReadDeviceAbsolute` call reads from addresses `0xC` through `0x18`.