XAPP387(v1.1) January 9, 2003

PicoBlaze 8-Bit Microcontroller for CPLD Devices

Summary

This application note describes the implementation of an 8-bit microcontroller design using a CoolRunner™-II CPLD. The PicoBlaze Microcontoller instructions can be customized to make an application-specific microcontroller.

CoolRunner-II devices, the latest CPLD family from Xilinx, offers both low power and high-speed performance. A complete VHDL code for PicoBlaze microcontroller design and C code for its assembler are available with this application note in **Source Code**, page 14.

Introduction

The PicoBlaze Microcontroller presented in this application note is a fully embedded macro derived from PicoBlaze, the Constant (k) Coded Programmable State Machine for Virtex™ and Spartan™. To simplify the generation of programs, a cross assembler written in C language is also provided.

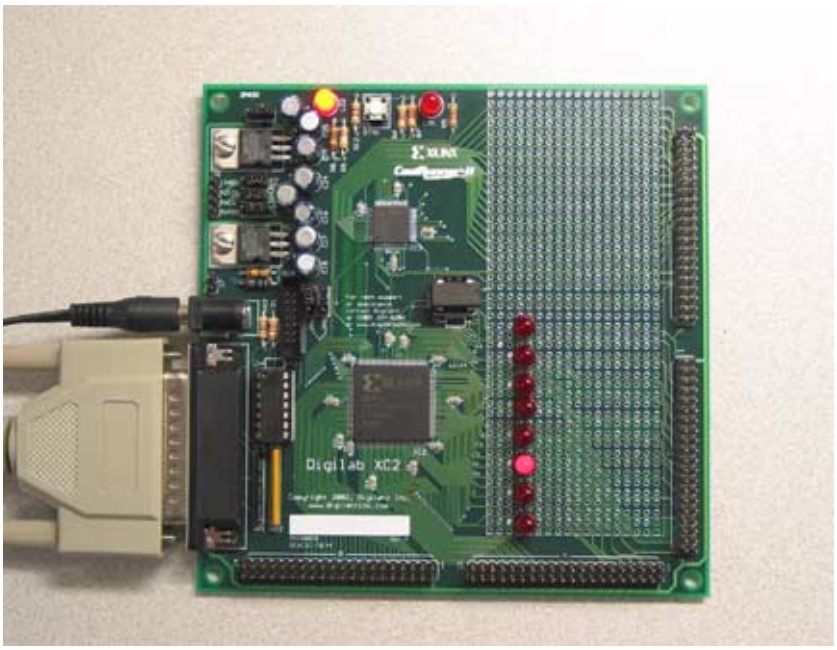

Figure 1: **PicoBlaze implemented on Digilent XC2 Demoboard**

The VHDL and C code for PicoBlaze can be easily customized to adjust size or change functionality. This makes the PicoBlaze an ideal application for content-sensitive microcontroller designs.

The PicoBlaze microcontroller provides 49 different instructions, eight 8-bit registers, 256 directly and indirectly addressable ports, and a maskable interrupt currently targeted to an XC2C256 device. Figure 2 is a block diagram of PicoBlaze architecture.

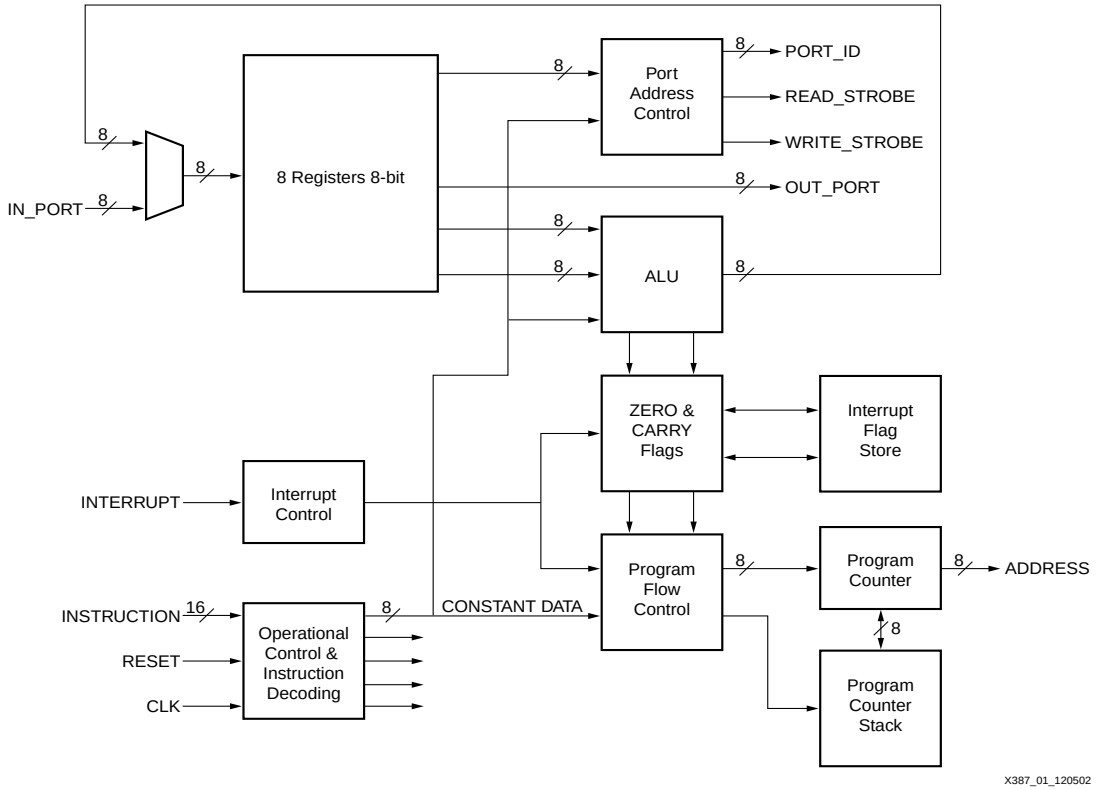

Figure 2: **PicoBlaze Architecture**

PicoBlaze Feature Set

General Purpose Registers

The register bank includes eight general-purpose 8-bit registers, s0 to s7. The register operations are completely general; no registers are reserved for special tasks or given priority over other registers.

ALU

The Arithmetic Logic Unit (ALU) provides all the simple operations expected in an 8-bit processing unit. All operations are performed using an operand provided by any register. The result is returned to the same register. For operations requiring a second operand, a second register is specified or a constant 8-bit value is embedded in the instruction. The ability to specify any constant value with no penalty to the program size or to its performance enhances the simple instruction set. To clarify, the ability to "ADD 1" is the equivalent to a dedicated INCREMENT operation. For operations requiring more than eight bits, addition and subtraction operations include an option to carry. Boolean operators (LOAD, AND, OR, XOR) provide the ability to manipulate and test values. There is also a very comprehensive Shift and Rotate group.

Flags/Program Flow Control

The ALU operation results affect the ZERO and CARRY flags. Using conditional and nonconditional program flow control instructions, this information determines the execution sequence of the program. JUMP commands specify absolute addresses within the program space.

CALL and RETURN commands provide subroutine facilities for commonly used sections of code. A CALL command is made to a specified absolute address, while a program counter

stack preserves the return address. The stack provides for a nested CALL with a depth of up to four levels, adequate for the program size supported.

Input/Output

The PicoBlaze solution has 256 input ports and 256 output ports. An 8-bit address value provided on the PORT bus together with a READ or WRITE strobe signal indicates the accessed port. The port address is either supplied in the program as an absolute value, or specified indirectly as the contents of any of the eight registers. Indirect addressing is ideal when accessing a block of memory constructed from external RAM.

During an INPUT operation, the value provided at the input port is transferred into any of the eight registers. An input operation is indicated by a READ_STROBE output pulse. Although using this signal in the input interface logic is not vital, it indicates that data has been acquired by PicoBlaze.

During an OUTPUT operation, the contents of any of the eight registers are transferred to the output port. A WRITE_STROBE output pulse indicates an output operation. This strobe signal is used in the design output interface logic, ensuring that only valid data is passed to external systems.

Interrupt

The process provides a single interrupt input signal. Using simple logic, multiple signals can be combined and applied to this one input signal. By default, the effect of the interrupt signal is disabled (masked) and is under program control to be enabled and disabled as required.

An active interrupt forces the PicoBlaze solution to initiate a "CALL FF" (i.e., a subroutine call to the last program memory location) for the designer to define a suitable course of action.

Automatically, the interrupt process preserves the contents of the current ZERO and CARRY flags and disables any further interrupt. A special RETURNI command is used to ensure that the end of an interrupt service routine restores the status of the flags and controls.

Constant (k) Coded Values

The PicoBlaze solution is in many ways a machine based on constants. Constant values are specified for use in the following aspects of a program:

- Constant data value for use in an ALU operation
- Constant port address to access a specific piece of information or control logic external to the PicoBlaze solution
- Constant address values for controlling the execution sequence of the program

The PicoBlaze instruction set coding is designed to allow constants to be specified within any instruction word. Hence, the use of a constant carries no additional overhead to the program size or its execution. This effectively extends the simple instruction set with a whole range of "virtual instructions."

Uniform Cycles

All instructions under all conditions execute over **two clock cycles**. When determining the execution time of a program, particularly when embedded into a real time situation, a uniform execution rate is of great value.

Program Length

The program length is 256 instructions. All address values are specified as 8-bits contained within the instruction coding. The fixed memory size promotes a consistent level of performance from the module; however, if necessary, the design can be expanded to support a larger memory range.

Complete PicoBlaze Instruction Set

This section lists a complete instruction set representing all op-codes in binary.

1. "X" and "Y" refer to the definition of the storage registers "s" in range 0 to 7.
2. "kk" represents a constant value in range 00 to FF.
3. "aa" represents an address in range 00 to FF.
4. "pp" represents a port address in range 00 to FF.
5. “C” and “D” represents the instruction decoding.

Program Control Group

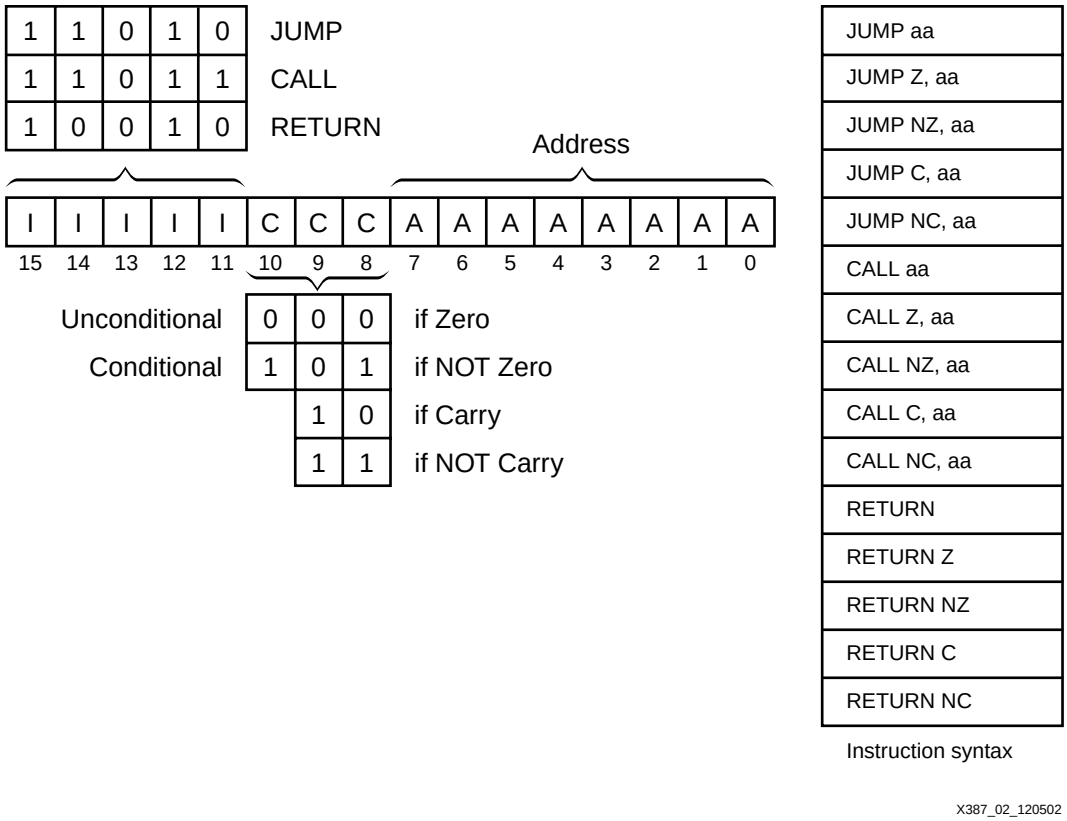

Figure 3: **Program Control Instructions**

JUMP - Under normal conditions, the program counter (PC) increments to point to the next instruction. The address space is currently fixed to 256 locations (00 to FF hex), making the program counter 8-bits wide. The top of the memory is FF hex and will increment to 00.

The JUMP instruction is used to modify the sequence by specifying a new address. However, the JUMP instruction can be conditional. A conditional JUMP is only performed if a test performed on either the ZERO flag or CARRY flag is valid. The JUMP instruction has no effect on the status of the flags.

Each JUMP instruction must specify the 8-bit address as a two-digit hexadecimal value. The assembler supports labels to simplify this process.

CALL - The CALL instruction is similar in operation to the JUMP instruction. It modifies the normal program execution sequence by specifying a new address. The CALL instruction is conditional. In addition to supplying a new address, the CALL instruction also causes the current PC value to be pushed onto the program counter stack. The CALL instruction has no effect on the status of the flags.

The program counter stack supports a depth of four address values, enabling a nested CALL

sequence to the depth of four levels to be performed. Since the stack is also used during an interrupt operation, at least one of these levels should be reserved when interrupts are enabled.

The stack is implemented as a separate buffer. When the stack is full, it overwrites the oldest value. Each CALL instruction must specify the 8-bit address as a two-digit hexadecimal value. To simplify this process, labels are supported in the assembler.

Hence, it is not necessary to reset the stack pointer when performing either a software or hardware reset. Therefore, there are no instructions to control the stack and no program memory is reserved for the stack.

RETURN - The RETURN instruction is the complement to the CALL instruction. The RETURN instruction is also conditional. The new PC value is formed internally by incrementing the last value on the program address stack, ensuring the program executes the instruction following the CALL instruction that resulted in the subroutine. The RETURN instruction has no effect on the status of the flags.

The programmer must ensure that a RETURN is only performed in response to a previous CALL instruction, so that the program counter stack contains a valid address. The cyclic implementation of the stack continues to provide values for RETURN instructions that cannot be defined. Each RETURN only specifies the condition for flag tests.

Shift and Rotate Group

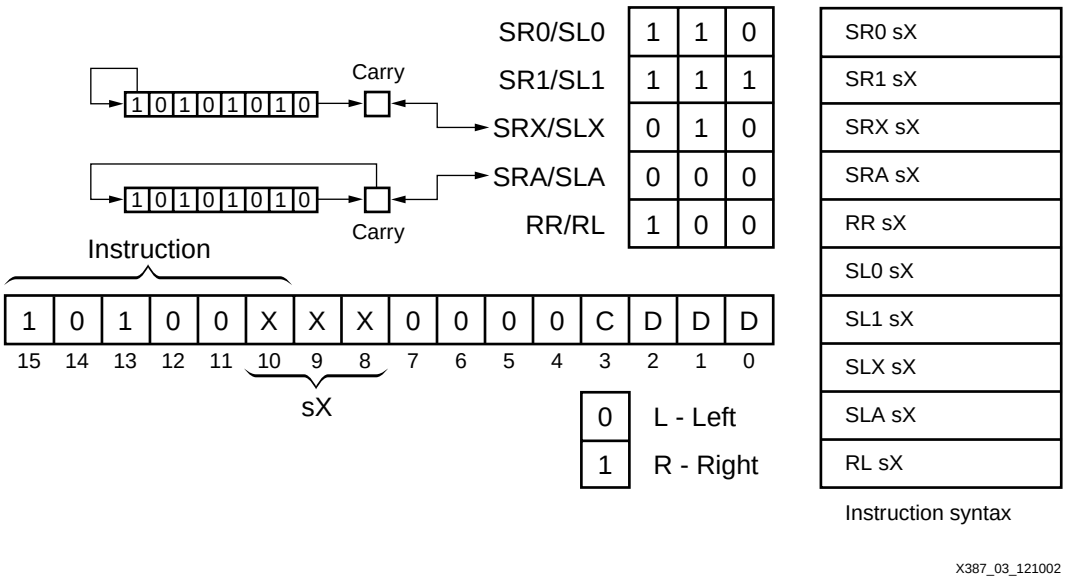

Figure 4: **Shift and Rotate Instructions**

SR0, SR1, SRX, SRA, RR - The shift and rotate right group all modify the contents of a single register to the right. All instructions in the group have an effect on the flags (see Figure 4).

SL0, SL1, SLX, SLA, RL - The shift and rotate left group all modify the contents of a single register to the left. All instructions in the group have an effect on the flags.

SR0/SL0 - Shifts register sX right/left by one place injecting “0”

SR1/SL1 - Shifts register sX right/left by one place injecting “1”

SRX/SLX - Shifts register sX right/left by one place injecting MSB/LSB

SRA/SLA - Shifts register sX right/left by one place injecting Carry flag

RR/RL - Rotates register sX right/left by one place injecting LSB/MSB

Logical Group

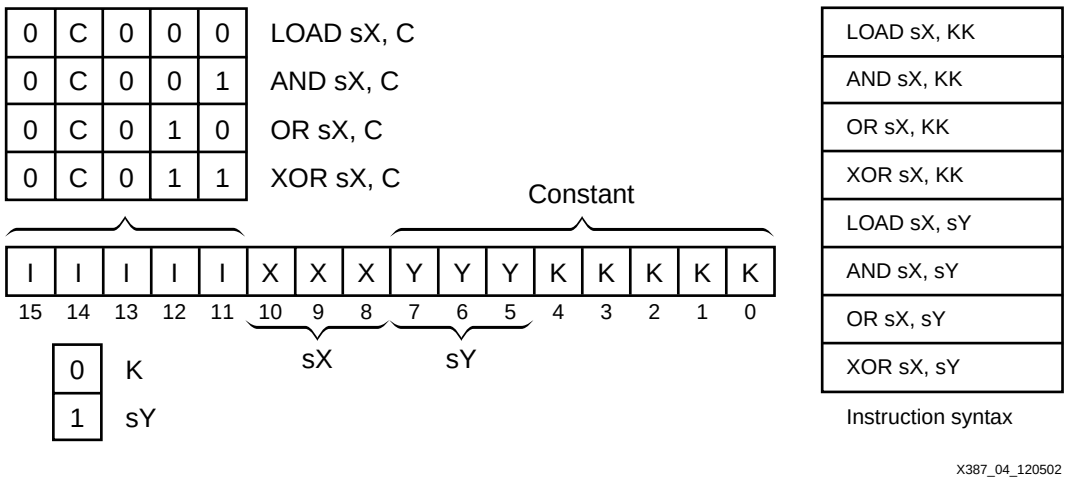

Figure 5: **Logical Instructions**

LOAD - The LOAD instruction specifies the contents of any register. The new value is either a constant or the contents of any other register. The LOAD instruction has no effect on the status of the flags.

Since the LOAD instruction does not affect the flags, it is used to reorder and assign register contents at any stage of the program execution. Because the load instruction is able to assign a constant with no impact to the program size or performance, the load instruction is the most obvious way to assign a value or clear a register.

Some implied "virtual" instructions are listed.

LOAD s0, s0 Load any register with its own contents achieves nothing and hence is a NO OPERATION consuming two clock cycles. This is used to form a delay in the program.

LOAD sX, 00 Load zero is the equivalent of a CLEAR register command.

AND - The AND instruction performs a bit-wise logical AND operation between two operands. For example, 00001111 AND 00110011 produces the result 00000011. The first operand is any register, and it is the register assigned the result of the operation. A second operand is also any register, or an 8-bit constant value. Flags are affected by this operation.

OR - The OR instruction performs a bit-wise logical OR operation between two operands. For example, 00001111 OR 00110011 produces the result 00111111. The first operand is any register. This register is assigned as the result of this operation. A second operand is also any register, or an 8-bit constant value. Flags are affected by the OR operation.

Useful in forming control signals, the OR instruction provides a way to force setting any bit of the specified register. The use of OR sX, 00 will set the zero flag if the contents of a register are zero without changing the contents of the register.

The following is a useful virtual instruction:

OR sX, 00 Clear CARRY flag and test register for ZERO.

XOR - The XOR instruction performs a bit-wise logical XOR operation between two operands. For example, 00001111 XOR 00110011 produces the result 00111100. The first operand is any register, and this register is assigned the result of the operation. A second operand is also any register, or an 8-bit constant value. The zero flag is affected by this operation and the carry flag will be cleared.

Arithmetic Group

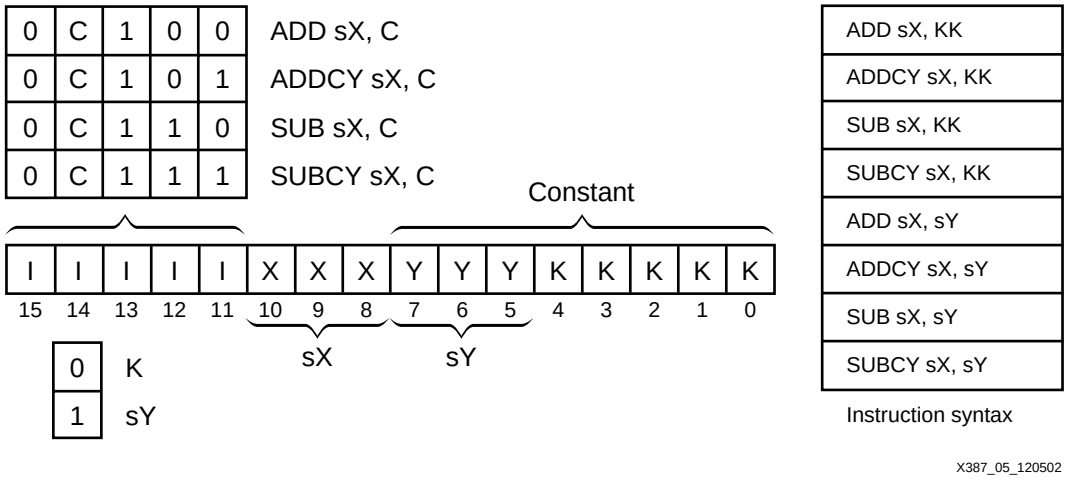

Figure 6: **Arithmetic Instructions**

ADD - The ADD instruction performs an 8-bit unsigned addition of two operands. The first operand is any register, and it is this register that is assigned the result of the operation. A second operand is also any register, or an 8-bit constant value. Flags are affected by this operation.

ADDCY - The ADDCY instruction performs an unsigned addition of two 8-bit operands together with the contents of the CARRY flag. The first operand is any register, and this register is assigned the result of the operation. A second operand is also any register, or an 8-bit constant value. Flags are affected by this operation.

SUB - The SUB instruction performs an 8-bit unsigned subtraction of two operands. The first operand is any register, and this register is assigned the result of the operation. The second operand is also any register, or an 8-bit constant value. Flags are affected by this operation.

SUBCY - The SUBCY instruction performs an 8-bit unsigned subtraction of two operands together with the contents of the CARRY flag. The first operand is any register, and this register is assigned the result of the operation. The second operand is also any register, or an 8-bit constant value. Flags are affected by this operation.

Input/Output Group

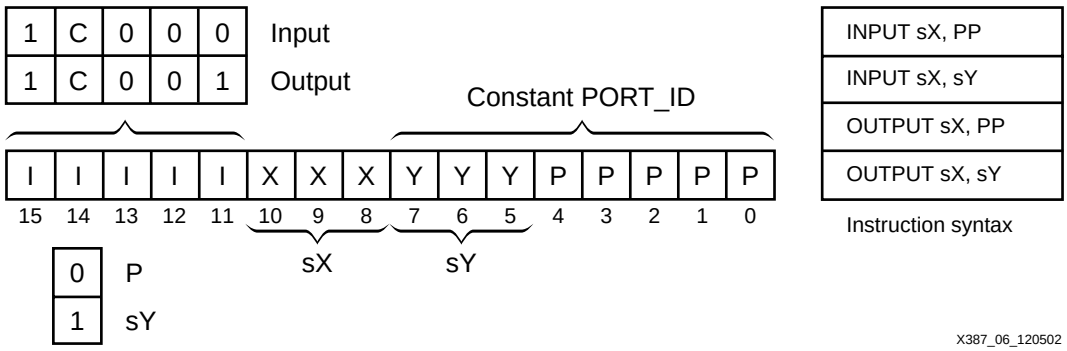

Figure 7: **Input/Output Instructions**

INPUT - The INPUT instruction enables data values external to the PicoBlaze solution to be transferred into any one of the internal registers. The port address (in the range 00 to FF) is defined by a constant value, or indirectly as the contents of any other register. Flags are not affected by this operation.

The user interface logic is required to decode the port address value and supply the correct data. The signal waveforms are shown in Figure 8. Note that the READ_STROBE provides an indicator that a port has been read, but it is not vital to qualify a valid address.

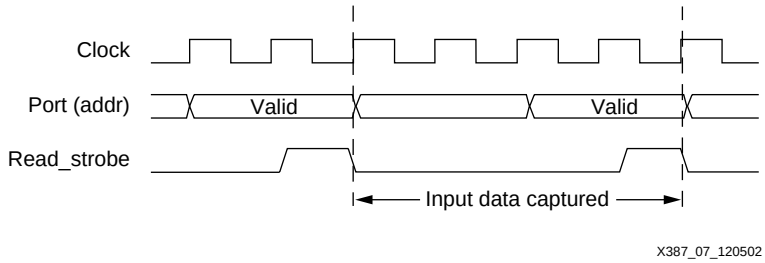

Figure 8: **Input Signal Waveform**

OUTPUT - The OUTPUT instruction enables the contents of any register to transfer to logic external to the PicoBlaze solution. The port address (in the range 00 to FF) is defined by a constant value, or indirectly as the contents of any other register. Flags are not affected by this operation.

The user interface logic is required to decode the port address value and enable the correct logic to capture the data value. The WRITE_STROBE is used in this case to ensure the transfer of valid data only. The signal waveforms are shown in Figure 9.

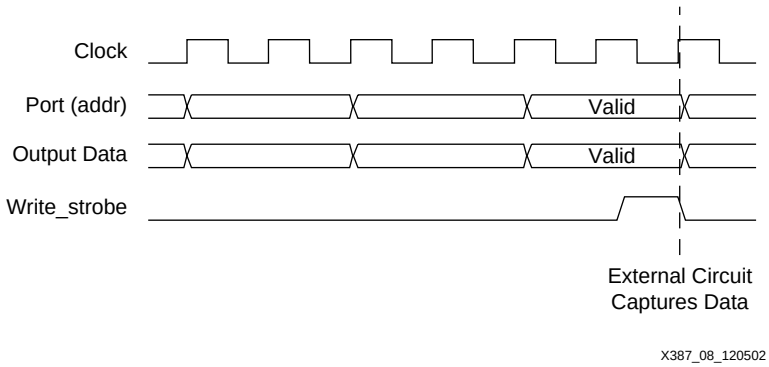

Figure 9: **Output Signal Waveform**

Interrupt Group

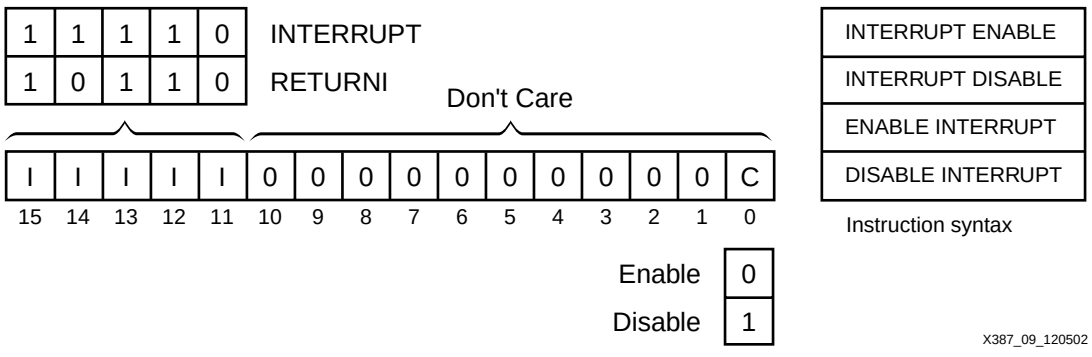

Figure 10: **Interrupt Instructions**

RETURNI - The RETURNI instruction is a special variation of the RETURN instruction. It concludes an interrupt service routine. The RETURNI is unconditional and always loads the program counter (PC) with the last address on the program counter stack. The address does

not increment in this case, because the instruction at the stored address needs to be executed. The RETURNI instruction restores the flags to the point of interrupt condition. It also determines the future ability of interrupts using ENABLE or DISABLE as an operand.

A RETURNI is only performed in response to an interrupt. Each RETURNI must specify if a further interrupt is enabled or disabled.

ENABLE INTERRUPT and DISABLE INTERRUPT - These instructions are used to set and reset the INTERRUPT ENABLE flag. Before using ENABLE INTERRUPT, a suitable interrupt routine must be associated with the interrupt address vector (FF). Never enable interrupts while performing an interrupt service.

PicoBlaze Assembler

Running the Assembler

An assembler ASM.EXE is provided to simplify the generation of programs. This assembler is written in C and compiled with Microsoft Visual Studio 6.0. It is a simple DOS program that can be run under a DOS window. Figure 11 illustrates the process for using the assembler.

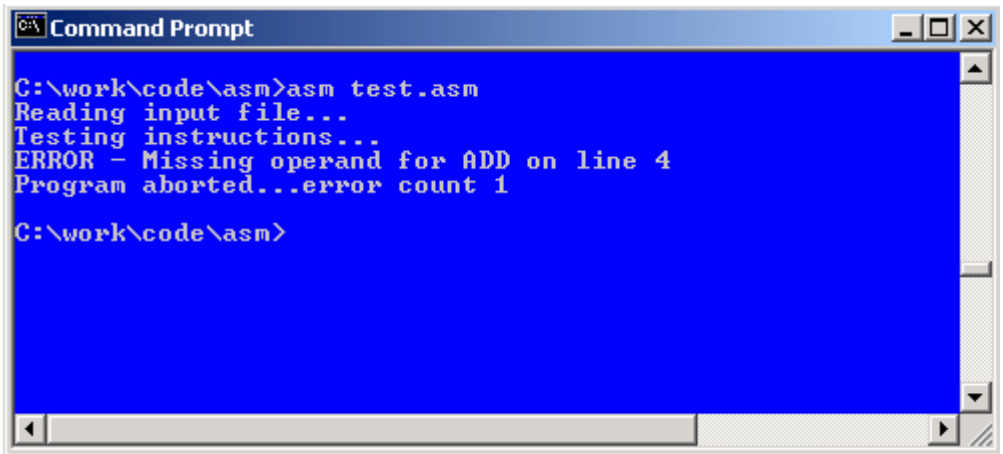

Figure 11: **PicoBlaze Assembler**

Programs are best written with Notepad type tools. The file is saved with a .asm file extension. Place the ASM.EXE file in the same directory as the program file or set the ASM.EXE directory in the PATH. Open a DOS box and navigate to the directory. Then run the assembler asm <filename>.asm. The assembler executes very quickly and the display often appears immediately.

Assembler Errors

Assembler error messages are displayed to help determine the reason for an error. The assembler also displays the line it was analyzing when it detected a problem. Figure 11 shows an example: the error messages are self-explanatory.

Assembler Files

The assembler reads the assembly program file and generates five output files, as shown in Figure 12. Note that all output files are overwritten each time the assembler is executed.

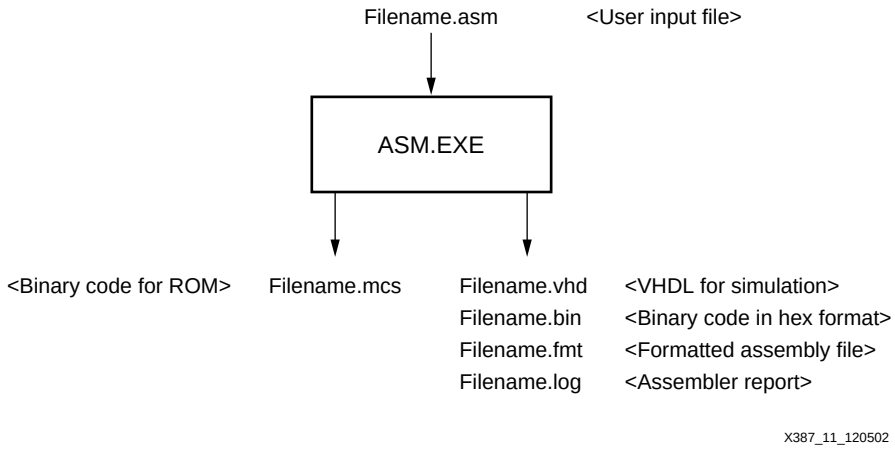

Figure 12: **Assembler Files**

Filename.vhd – This file is a VHDL module for program memory generated by the assembler and suitable for synthesis and simulation.
Filename.bin – This file is the binary code for the program memory in hex format and it is suitable for program debugging.
Filename.fmt – This file is the original program reformatted for easier reading. Looking at this file is also a good way to see that everything has been interpreted as intended.
Filename.log – This log file shows the assembler process performed and any error messages generated during the process.
Filename.mcs – This file is the binary code for the program memory in Intel's MCS-86 format. This file can be used to program an external memory if needed.

Program Syntax

The best way to understand the difference between valid and invalid syntax is to look at the examples and try the assembler. However, there are some simple rules which can be of initail assistance:

No blank lines – Use a semicolon for blank lines

Comments – Any item on a line following a semicolon (;)

Constant – Must be specified in the form of a two-digit hexadecimal value (00 – FF)

Line Labels – Identify program lines for JUMP or CALL instructions; should be followed by a colon (:)

Instructions – The instructions should be of the formatted like those described in the section **Complete PicoBlaze Instruction Set**, page 4 of this document. Instructions and the first operand must be separated by at least one space.

The assembler supports three assembler directives. These are commands included in the program which are used purely by the assembly process and do not correspond to instructions executed by the assembler.

CONSTANT Directive – Assigns an 8-bit constant value to a label

NAMEREG Directive – Assigns a new name to any of the eight registers

ADDRESS Directive – Forces the instructions that follow it to commence at a new address value.

The assembler will accept any mixture of upper and lower case characters for the instruction and automatically convert them to upper case. A simple example of some acceptable instruc-

tions is shown in Figure 13.

Note: This example is not coded in an efficient manner; it is only intended to show some of the program syntax.

The compiled code and the memory addresses are also shown in Figure 14.

```
        Constant shifter_port, 04    ;declare port
        Namereg s7, shifter_reg      ;declare register
Loop1:  Load shifter_reg, 01         ;init shifter reg
Loop2:  Output shifter_reg, shifter_port
        SL0 shifter_reg              ;shift left with 0
        Jump NZ, loop2               ;goto loop2 when s7<>0
        Jump loop1                   ;goto loop1
```

X387_12_120502

Figure 13: **A Simple Example**

Address	Hex	Binary	
00	0701	0000011100000001	Load s7, 01
01	8F04	1000111100000100	Output s7, 04
02	A706	1010011100000110	SL0 s7
03	D501	1101010100000001	Jump NZ, 01
04	D000	1101000000000000	Jump 00

X387_13_120502

Figure 14: **Compiled Code and Addresses**

Using the PicoBlaze Macro

It is typical in a microcontroller environment to store the program code in an external memory for the microcontroller. The designer writes assembly code and the PicoBlaze assembler can compile the assembly code into binary, generating Intel MCS format files to be used to program the external memory. The PicoBlaze operation with the external program memory block diagram is shown in Figure 15.

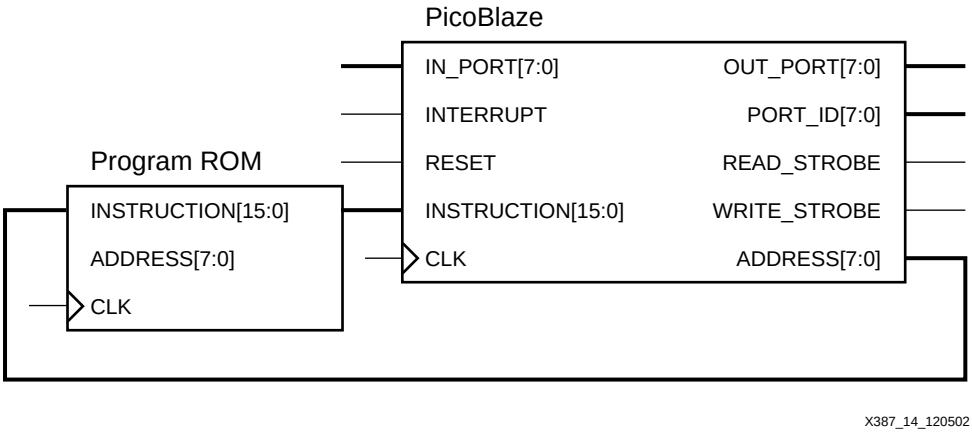

Figure 15: **PicoBlaze Operation Block Diagram**

The PicoBlaze assembler can also generate a VHDL code to model the program memory. User can use this VHDL module and the PicoBlaze module for board level functional simulation.

How to Customize the PicoBlaze Microcontroller

For some content-sensitive microcontroller applications it is ideal to have a customized microcontroller and its own assembler. In a standard microcontroller application the program code in the memory can be extracted, making reverse engineering very easy. With a customized microcontroller and assembler it is almost impossible to know the microcontroller behavior by just studying the program code.

The PicoBlaze microcontroller VHDL code and its associated assembler C code are made very easy to customize. The VHDL and C code shown in Figure 16 are the instruction ID declarations. Each instruction ID is defined in both VHDL and C code in *PicoBlaze.vhd* and *asm.cpp*. Users can make their own instruction ID decoding by changing the ID in these two files.

C code for Assembler

```
/* program control group */
char *jump_id = "11010";
char *call_id = "11011";
char *return_id = "10010";

/* logical group */
char *load_k_to_x_id = "00000";
char *load_y_to_x_id = "01000";
char *and_k_to_x_id = "00001";
char *and_y_to_x_id = "01001";
char *or_k_to_x_id = "00010";
```

VHDL for C PicoBlaze Microcontroller

```
--
-- program control group
constant jump_id : std_logic_vector(4 downto 0) := "11010";
constant call_id : std_logic_vector(4 downto 0) := "11011";
constant return_id : std_logic_vector(4 downto 0) := "10010";
--
-- logical group
constant load_k_to_x_id : std_logic_vector(4 downto 0) .= "00000";
constant load_y_to_x_id : std_logic_vector(4 downto 0) .= "01000";
constant and_k_to_x_id : std_logic_vector(4 downto 0) := "00001";
constant and_y_to_x_id : std_logic_vector(4 downto 0) := "01001";
constant or_k_to_x_id : std_logic_vector(4 downto 0) := "00010";
```

X387_16_120502

Figure 16: **Instruction ID Code**

Adding an instruction

Here are the steps to add a new instruction to the PicoBlaze microcontroller:

- Modify PicoBlaze.vhd
- Add a constant with the instruction code:

constant new_instruction_id : std_logic_vector(4 downto 0) := “10101";

- Add instruction to the decoding signal

  i_new_instruction <= '1' when instruction(15 downto 11) = new_instruction_id else '0';
- Define the VHDL component with the functionality of the new instruction
- Add the new component to PicoBlaze.vhd
- Add the new instruction to register_and_flag_enable.vhd for register decoding enable

The assembler will also need to be modified by the following steps

- Add the new instruction to asm.cpp

  char *new_instruction_id = “10101“;
- Add the new instruction to the instruction_set
- Add the case to test_instructions function of asm.cpp
- Add the case to write_program_word function of asm.cpp
- Recompile asm.cpp to create asm.exe

Subtracting an instruction

Subtracting an instruction is the reverse of adding an instruction for PicoBlaze.vhd. There is no need to modify the assembler since the instruction is never used in the program.

A simple example of this addition/subtraction is included in the PicoBlaze source code. A new instruction called “FLIP” has been added. All this instruction does is reverse the order of bits in a register from MSB to LSB, MSB –1 to LSB + 1 and so on. The following VHDL code explains its functionality:

```
bus_width_loop: for i in 0 to 7 generate
begin
     FF:
     process (clk)
      begin
          if (clk'event and clk = '1') then
             Y(i) <= operand(7-i);
          end if;
     end process FF;
 end generate bus_width_loop;
```

Without this new instruction it would need several instructions to do the same function. This newly added instruction can be found in the source code and commented with “added new instruction here”.

Design Example

Some microcontroller design examples can be found in Xilinx application note XAPP213. These design examples are modified to accommodate the PicoBlaze microcontroller resources and used to verify the PicoBlaze microcontroller functionality.

Design Implementation

The PicoBlaze design described in this application note is targeted to a XC2C256-5TQ144 CoolRunner-II device. The device utilization data is shown in the following table.

Table 1: **PicoBlaze XC2C256 Device Utilization**

Device Resource	Available	Used	% Utilization
Macrocells	256	212	83%
I/O Pins	118	53	45%
Product Terms	896	642	72%
Registers	256	155	61%
Function Block Inputs	640	451	70%

Source Code

XAPP387 - **http://www.xilinx.com/products/xaw/coolvhdlq.htm**

Conclusion

PicoBlaze microcontroller for CoolRunner-II devices is a derivative of the Virtex PicoBlaze. It is designed to allow easy customization, and is most effectively implemented in content-sensitive applications. This design is targeted for a 256-macrocell XC2C256-5TQ144 CoolRunner-II device.

The CoolRunner-II device family features very low power consumption and high performance. Its flexible architecture is the best choice for PicoBlaze microcontroller type System On a Chip designs.

References

1. Ken Chapman. Xilinx Application Note XAPP213 PicoBlaze 8-Bit Microcontroller for Virtex Devices.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/17/02	1.0	Initial Xilinx release.
01/09/03	1.1	Minor revisions.