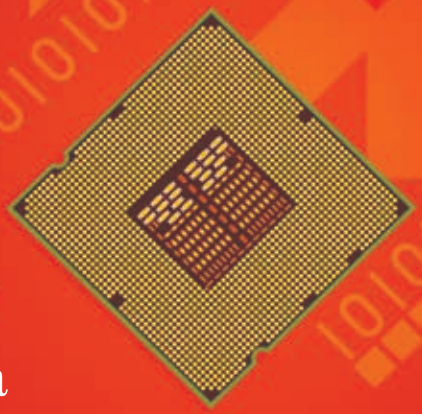
Getting the Most out of Your PicoBlaze Microcontroller

by Adam P. Taylor
Head of Engineering – Systems
e2v
aptaylor@theiet.org



Many FPGA applications can benefit from using a simple soft-core processor to ease the generation of sequential control structure.

The PicoBlaze™ is a compact 8-bit soft-core microcontroller that FPGA engineers instantiate within their selected Xilinx® FPGA. Once implemented, this core is completely contained within the FPGA fabric using only logic slices and Block RAMs; it requires no external volatile or nonvolatile memory.

Thanks to its small implementation footprint, it is possible for an FPGA to contain multiple PicoBlaze instantiations, with each instantiation used to implement control structures typically created by state machines. The result is a reduction in the development time along with a standardized approach to control structure generation. Thanks to the underlying high performance of the Xilinx FPGA fabric, PicoBlaze instantiations are often capable of outperforming many discrete 8-bit microcontrollers.

Let's take a look at how we can best utilize this handy device within our designs.

PICOBLAZE ARCHITECTURE

Before we can use the core, it is first a good idea to understand a little about its architecture. PicoBlaze is a very simple 8-bit microcontroller that is based around a RISC architecture (see Figure 1). The controller has a 12-bit address port, which means it can address as many as 4,096 memory locations. Each address location contains an 18-bit instruction that defines the operation the core must perform. Inputs and outputs to and from the core are possible via two 8-bit ports (one input, one output). The controller also provides an 8-bit identification port, allowing for up to 256 peripherals to be read from or written to. There is also a size-selectable scratchpad that can be 64, 128 or 256 bytes. As with all micros, PicoBlaze contains an arithmetic logic unit and support for one interrupt. These capabilities mean the controller offers many advantages to the FPGA design engineer.

One of the most important aspects of PicoBlaze is its highly deterministic nature, which means all instructions require two clock cycles for execution and interrupts are serviced with four clock cycles maximum. (You can find more detailed information on the PicoBlaze architecture within the Xilinx user guide that comes with your download.)

WHY USE PICOBLAZE?

FPGA applications usually require a combination of parallel and sequential operations, with data flow being predominantly parallel and control structures predominantly implemented as sequential structures, for example state machines (see *Xcell Journal* issue 81, "How to Implement State Machines in Your FPGA"). However, complex control structures if implemented as a state machine can become unwieldy, increasing the verification time and making modifications later in the development cycle more difficult. Complicated state machines also take more time to develop and if several are required, this time can be considerable.

You can also use PicoBlaze for serial communication control over RS232, I2C and SPI. In fact, anything that you would use a typical 8-bit micro for can be implemented within a PicoBlaze, with the upside of higher performance. Engineers have used PicoBlaze to implement PID controllers in control systems. They have used it with I2C, SPI or parallel DACs to create reference waveforms that range from simple square, sawtooth and triangular to more com-

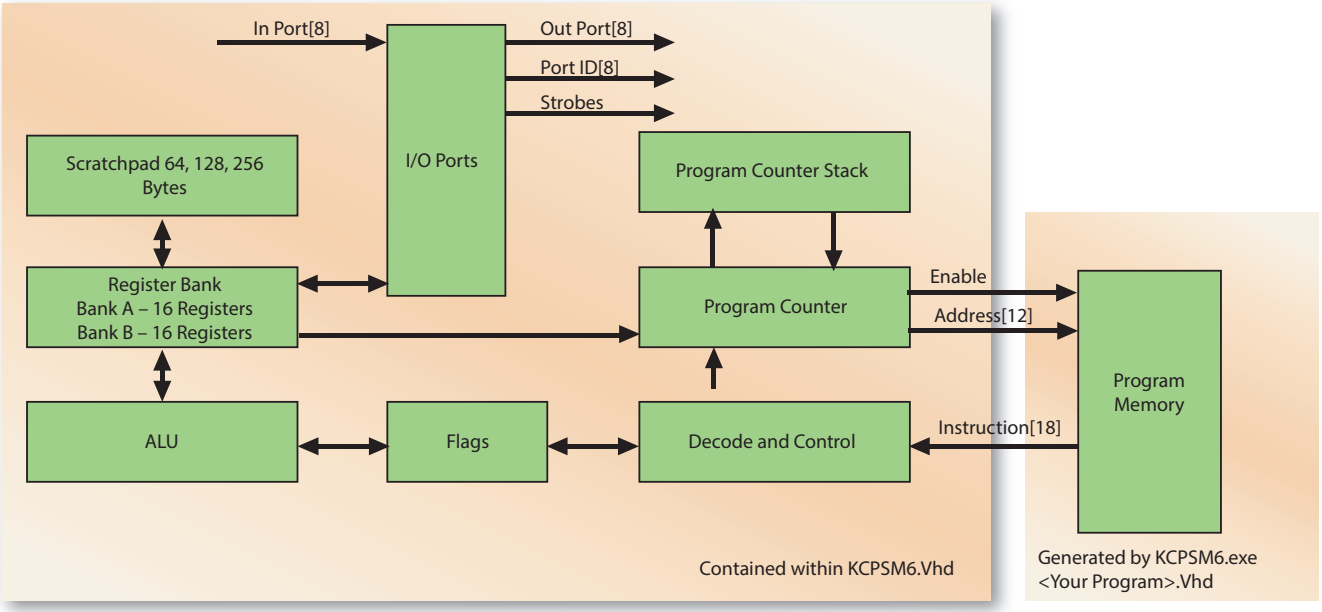


Figure 1 – PicoBlaze architecture, with processor in left box and memory at right

plicated sine/cosine waveforms (using the shift-and-add CORDIC algorithm).

Instantiating a PicoBlaze microcontroller within your FPGA to implement these sequential functions can result in a faster development time and allow for simplified modifications later in the development life cycle. And of course, being a soft core, PicoBlaze also helps address obsolescence issues and it encourages design reuse as ASM modules are developed.

FIRST PICOBLAZE INSTANTIATION

You can quickly implement a PicoBlaze within your design by following a few initial steps. First, make sure you have the latest version of the microcontroller for the device you are targeting. Xilinx makes these available from the PicoBlaze Lounge, offering versions that support the latest 7 series devices to those that work with older Spartan®-3 and Virtex®-4 devices.

Having downloaded the correct version of the processor, extract those files into your working directory and ensure that you read the "readme" file, paying close attention to the setting of the PATH and Xilinx environment variables as required. Within your working directory, you will now notice the following files or applications, along with the usual readme and license files and user guides.

- KCPSM6.Vhd: This is the actual source code for the PicoBlaze.
- KCPSM6.exe. This is the assembler program that you will use to generate the machine code and memory files required.
- ROM_Form.vhd. The assembler executable uses this file to generate the VHDL file your created program will reside in.
- KCPSM6_design_template vhd. This is a template instantiation of a PicoBlaze processor.
- All_kcpsm6_syntax.psm. This file is a definition of all assembler commands and syntax.

```
NAMEREG s0,led  ;rename S0 register to led
;As 8 bit processor we need four delay loops 256 * 256 * 256 *
256 = 4294967296
CONSTANT max1, 80 ;set delay
CONSTANT max2, 84 ;set delay
CONSTANT max3, 1e ;set delay
CONSTANT max4, 00 ;set delay
main:  LOAD led, 00; load the led output register with 00
flash:  XOR led, FF; xor the value in led register with FF i.e.
toggle
       OUTPUT led,01; output led register with port ID of 1
       CALL delay_init; start delay
       JUMP flash; loop back to beginning
delay_init: LOAD s4, max4;
                    LOAD s3, max3;
                    LOAD s2, max2;
                    LOAD s1, max1;
delay_loop:   SUB s1, 1'd; subtract 1 decimal from s1
                    SUBCY s2, 0'd; carry subtraction
                    SUBCY s3, 0'd; carry subtraction
                    SUBCY s4, 0'd; carry subtraction
                    JUMP NZ, delay_loop;
                    RETURN
```

Figure 2 – Snippet of assembler code for a program to flash LEDs

```
kcpsm6.exe
KCPSM6 Assembler v2.63
  Ken Chapman - Xilinx Ltd - 20th December 2013

Enter name of PSM file: test.psm

Reading top level PSM file...
  C:\hdl_projects\picoblaze\test.psm

A total of 21 lines of PSM code have been read

Checking line labels
Checking CONSTANT directives
Checking STRING directives
Checking TABLE directives
Checking instructions

Writing formatted PSM file...
  C:\hdl_projects\picoblaze\test.fmt

Expanding text strings
Expanding tables
Resolving addresses and Assembling Instructions
  Last occupied address: 00E hex
  Nominal program memory size: 1K (1024)    address(9:0)
  Occupied memory locations: 15
  Assembly completed successfully

Writing LOG file...
  C:\hdl_projects\picoblaze\test.log
Writing HEX file...
  C:\hdl_projects\picoblaze\test.hex
Writing VHDL file...
  C:\hdl_projects\picoblaze\test.vhd

KCPSM6 Options.....
        R - Repeat assembly with 'test.psm'
        N - Assemble new file.
        Q - Quit
```

Figure 3 – Using the KCPSM6 assembler to generate your memory files

At the very simplest level, you need declare only two components within your design.

For our example design, the final step is to create a new project in the ISE® Design Suite within which we can instantiate the PicoBlaze and its program memory if you are not adding them to an existing project.

Once we have completed the above steps, we are ready to start creating a PicoBlaze processor within our application. At the very simplest level, you need declare only two components within your design: the processor itself and the program memory as shown in Figure 1 (processor is within the left box and memory the right box, to provide context). Of course, if you have more than one instantiation you will have a number of memory components, each containing a different program. However, the first thing we need to do is understand the development flow of a typical project.

DEVELOPMENT FLOW

Creating your first PicoBlaze instantiation is simple. The first step is to create a blank text file using an editor like Notepad++. This file should have the file extension .PSM—for instance, test.psm. You program the microcontroller using the PicoBlaze assembler. Xilinx provides detailed information on this syntax in the file All_kcpsm6_syntax.psm, which comes with your download. However, it is easy to understand and learn this syntax. Figure 2 is an example of an assembler code snippet, which is a simple program to flash LEDs at 2-Hz frequency with a 40-MHz clock.

Once you are happy with your assembler program, the next stage is to run this program through the assembler executable that came with your download. Doing so will generate a memory file (VHDL for use within your FPGA), a log file and a hex file whose use we will look at later. Figure 3 shows the assembler process having been run for the code snippet above. Having run the assembler, you are now in a position to instantiate the PicoBlaze within your FPGA.

You are now in possession of the two VHDL files required: KCPSM6.vhd and the VHDL file created by the assembler program containing your application (in this case, test.vhd). The second stage is to declare the two components (KCPSM6 and Memory) within your VHDL design and instantiate them as shown in Figure 4. This simple VHDL example can be seen in the code snippet in Figure 5, which implements a PicoBlaze that will flash LEDs on an LX9 Spartan® development board.

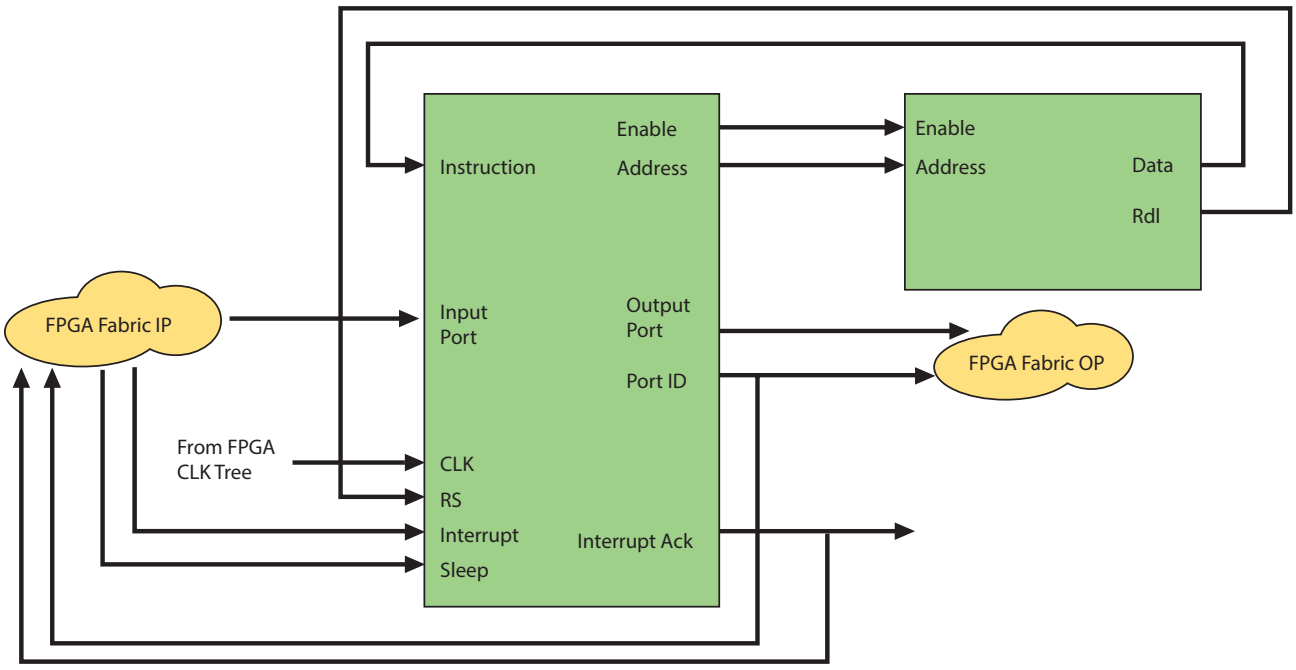


Figure 4 – PicoBlaze context diagram

SIMULATION AND VERIFICATION

Once you have instantiated the design files within your application, you will of course want to verify the performance of the system or module within a simulation environment before you progress to synthesis and implementation. As PicoBlaze uses logic slices and Block RAMs, it can be simulated very simply in programs like Mentor Graphics' ModelSim or Xilinx's ISim in ISE (or Xsim in the Vivado® Design Suite if that is where you are implementing your PicoBlaze).

Since the Block RAM contains the instructions for your program, simulation is simple. Essentially, all you need to provide is a clock and other inputs and outputs as required by the instantiation, Figure 6 shows ISim results for a PicoBlaze simulation and shows the two clock cycles between the loading of instructions.

UPDATING YOUR PROGRAM

One of the great benefits of having the PicoBlaze contained within the FPGA (and bit file) and is that following configuration of the FPGA the core will start executing the program within its RAM. However, in some cases you may need to modify the program the core is executing. While you could rerun the implementation stage including an updated memory file, depending upon the complexity of the remaining design this may take considerable time, especially if you are only trying out possibilities in the lab. It is therefore possible to update the program memory the core uses to modify the program and try it out before you rerun the implementation stage using the JTAG loader program also provided with your download.

The initial step in using the JTAG loader is to enable it within your design setting. Use the generic C_JTAG_LOADER_ENABLE : integer := 1 within your instantiation of one program memory. Note that you can set this parameter for only one memory instantiation within your design at a time.

Having enabled this facility within your design, you must first select the correct version for the operating system you are using from the JTAG_loader di-

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity pico_wave_top is
    Port ( clk : in  STD_LOGIC;
           led  : out  STD_LOGIC_VECTOR (3 downto 0));
end pico_wave_top;

architecture Behavioral of pico_wave_top is

component kcpsm6 is
  generic(   hwbuild : std_logic_vector(7 downto 0) := X"00";
             interrupt_vector : std_logic_vector(11 downto 0) := X"3FF";
             scratch_pad_memory_size : integer := 64);
  port (     address : out std_logic_vector(11 downto 0);
             instruction : in std_logic_vector(17 downto 0);
             bram_enable : out std_logic;
             in_port : in std_logic_vector(7 downto 0);
             out_port : out std_logic_vector(7 downto 0);
             port_id : out std_logic_vector(7 downto 0);
             write_strobe : out std_logic;
             k_write_strobe : out std_logic;
             read_strobe : out std_logic;
             interrupt : in std_logic;
             interrupt_ack : out std_logic;
             sleep : in std_logic;
             reset : in std_logic;
             clk : in std_logic);
  end component kcpsm6;

component test is
  generic(     C_FAMILY : string    := "S6";
               C_RAM_SIZE_KWORDS       : integer := 1;
               C_JTAG_LOADER_ENABLE : integer := 1);
  Port (       address : in std_logic_vector(11 downto 0);
               instruction : out std_logic_vector(17 downto 0);
               enable : in std_logic;
                      rdl : out std_logic;
                      clk : in std_logic);
  end component test;


SIGNAL instruction : std_logic_vector(17 DOWNTO 0);
SIGNAL address : std_logic_vector(11 DOWNTO 0);
SIGNAL enable : std_logic;
SIGNAL rd1 : std_logic;
SIGNAL kcpsm6_output : std_logic_vector(7 downto 0);
SIGNAL port_id : std_logic_vector(7 downto 0);
SIGNAL write_strobe:std_logic;
begin

ram_inst : test PORT MAP  (
             address => address,
             instruction => instruction,
             enable => enable,
             rdl => rd1,
             clk => clk);
```

```
pico_inst :  kcpsm6 PORT MAP (
             address => address,
             instruction => instruction,
             bram_enable => enable,
             in_port  => (OTHERS =>'0'),
             out_port => kcpsm6_output,
             port_id  => port_id,
             write_strobe => write_strobe,
             k_write_strobe => open,
             read_strobe => open,
             interrupt  => '0',
             interrupt_ack => open,
             sleep => '0',
             reset => rd1,
             clk => clk);

output_ports: process(clk)
begin
if rising_edge(clk) then
if write_strobe = '1' then
-- 4 LEDs at port address 01 hex Spartan LX9
   Development board
               if port_id(0) = '1' then
                    led <= kcpsm6_output(3 DOWNTO 0);
               end if;
end if;
end if;
end process;
end Behavioral;
```

Figure 5 – Code snippet for a PicoBlaze that will flash LEDs on an LX9 Spartan development board

rectory and copy it to your working directory (where you hex file is located). Now you can open a command window and navigate to your working directory and use the command below.

jtagloader –l <Your Project Name>.hex

Note: I have renamed the version of the executable for my OS jtagloader.exe.

This action will download the hex file created when you ran the assembler on your latest PSM file, with results as shown in Figure 7. As this file downloads, you will notice that the JTAG loader halts the core execution and downloads the new program to memory before releasing the core reset, at which point it starts running your new program.

Once you are happy with the updated behavior of the PSM file, you can rerun the implementation and bit file generation, ensuring that the next time the device is configured it will execute the updated program.

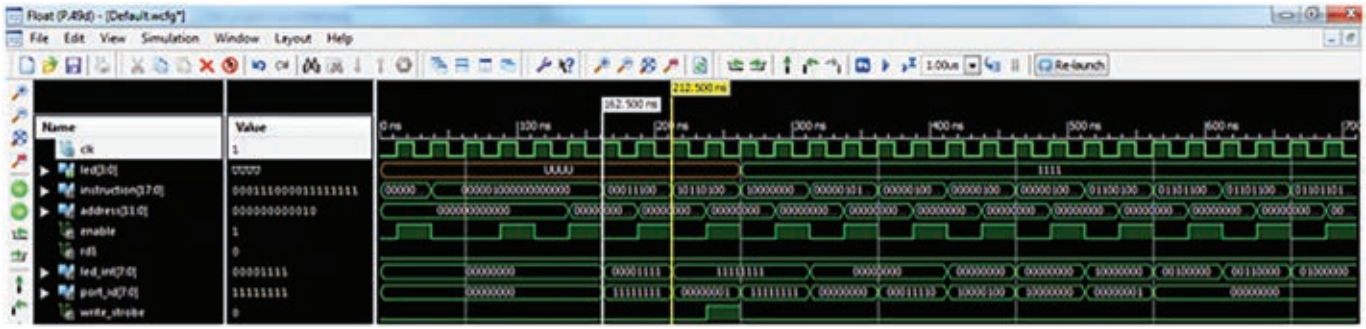


Figure 6 – ISim simulation results

Figure 7 – JTAG loader in action