



XAPP213 (v2.1) February 4, 2003

PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices

Author: Ken Chapman

Summary

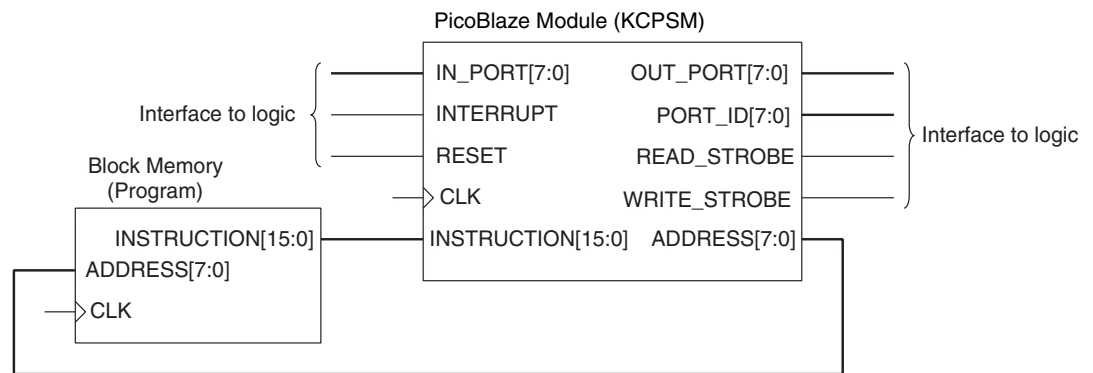
The Constant (k) Coded Programmable State Machine (KCPSM) solution presented in this application note is a fully embedded 8-bit microcontroller macro for the Virtex™, Virtex-E, Spartan™-II and Spartan-IIE devices (see [XAPP627](#) for Virtex-II designs). Although it could be used for processing of data, the PicoBlaze macro is most likely to be employed in applications requiring a complex, but non-time-critical state machine.

This revised version (for ISE compatibility) of the popular KCPSM macro (PicoBlaze) has still been developed with one dominant factor being held above all others—its size. The result is a microcontroller that occupies just 76 Spartan-IIE slices, which is 9% of the smallest XC2S50E device and 2.5% of the XC2S300E device. Together with this small amount of logic, a single block RAM is used to form a ROM store for a program of up to 256 instructions. Even with such size constraints, the performance is respectable at approximately 40 MIPS, depending on device speed grade.

The PicoBlaze module is totally embedded into the device and requires no external support. Any logic can be connected to the module inside the Virtex or Spartan device meaning that any additional features can be added to provide ultimate flexibility. It is not so much what is inside the PicoBlaze module that makes it useful, but the environment in which it lives.

Introduction

Figure 1 is a block diagram of a PicoBlaze module. The Spartan PicoBlaze modules require no external support and provide a flexible environment for other logic connections into the PicoBlaze module.



x213_01_121302

Figure 1: PicoBlaze Module Block Diagram

The PicoBlaze module is supplied as VHDL and as a precompiled soft macro that is handled by the place and route tools to merge with the logic of a design. This plot (**Figure 2**) from the FPGA Editor viewer shows the macro in isolation within the smallest Spartan-IIE device.

© 2003 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and further disclaimers are as listed at <http://www.xilinx.com/legal.htm>. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

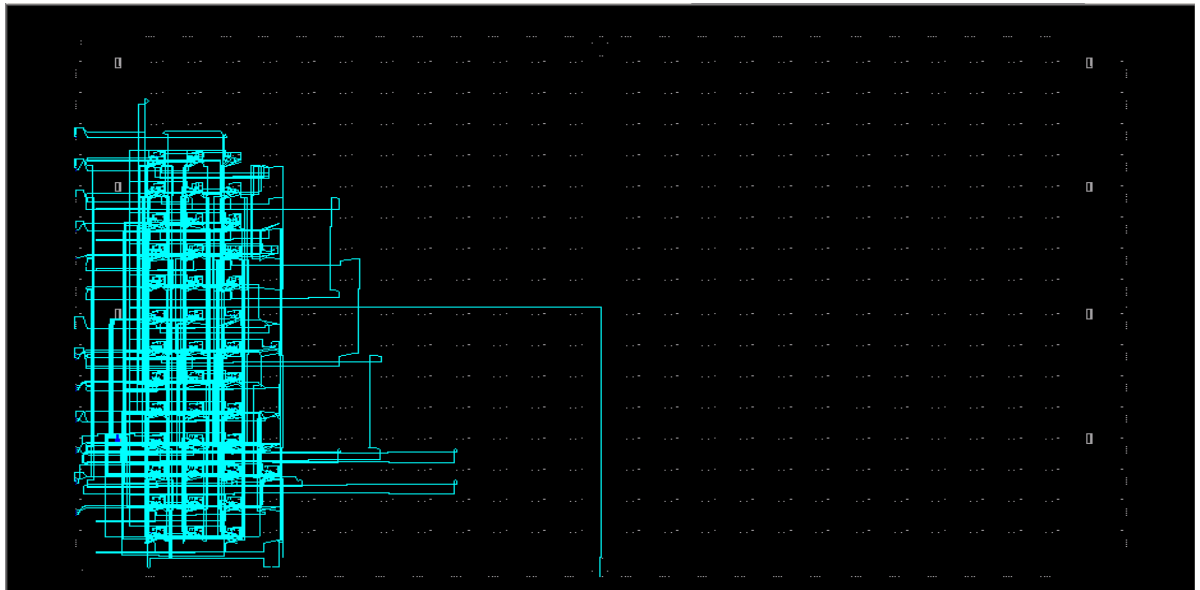


Figure 2: FPGA Editor View of a PicoBlaze Macro in an XC2S50E Spartan-IIE Device

In the larger devices, the PicoBlaze module is virtually free (Figure 3). The potential to place multiple PicoBlaze modules within a single design is obvious. Whenever a non-time-critical complex state machine is required, this macro is easy to insert and greatly simplifies the design.

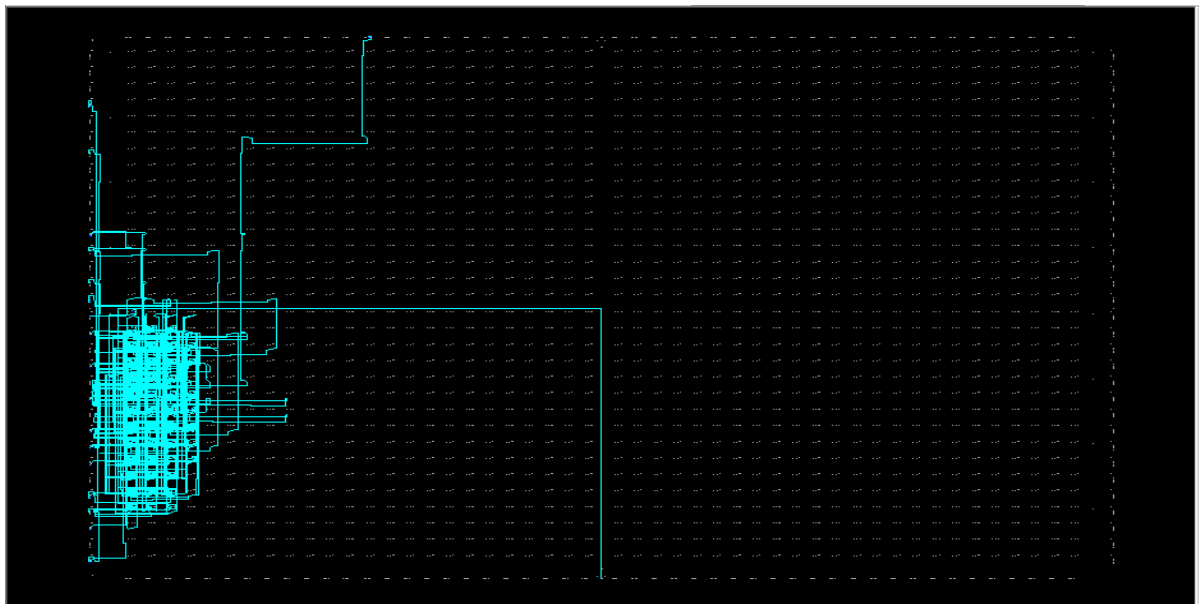


Figure 3: FPGA Editor View of a PicoBlaze Macro in an XC2S300E Spartan-IIE Device

PicoBlaze Resource Information

The following device resource information is taken from the ISE reports for the PicoBlaze macro in an XC2S50E device. The reports reveal the features that are utilized and the efficiency of the macro. The 76 “slices” reported by the map process in this case can reduce to the minimum of 66 “slices” when greater packing is used to fit a complete design into a device.

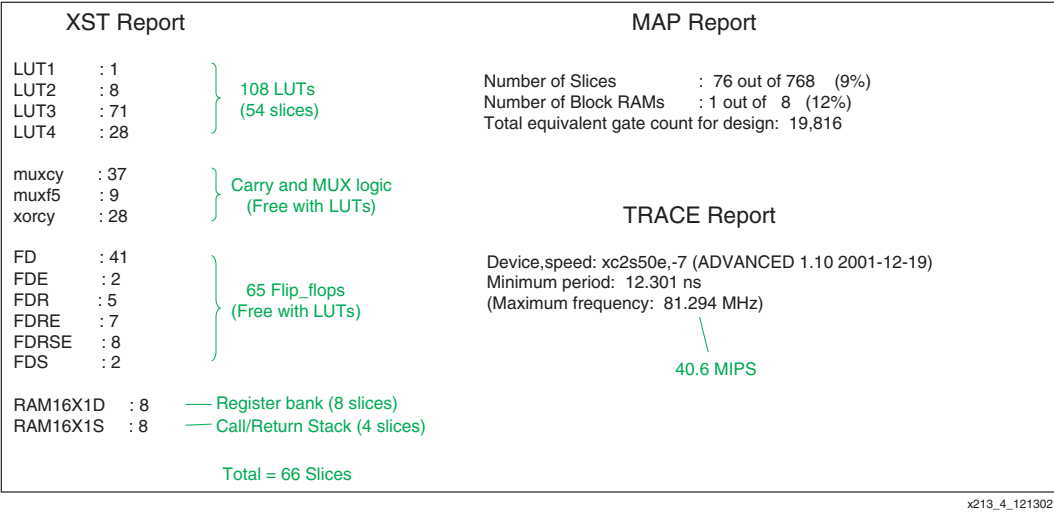


Figure 4: Device Resource Information

PicoBlaze Architecture

Figure 5 shows the PicoBlaze architecture.

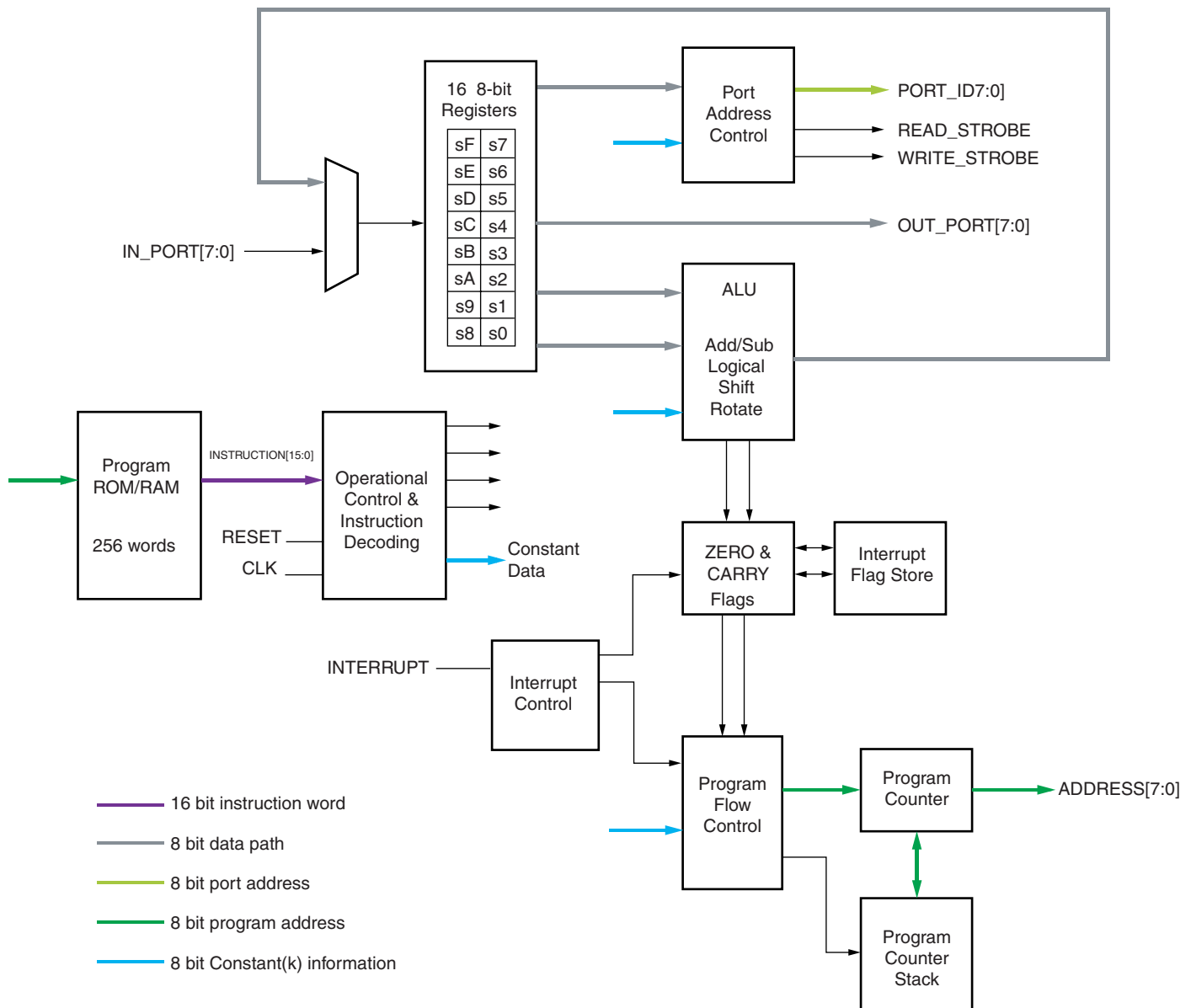


Figure 5: PicoBlaze Architecture

X213_05_012703

PicoBlaze Feature Set

General-Purpose Registers

The feature set includes 16 general-purpose 8-bit registers, specified as s0 to sF (can be renamed in the assembler). All register operations are completely flexible, with no registers reserved for special tasks or given any priority over any other register. No accumulator exists as any register can be adopted for use as an accumulator.

Arithmetic Logic Unit

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 can be specified or a constant 8-bit value can be supplied. The ability to specify any constant value with no penalty to the program size or performance enhances the simple instruction set. To clarify, the ability to "ADD 1" is the equivalent of a dedicated INCREMENT operation. For operations requiring more than eight bits, addition and subtraction operations have an option to include CARRY. Bit-wise operators (LOAD, AND, OR, XOR) provide the ability to manipulate and test values. There is also a comprehensive Shift and Rotate group.

Flags Program Flow Control

The ALU operation results affect the ZERO and CARRY flags. This information determines the execution sequence of the program using conditional and non-conditional program flow control instructions. 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 15 levels, more than adequate for the program size supported.

Reset

The RESET input forces the processor back into the initial state. The program executes from address 00 and interrupts are disabled. The status flags and CALL/RETURN stack are also reset. Note that the register contents are not affected.

Input/Output

The PicoBlaze module has 256 input ports and 256 output ports. An 8-bit address value provided on the PORT_ID bus together with READ_STROBE or WRITE_STROBE signals indicates the accessed port. The port address can be either supplied in the program as an absolute value, or specified indirectly as the contents of any of the 16 registers. Indirect addressing is ideal when accessing a block of memory either constructed from block or distributed RAM within or external to the Virtex-II device.

During an INPUT operation, the value provided at the input port is transferred into any of the 16 registers. An input operation is indicated by a READ_STROBE output pulse. Although using this signal in the design input interface logic is not always vital, it indicates that data has been acquired by the PicoBlaze module.

During an OUTPUT operation, the contents of any of the 16 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 to ensure that only valid data is passed to external systems. Typically, WRITE_STROBE is used as a clock enable or write enable signal.

Interrupt

The processor 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 and is then under program control to be enabled and disabled as required.

An active interrupt forces the PicoBlaze macro to initiate a "CALL FF" (i.e., a subroutine call to the last program memory location) for the user to define a suitable course of action. Automatically, the interrupt process preserves the current ZERO and CARRY flag contents and disables any further interrupts. A special RETURNI command ensures that the end of an interrupt service routine restores the status of the flags and controls the enable of future interrupts.

Constant (k) Coded Values

The PicoBlaze module is in many ways a state 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 module
- 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.”

Constant 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 constant execution rate is of great value.

Constant Program Length

The program length is 256 instructions, conforming to the 256 x 16 format of a single Virtex-E or Spartan-IIIE block RAM. 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. See the **Hints and Tips** section for help with larger programs. Virtex-II devices enable programs up to 1024 instructions (see [XAPP627](#)).

Using the PicoBlaze Macro

The PicoBlaze macro is used principally in a VHDL design flow. It is provided as source VHDL (**kcpsm.vhd**), which has been written for optimum and predictable implementation in a Virtex-E or Spartan-II device. The code is suitable for implementation and simulation of the macro and has been developed and tested using XST for implementation and ModelSim™ for simulation. The code should not be modified in any way.

```
component kcpasm
  Port (
    address : out std_logic_vector(7 downto 0);
    instruction : in std_logic_vector(15 downto 0);
    port_id : out std_logic_vector(7 downto 0);
    write_strobe : out std_logic;
    out_port : out std_logic_vector(7 downto 0);
    read_strobe : out std_logic;
    in_port : in std_logic_vector(7 downto 0);
    interrupt : in std_logic;
    reset : in std_logic;
    clk : in std_logic);
end component;
```

Figure 6: VHDL Component Declaration of KCPSM

```

processor: kcpsm
  port map(
    address => address_signal,
    instruction => instruction_signal,
    port_id => port_id_signal,
    write_strobe => write_strobe_signal,
    out_port => out_port_signal,
    read_strobe => read_strobe_signal,
    in_port => in_port_signal,
    interrupt => interrupt_signal,
    reset => reset_signal,
    clk => clk_signal);

```

Figure 7: VHDL Component Instantiation of the KCPSM

Connecting the Program ROM

The principal method by which the PicoBlaze program ROM is used is in a VHDL design flow. The PicoBlaze assembler generates a VHDL file in which a block RAM and its initial contents are defined (see **Assembler Directives** for more detail). This VHDL file can be used for implementation and simulation of the processor. It has been developed and tested using XST for implementation and ModelSim for simulation.

```

component prog_rom
  Port (
    address : in std_logic_vector(7 downto 0);
    instruction : out std_logic_vector(15 downto 0);
    clk : in std_logic);
end component;

```

Figure 8: VHDL Component Declaration of Program ROM

```

program: prog_rom
  port map(
    address => address_signal,
    instruction => instruction_signal,
    clk => clk_signal);

```

Figure 9: VHDL Component Instantiation of Program ROM

To aid with development, a VHDL file called "**embedded_kcpsm.vhd**" is also supplied in which the PicoBlaze macro is connected to its associated block RAM program ROM. This entire module can be embedded in the design application, or simply used to cut and paste the component declaration and instantiation information into the user's own code.

Notes:

1. The name of the program ROM (shown as "**prog_rom**" in the above examples) depends on the name of the user's program. For example, if the user's program file was called "**phone.psm**," then the assembler generates a program ROM definition file called "**phone.vhd**."

Alternative Design Flows

Although the primary design flow is VHDL, the PicoBlaze module can be used in any design flow supported by Xilinx using the following files:

kcpsm.ngc

The NGC file provided was made by synthesizing the **kcpsm.vhd** file with XST (without inserting I/O buffers).

This file can be used as a “black box” in a design, and it will be merged with the rest of the user’s design during the translate phase (ngdbuild).

Note that buses are defined in the style IN_PORT<7:0> with individual signals in_port_0 through in_port_7.

prog_rom.coe

The COE file generated by the assembler is suitable for use with the Xilinx Core Generator.

The file defines the initial contents of a block ROM. The files generated by Core Generator can then be used as normal in the chosen design flow and connected to the PicoBlaze “black box” in the user’s design.

Notes:

1. It is recommended that “**embedded_kcpsm.vhd**” be used for the generation of an ECS schematic symbol.

Simulation

If the NGC file is used in the design flow, then some form of back annotated net list needs to be used for simulation of the design in order to fill in the “black box” details required by the user’s simulator.

PicoBlaze Instruction Set

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

1. “X” and “Y” refer to the definition of the storage registers “s” in range 0 to F.
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.

Program Control Group

```
JUMP aa
JUMP Z, aa
JUMP NZ, aa
JUMP C, aa
JUMP NC, aa
```

```
CALL aa
CALL Z, aa
CALL NZ, aa
CALL C, aa
CALL NC, aa
```

```
RETURN
RETURN Z
RETURN NZ
RETURN C
RETURN NC
```

Notes:

1. Call and Return supports a stack depth of up to 15.

Logical Group

```
LOAD sX, kk  
AND sX, kk  
OR sX, kk  
XOR sX, kk
```

```
LOAD sX, sY  
AND sX, sY  
OR sX, sY  
XOR sX, sY
```

Arithmetic Group

```
ADD sX, kk  
ADDCY sX, kk  
SUB sX, kk  
SUBCY sX, kk
```

```
ADD sX, sY  
ADDCY sX, sY  
SUB sX, sY  
SUBCY sX, sY
```

Shift and Rotate Group

```
SR0 sX  
SR1 sX  
SRX sX  
SRA sX  
RR sX
```

```
SL0 sX  
SL1 sX  
SLX sX  
SLA sX  
RL sX
```

Input/Output Group

```
INPUT sX, pp  
INPUT sX, (sY)
```

```
OUTPUT sX, pp  
OUTPUT sX, (sY)
```

Interrupt Group

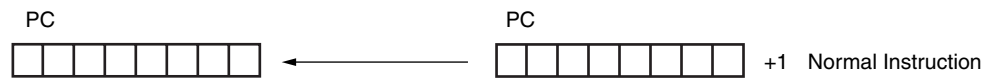
```
RETURNI ENABLE  
RETURNI DISABLE
```

```
ENABLE INTERRUPT  
DISABLE INTERRUPT
```

Program Control Group

JUMP

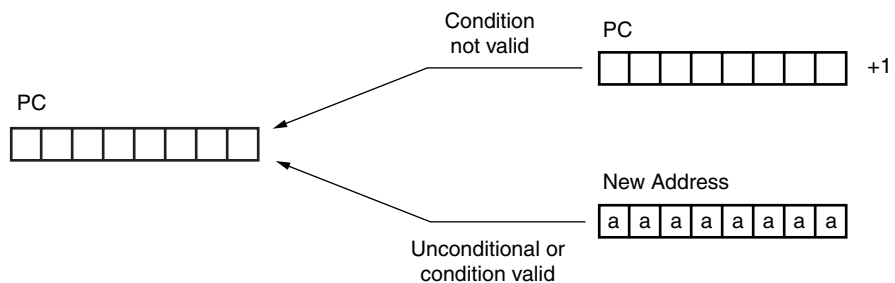
Under normal conditions, the program counter (PC) increments to point to the next instruction (Figure 10). The address space is fixed to 256 locations (00 to FF hex), making the program counter 8-bits wide. The top of the memory is FF hex and increments to 00.



x213_08_062100

Figure 10: Program Counter

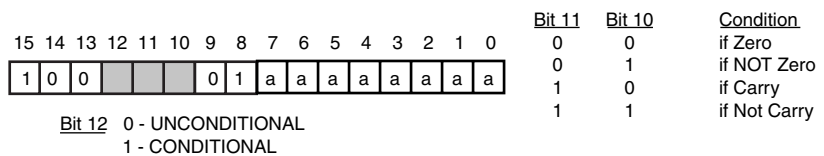
The JUMP instruction can be used to modify this sequence by specifying a new address. However, the JUMP instruction can be conditional. A conditional JUMP is only performed if a test performed on either the ZERO flag or CARRY flag is valid. The JUMP instruction has no effect on the status of the flags (Figure 11).



x213_09_062100

Figure 11: JUMP Instruction

Each JUMP instruction must specify the 8-bit address as a two-digit hexadecimal value. The assembler supports labels to simplify this process (Figure 12).



x213_12_120302

Figure 12: JUMP Instruction Specification

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 can also be conditional. In addition to supplying a new address, the CALL instruction also causes the current program counter (PC) value to be pushed onto the program counter stack. The CALL instruction has no effect on the status of the flags (Figure 13).

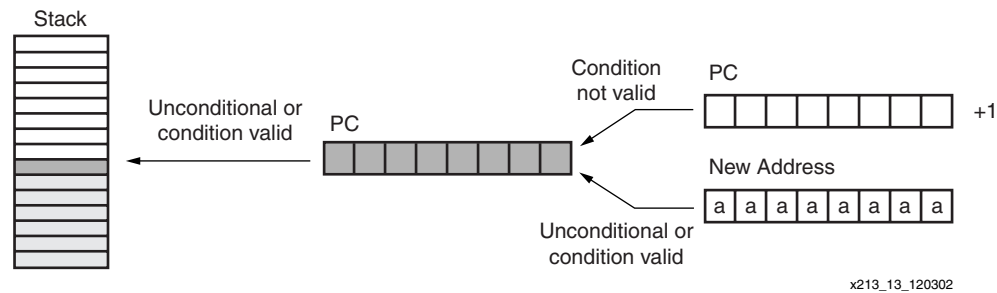


Figure 13: CALL Instruction

The program counter stack supports a depth of 15 address values, enabling nested CALL sequences to a depth of 15 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 cyclic buffer. When the stack is full, it overwrites the oldest value. Hence, it is not necessary to reset the stack pointer when performing a software reset. This also explains why there are no instructions to control the stack and why no program memory needs to be reserved for the stack. 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. (Figure 14).

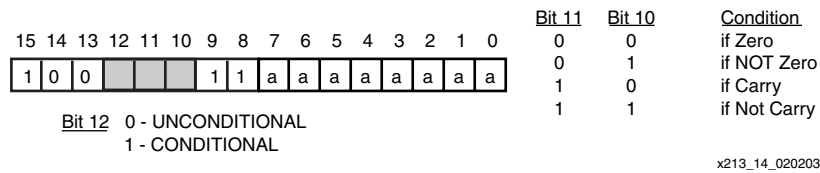


Figure 14: CALL Instruction Specification

RETURN

The RETURN instruction is the complement to the CALL instruction. The RETURN instruction is also conditional. In Figure 15, the new program counter (PC) value is formed internally by incrementing the last value on the program address stack, ensuring that the program executes the instruction following the CALL instruction which resulted in the subroutine. The RETURN instruction has no effect on the status of the flags.

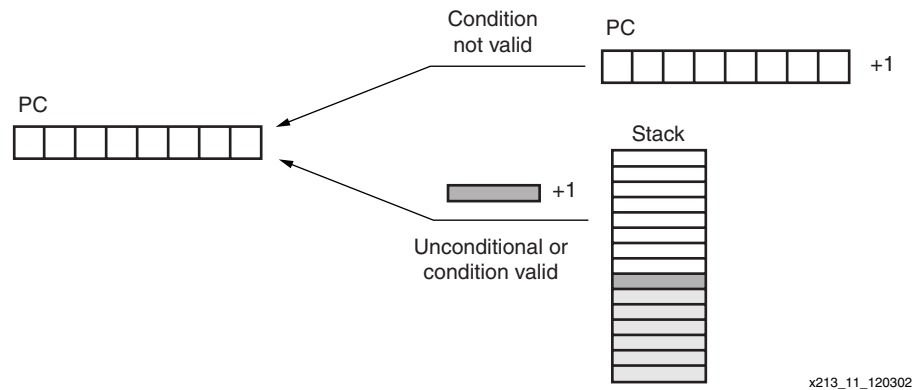


Figure 15: RETURN Instruction

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 (Figure 16). The cyclic implementation of the stack continues to provide values for RETURN instructions which cannot be defined. Each RETURN only specifies the condition for flag tests.

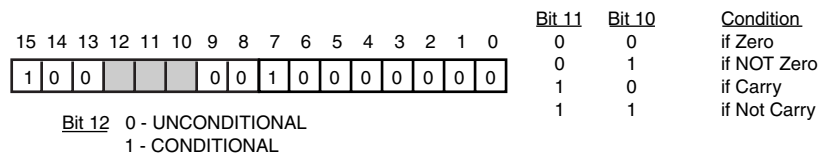


Figure 16: RETURN Instruction Specification

Interrupt Group

RETURNI

The RETURNI instruction (Figure 17) 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 address stored 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 and DISABLE as an operand.

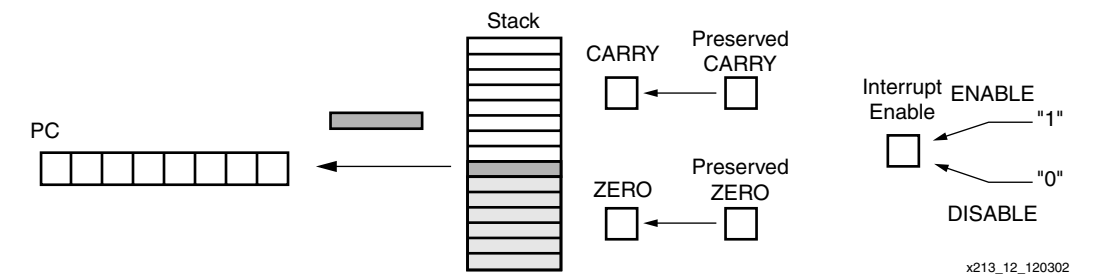


Figure 17: RETURNI Instruction

The programmer must ensure that a RETURNI (Figure 18) is only performed in response to an interrupt. Each RETURNI must specify if a further interrupt is enabled or disabled.

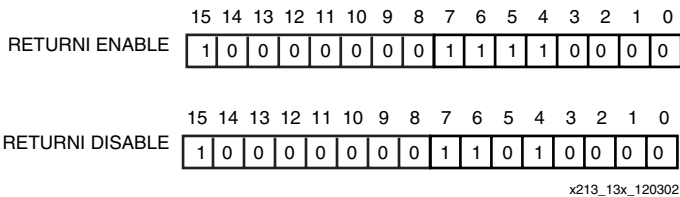


Figure 18: RETURNI Instruction Specification

ENABLE INTERRUPT and DISABLE INTERRUPT

These instructions are used to set and reset the INTERRUPT ENABLE flag (Figure 19). Before using ENABLE INTERRUPT (Figure 20), a suitable interrupt routine must be associated with the interrupt address vector (FF). Never enable interrupts while performing an interrupt service routine.

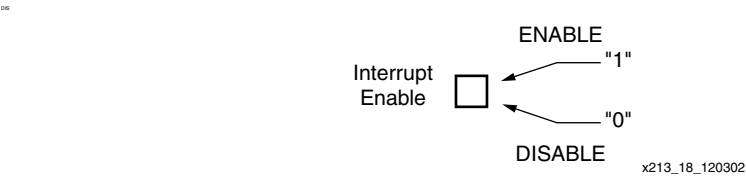


Figure 19: ENABLE/DISABLE INTERRUPT Instruction

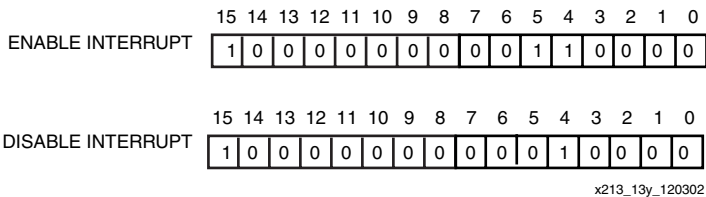


Figure 20: ENABLE/DISABLE INTERRUPT Instruction Specification

Logical Group

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 (Figure 21).

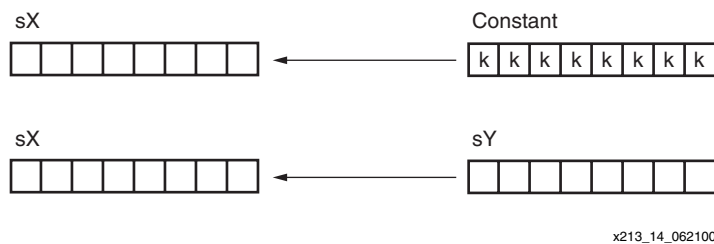


Figure 21: LOAD Instruction

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 Loading any register with its own contents achieves nothing and hence is a NO OPERATION consuming two clock cycles. This may be used to form a delay in the program.

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

Each LOAD instruction (Figure 22) must specify the first operand register as “s” followed by a single hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.

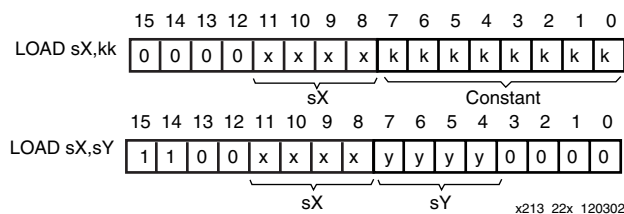


Figure 22: LOAD Instruction Specification

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 (Figure 23). Flags are affected by this operation. The AND operation can be used to perform tests on the contents of a register. The status of the ZERO flag then controls the flow of the program

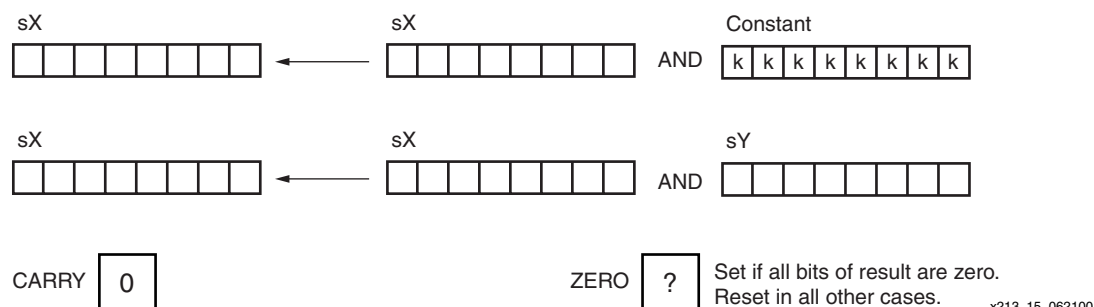


Figure 23: AND Instruction

Each AND instruction (Figure 24) must specify the first operand register as “s” followed by a single hexadecimal digit. This register also forms the destination for the result. The second operand specifies a second register value in a similar way, or specifies an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.

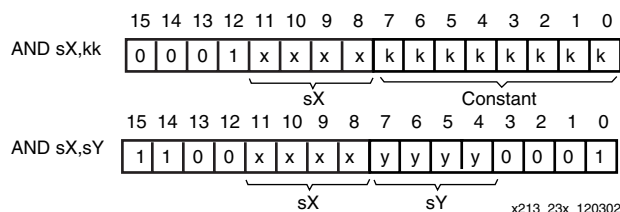


Figure 24: AND Instruction Specification

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 the result of this operation. A second operand is also any register, or an 8-bit constant value (Figure 25). Flags are affected by the OR operation. The OR instruction provides a way to force setting any bit of the specified register, which can be used to form control signals.

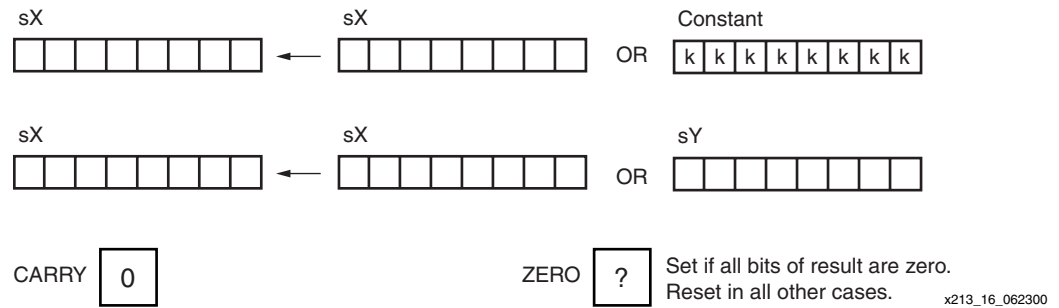


Figure 25: OR Instruction

Each OR instruction (Figure 26) must specify the first operand register as “s” followed by a single hexadecimal digit. This register also forms the destination for the result. The second operand must then specify a second register value in a similar way, or specify an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.

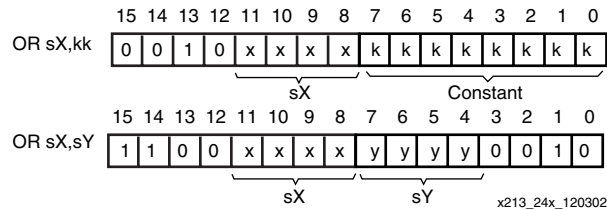


Figure 26: OR Instruction Specification

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. Flags are affected by this operation (Figure 27). The XOR operation can be used to invert bits contained in a register, which is used in forming control signals.

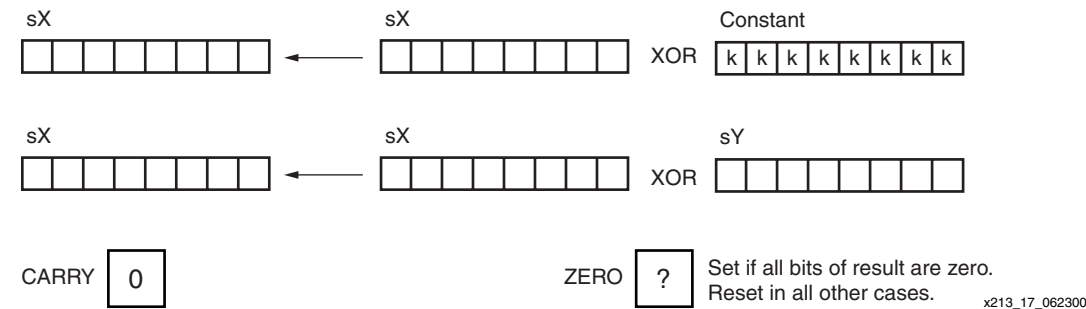


Figure 27: XOR Instruction

Each XOR instruction (Figure 28) must specify the first operand register as “s” followed by a single hexadecimal digit. This register also forms the destination for the result. The second operand must then specify a second register value in a similar way, or specify an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.

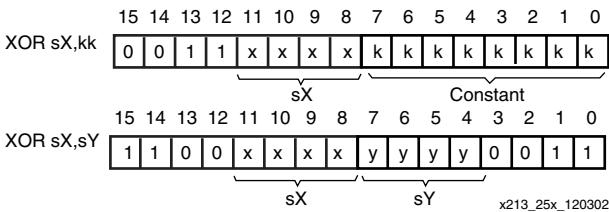
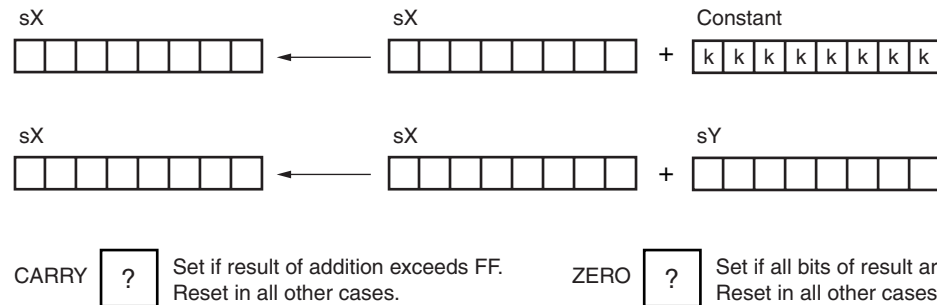


Figure 28: XOR Instruction Specification

Arithmetic Group

ADD

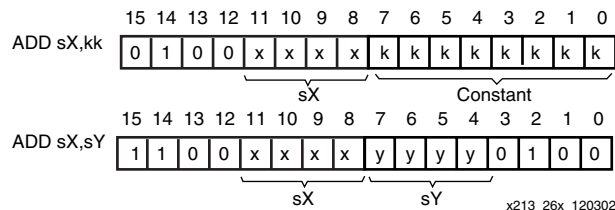
The ADD instruction performs an 8-bit addition of two operands. The first operand is any register, and it is this register that is assigned the result of the operation. A second operand is also any register, or an 8-bit constant value (Figure 29). Flags are affected by this operation. Note that this instruction does not use the CARRY as an input, and hence, there is no need to condition the flags before use. The ability to specify any constant is useful in forming control sequences or counters.



x215_18_062300

Figure 29: ADD Instruction

Each ADD instruction (Figure 30) must specify the first operand register as “s” followed by a single hexadecimal digit. This register forms the destination for the result. The second operand must then specify a second register value in a similar way, or specify an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.



x213_26x_120302

Figure 30: ADD Instruction Specification

ADDCY

The ADDCY instruction performs an 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 (Figure 31). Flags are affected by this operation. The ADDCY operation is used in the formation of adder and counter processes exceeding eight bits.

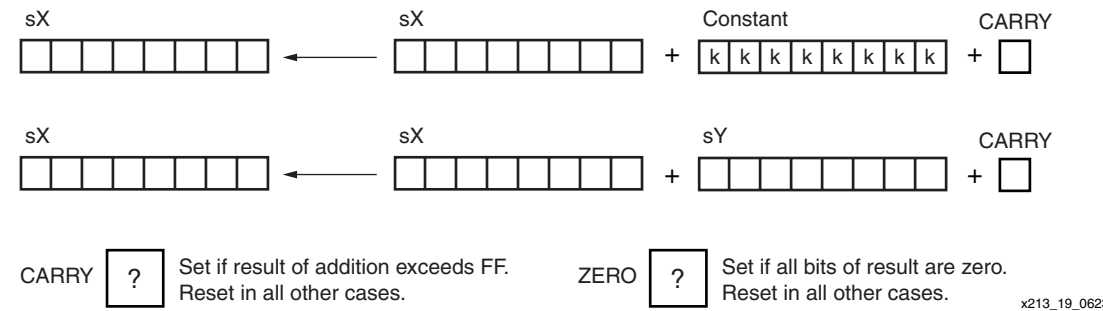


Figure 31: ADDCY Instruction

Each ADDCY instruction (Figure 32) must specify the first operand register as “s” followed by a single hexadecimal digit. This register also forms the destination for the result. The second operand must then specify a second register value in a similar way, or specify an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.

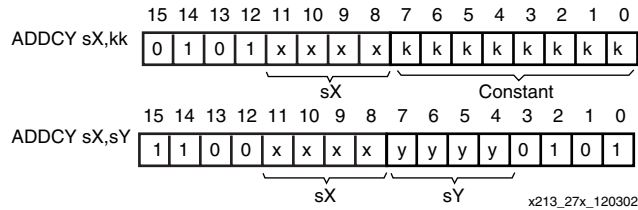


Figure 32: ADDCY Instruction Specification

SUB

The SUB instruction performs an 8-bit 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 (Figure 33). Flags are affected by this operation. Note that this instruction does not use the CARRY as an input and, hence, there is no need to condition the flags before use. The CARRY flag indicates when an underflow has occurred. For example, if “s5” contains 27 hex and the instruction SUB s5,35 is performed, then the stored result is F2 hex and the CARRY flag is set.

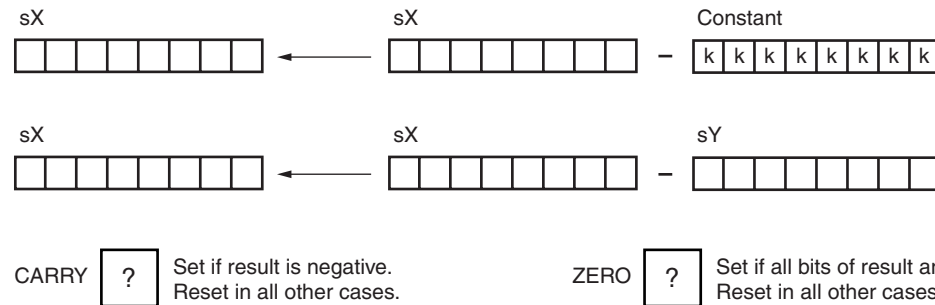


Figure 33: SUB Instruction

Each SUB instruction (Figure 34) must specify the first operand register as “s” followed by a single hexadecimal digit. This register also forms the destination for the result. The second operand must then specify a second register value in a similar way, or specify an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.

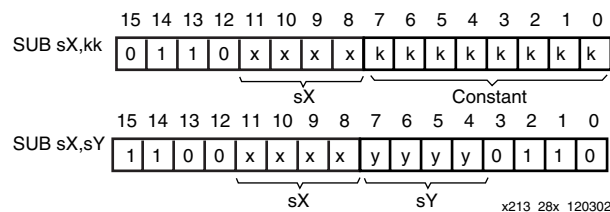


Figure 34: SUB Instruction Specification

SUBCY

The SUBCY instruction performs an 8-bit 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 (Figure 35). Flags are affected by this operation. The SUBCY operation is used in the formation of subtract and down-counter processes exceeding 8 bits.

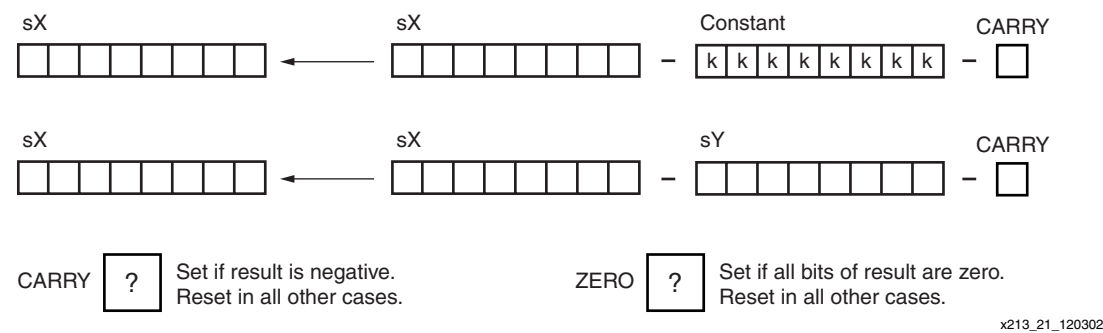


Figure 35: SUBCY Instruction

Each SUBCY instruction (Figure 36) must specify the first operand register as “s” followed by a single hexadecimal digit. This register also forms the destination for the result. The second operand must then specify a second register value in a similar way, or specify an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.

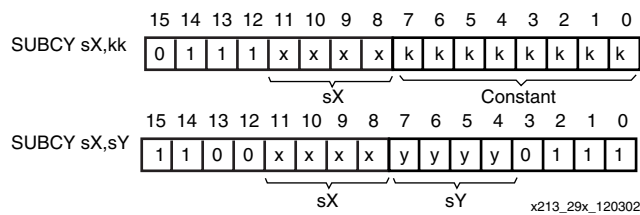
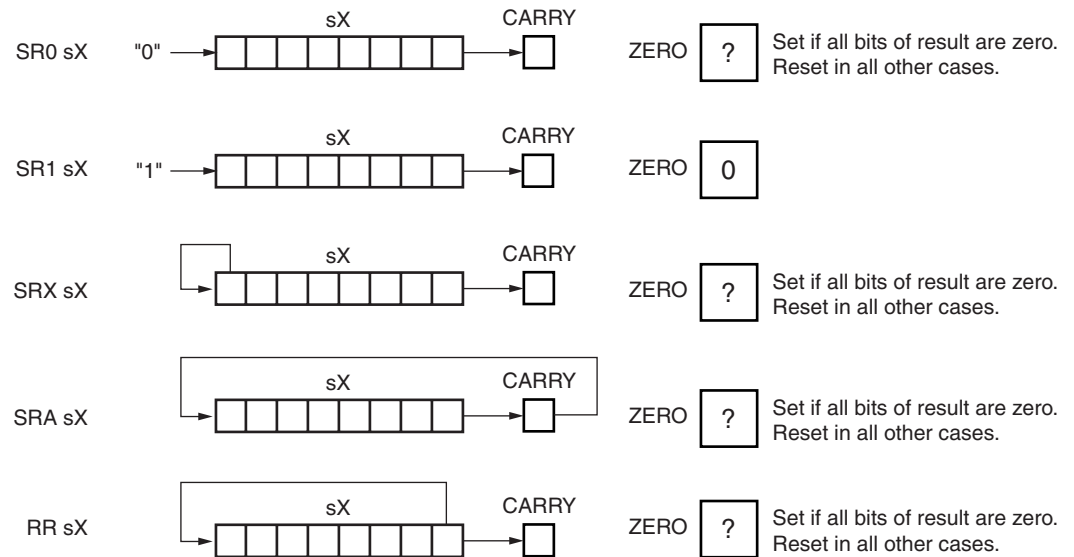


Figure 36: SUBCY Instruction Specification

Shift and Rotate Group

SR0, SR1, SRX, SRA, RR

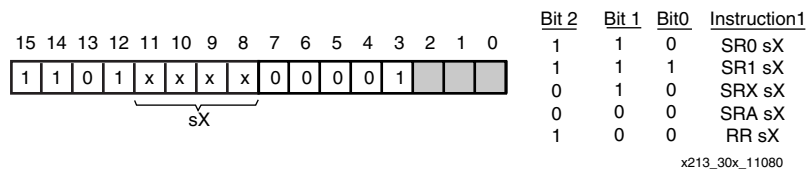
The shift and rotate right group all modify the contents of a single register (Figure 37). All instructions in the group have an effect on the flags.



x213_22_090100

Figure 37: Right Shift Register Instructions

Each instruction must specify the register as "s" followed by a single hexadecimal digit (Figure 38). The assembler supports register naming to simplify programming.

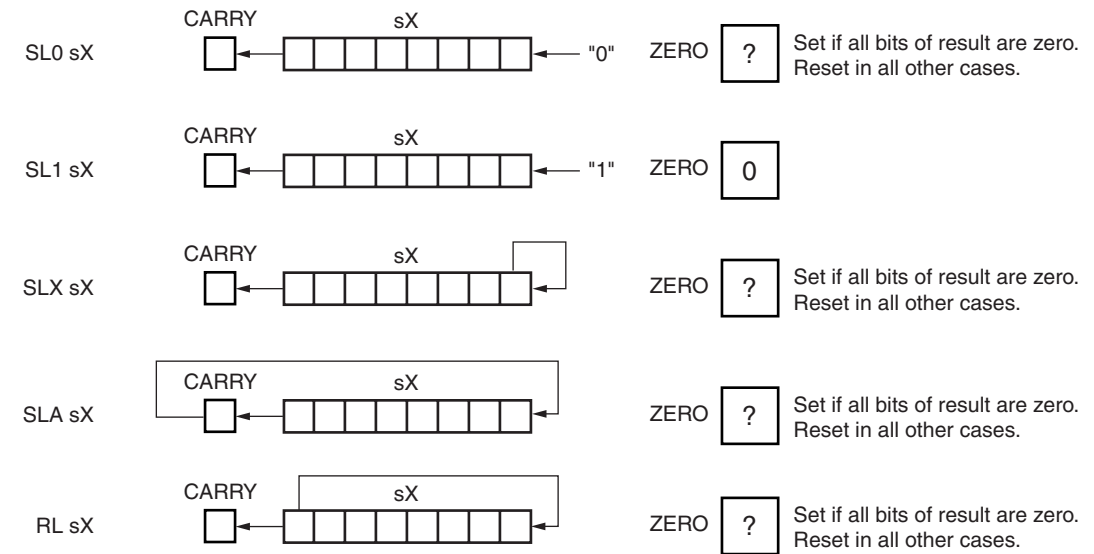


x213_30x_11080

Figure 38: Right Shift Register Instruction Specification

SL0, SL1, SLX, SLA, RL

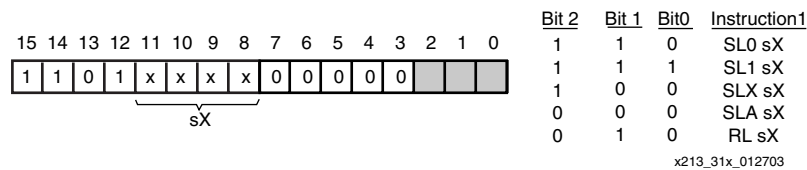
The shift and rotate left group all modify the contents of a single register (Figure 39). All instructions in the group have an effect on the flags.



x213_23_062300

Figure 39: Left SHIFT Register Instructions

Each instruction must specify the register as “s” followed by a single hexadecimal digit (Figure 40). The assembler supports register naming to simplify programming.



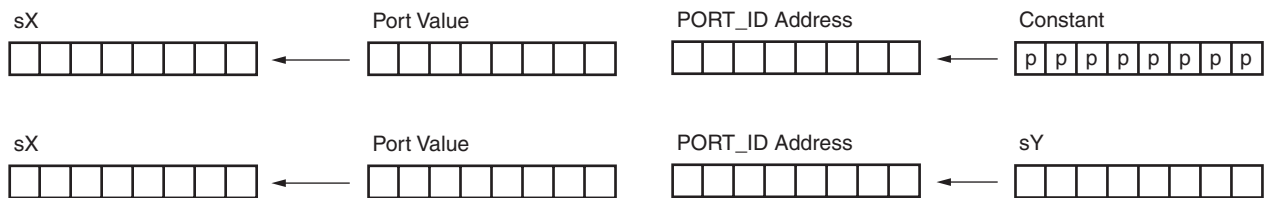
x213_31x_012703

Figure 40: Left SHIFT Register Instruction Specification

Input and Output Group

INPUT

The INPUT instruction enables data values external to the PicoBlaze module to be transferred into any one of the internal registers (Figure 41). The port address (in the range 00 to FF) is defined by a constant value, or indirectly as the contents of the any other register. The flags are not affected by this operation.

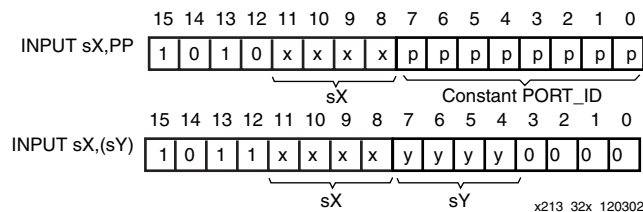


x213 24 110502

Figure 41: INPUT Instruction

The user interface logic is required to decode the PORT_ID port address value and supply the correct data to the IN_PORT. The READ_STROBE is set during an input operation (see **READ and WRITE STROBES**), but is not vital for the interface logic to decode this strobe in most applications. However, it can be useful for determining when data has been read, such as when reading a FIFO buffer.

Each INPUT instruction (Figure 42) must specify the destination register as “s” followed by a single hexadecimal digit. It must then specify the input port address using a register value in a similar way, or specify an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.



x213_32x_120302

Figure 42: INPUT Instruction Specification

OUTPUT

The OUTPUT instruction enables the contents of any register to be transferred to logic external to the PicoBlaze module. The port address (in the range 00 to FF) is defined by a constant value, or indirectly as the contents of the any other register (Figure 43). The flags are not affected by this operation.

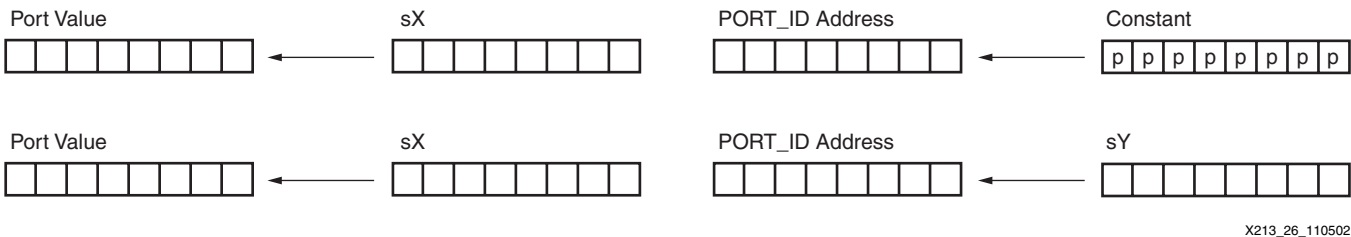


Figure 43: OUTPUT Instruction

The user interface logic is required to decode the PORT_ID port address value and capture the data provided or the OUT_PORT. The WRITE_STROBE is set during an output operation (see READ and WRITE STROBES) and should be used to clock enable the capture register (or write enable a RAM).

Each OUTPUT instruction (Figure 44) must specify the source register as “s” followed by a single hexadecimal digit. It must then specify the output port address using a register value in a similar way, or specify an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.

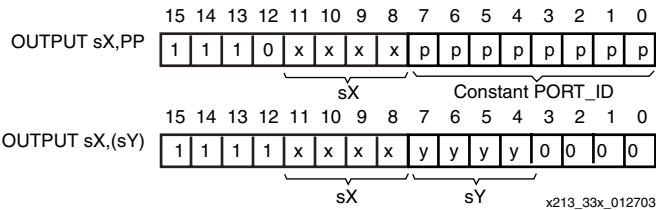


Figure 44: OUTPUT Instruction Specification

READ and WRITE STROBES

These pulses are used by external circuits to confirm input and output operations. In the waveforms (Figure 45), it is assumed that the content of register sE is 47, and the content of register sA is 42.

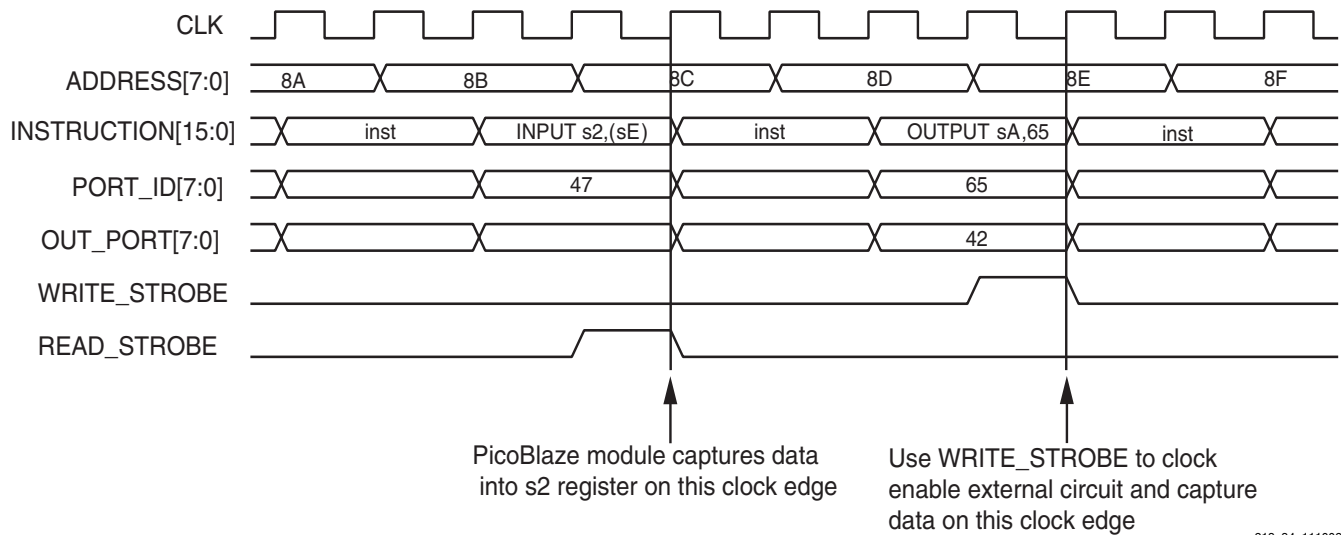


Figure 45: READ and WRITE Strokes

PORT_ID[7:0] is provided with the full two clock cycles to be decoded by external logic. The WRITE_STROBE is provided on the second clock cycle to confirm an active write by the PicoBlaze module. In most cases, the READ_STROBE is not utilized by the external decoding logic, but again occurs in the second cycle and indicates the actual clock edge on which data is read into the specified register.

Notes:

1. For timing critical designs, timing specifications can allow two clock cycles for PORT_ID and data paths, and only the strobes need to be constrained to a single clock cycle.

RESET

The PicoBlaze module contains an internal reset control circuit to ensure the correct start up of PicoBlaze following device configuration or global reset. This reset can also be activated within the user's design.

The PicoBlaze reset is sampled synchronous to the clock and used to form a controlled internal reset signal which is distributed locally as required. A small 'filter' circuit (see [Figure 46](#)) ensures that the release of the internal reset is clean and controlled. The reset input can be tied to logic 0 if not required, and the filter is still used to ensure correct power-up sequence ([Figure 47](#) and [Figure 48](#)).

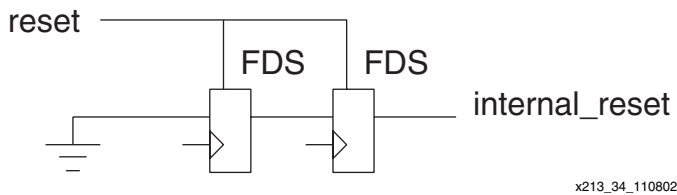


Figure 46: PicoBlaze Filter Circuit

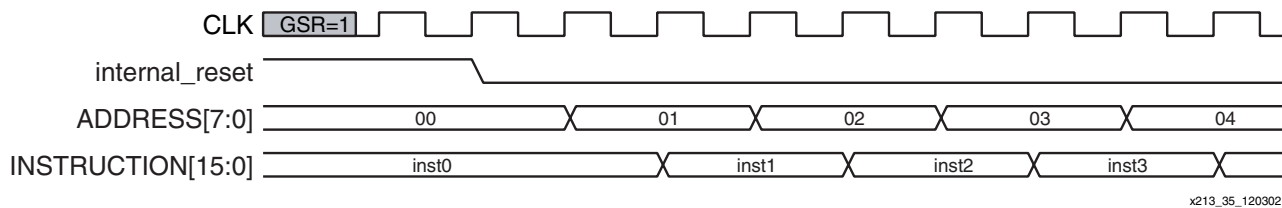


Figure 47: Release of Reset after Configuration

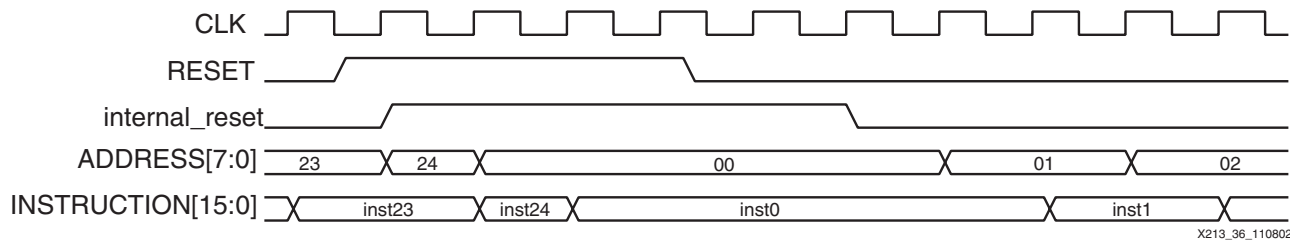


Figure 48: Application of User Reset Input

PicoBlaze Assembler

The PicoBlaze Assembler ([Figure 49](#)) is provided as a simple DOS executable file together with two template files. The files **KCP_{SM}.EXE**, **ROM_{form}.vhd**, and **ROM_{form}.coe** should be copied into the user's working directory.

Programs are best written with either the standard Notepad or Wordpad tools. The file is saved with a **.psm** file extension (8-character name limit).

Open a DOS box and navigate to the working directory. Then run the assembler **kcpsm <filename> [.psm]** to assemble the program. It all happens very fast.

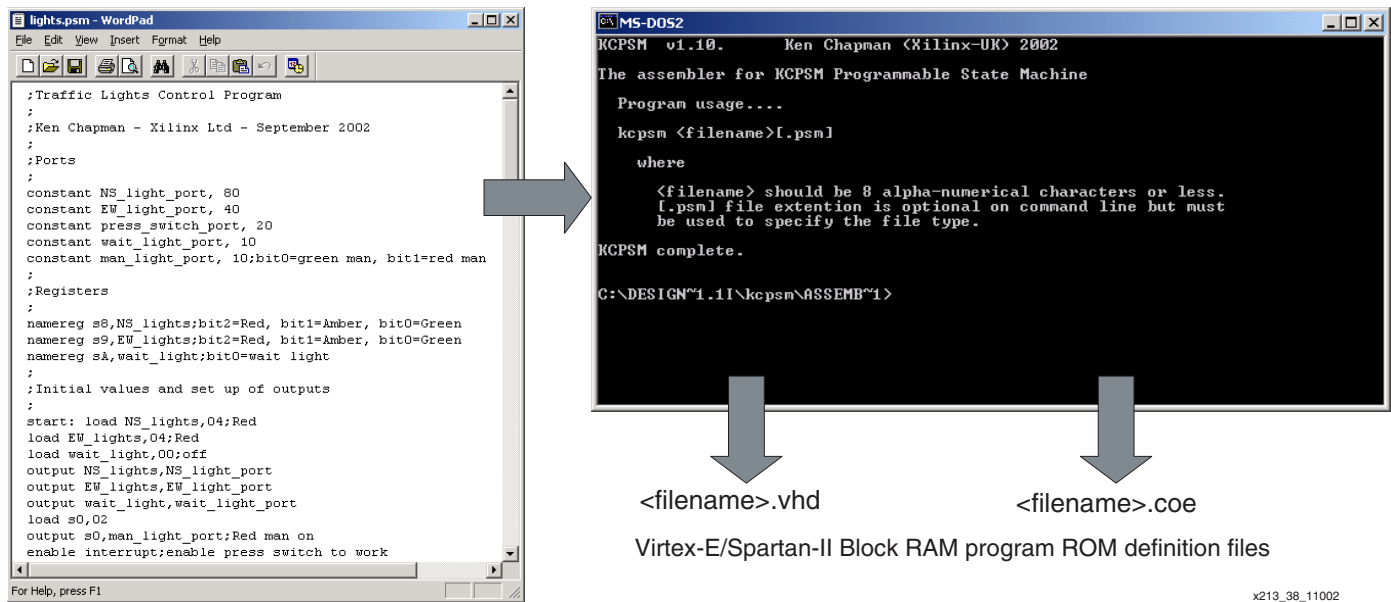


Figure 49: PicoBlaze Assembler

Assembler Errors

The assembler stops as soon as an error is detected (Figure 50). A short message is displayed to help determine the reason for the error. The assembler also displays the line it was analyzing when the problem was detected. The user should fix each reported problem in turn and re-execute the assembler.

Since the execution of the assembler is very fast, the display appears to be immediate. The user can review everything that the assembler has written to the screen, by redirecting the DOS output to a text file using: **kcpsm <filename> [.psm] > screen_dump.txt**

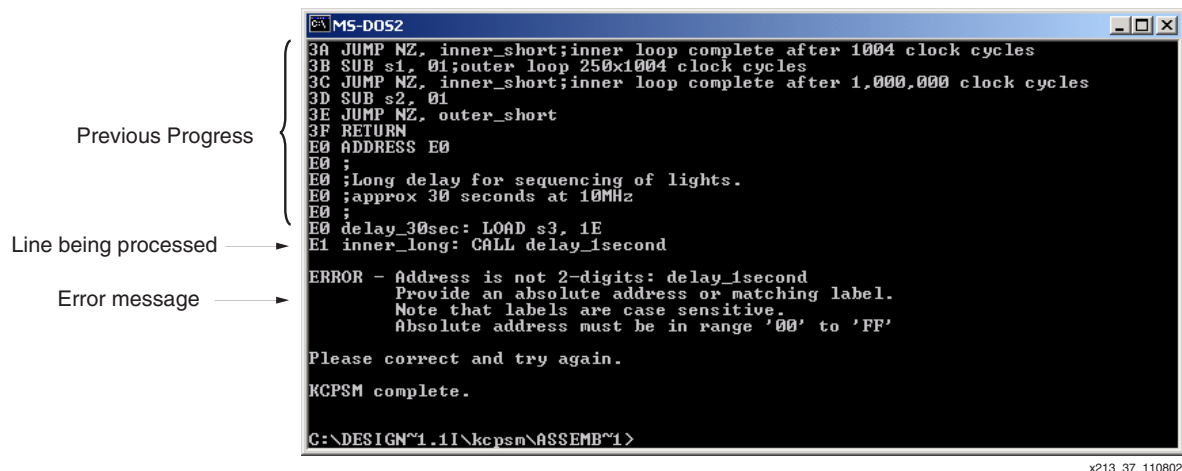
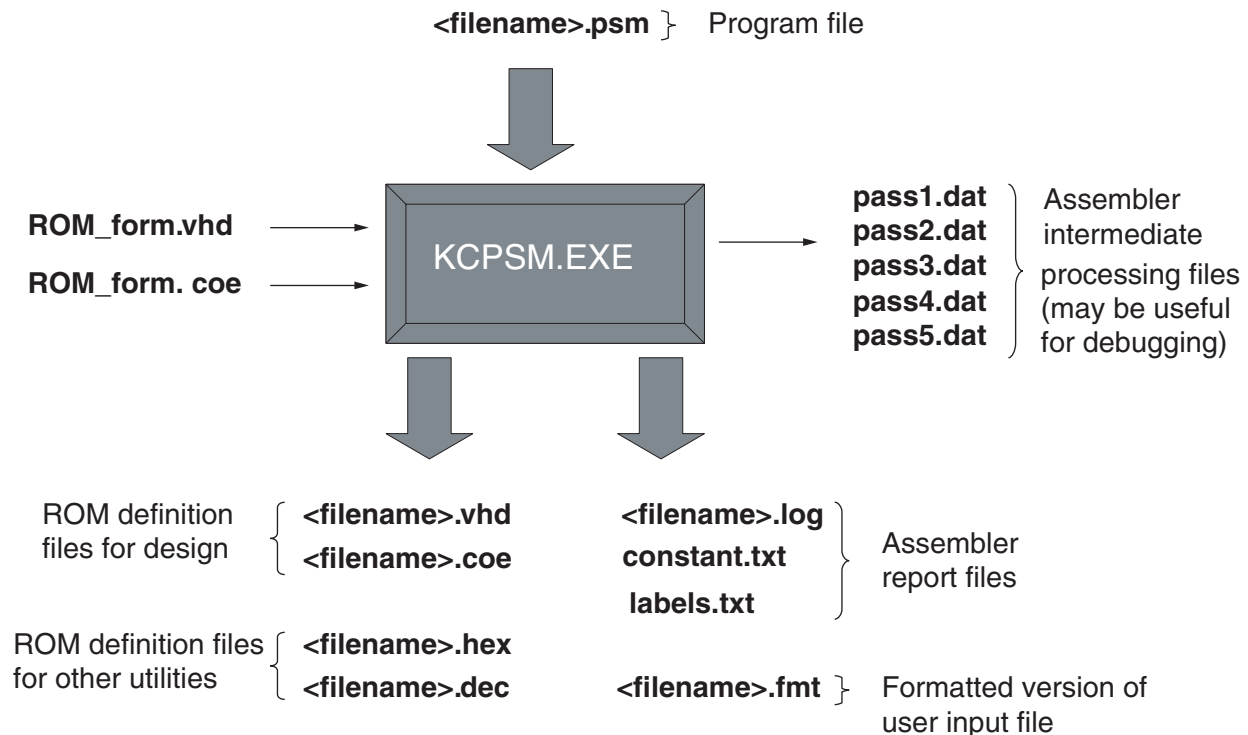


Figure 50: Assembler Error Display

Assembler Files

The PicoBlaze Assembler actually reads three input files and generates thirteen output files. See [Figure 51](#) for a diagram of these files.



x213_49_120302

Figure 51: Files Associated with Assembler

The *hex* and *dec* files provide the program ROM contents in unformatted hexadecimal and decimal for conversion to other formats not supported directly by the assembler. There is no further description in this application note.

ROM_form.vhd File

This file provides the template for the VHDL file generated by the assembler and suitable for synthesis and simulation. This file is provided with the assembler and must be placed in the working directory.

The supplied **ROM_form.vhd** template file defines a single-port block RAM for Virtex-E or Spartan-II devices configured as a ROM. The user can adjust this template to define the type of memory desired. The template supplied includes additional notes on how the template works.

The assembler reads the **ROM_form.vhd** template and simply copies the information into the output file **<filename>.vhd**. There is no checking of syntax, so any alterations are the responsibility of the user. See [Figure 52](#).

```

entity {name} is
    Port (
        address : in std_logic_vector(7 downto 0);
        instruction : out std_logic_vector(15 downto 0);
        clk : in std_logic);
    end {name};
--
architecture low_level_definition of {name} is
    .
    .
    attribute INIT_00 of ram_256_x_16 : label is "{INIT_00}";
    attribute INIT_01 of ram_256_x_16 : label is "{INIT_01}";
    attribute INIT_02 of ram_256_x_16 : label is "{INIT_02}";

```

Figure 52: ROM_form.vhd

The template contains some special text strings surrounded by {} brackets. These are {begin template}, {name}, and a whole family of initialization identifiers, such as {INIT_01}. The assembler uses {begin template} to identify where the VHDL definition begins. It then intercepts and replaces all other special strings with the appropriate information. {name} is replaced with the name of the input program .psm file.

ROM_form.coe File

This file provides the template for the coefficient file generated by the assembler and suitable for the Core Generator. This file is provided with the assembler and must be placed in the working directory.

The supplied **ROM_form.coe** template file defines a Dual Port Block RAM for Virtex-E or Spartan-II devices in which the A-port is read only and the B-port is read/write. The user can adjust this template to define the type of memory for the Core Generator to implement.

The assembler reads the **ROM_form.coe** template and simply copies the information into the output file <filename>.coe. There is no checking of syntax, so any alterations are the responsibility of the user.

The template can contain the special text string {name} which the assembler intercepts and replaces with the name of the program file. In **Figure 53**, {name} has been replaced with "simple."

```

ROM_form.coe
component_name={name};
width_a=16;
depth_a=256;
.
.
memory_initialization_radix=16;
global_init_value=0000;
memory_initialization_vector=

                                ↓ KCPSM Assembler

<filename>.coe
component_name=simple;
width_a=16;
depth_a=256;
.
.
memory_initialization_radix=16;
global_init_value=0000;
memory_initialization_vector=
039F, 0342, 059F, 0542, C370, C3A0, C570, C5A0, ...

```

x213_50_110402

Figure 53: ROM_form.coe

It is vital that the last line of the template contains the key words:

memory_initialization_vector=

These words are used by the Core Generator to identify the data values that follow. The assembler appends the 256 values required. Indeed, the template could simply contain this one line provided the Core Generator GUI is used to set up all other parameters.

<filename>.fmt File

When a program passes through the assembler, additional files to the .vhd and .coe files are produced to assist the programmer. One of these files is called <filename>.fmt, which is the original program but in a formatted state. Looking at this file is an easy way for the programmer to see that everything has been interpreted correctly. The <filename>.fmt file:

- Formats labels and comments
- Puts all commands in upper case
- Correctly spaces operands
- Gives registers an 'sX' format
- Converts hex constants to upper case

See [Figure 54](#).

The <filename>.fmt file will typically be renamed <filename>.psm by the user so that this nicely formatted version can be adopted as the program file.

```

<filename>.psm
constant max_count, 18;count to 24 hours
namereg s4,counter_reg;define register for counter
constant count_port, 12
start: load counter_reg,00;initialise counter
loop:output counter_reg,count_port
add counter_reg,01;increment
load s0,counter_reg
sub s0,max_count;test for max value
jump nz,loop;next count
jump start;reset counter

```

KCPSM Assembler

```

<filename>.fmt
        CONSTANT max_count, 18           ;count to 24 hours
        NAMEREG s4, counter_reg         ;define register for counter
        CONSTANT count_port, 12
start:  LOAD counter_reg, 00             ;initialise counter
loop:   OUTPUT counter_reg, count_port
        ADD counter_reg, 01             ;increment
        LOAD s0, counter_reg
        SUB s0, max_count               ;test for max value
        JUMP NZ, loop                  ;next count
        JUMP start                     ;reset counter

```

x213_52_120302

Figure 54: <filename>.fmt File

<filename>.log File

The .log file (Figure 55) provides the user with the most detail about the assembly process which has been performed. This is where the user can observe how each instruction and directive has been used. Address and opcode values are associated with each line of the program and the actual values of addresses, registers, and constants defined by labels are specified.

x213 53 12030

y213 54 1203

at the start of the assembly process. If there is an error detected by the assembler, the .dat files are only complete until the point of the last successful processing.

The .dat. files segment the information from each line into the different fields. Each pass resolves more information. The example shown here is related to the line:

```
ADD counter_reg, 01 ;increment
```

It can be seen that **pass1.dat** has purely segmented the fields of the line. In the final **pass5.dat**, the assembler has resolved all the relevant information.

```
Part of pass1..dat

    LABEL-
    INSTRUCTION-add
    OPERAND1-counter_reg
    OPERAND2-01
    COMMENT-;increment

Part of pass5..dat

    ADDRESS-02
    LABEL-
    FORMATTED-ADD counter_reg, 01
    LOGFORMAT-ADD counter_reg[s4], 01
    INSTRUCTION-ADD
    OPERAND1-counter_reg
    OP1 VALUE-s4
    OPERAND2-01
    OP2 VALUE-01
    COMMENT-;increment
```

x2113_55_012703

Figure 57: **pass.dat** Files

Program Syntax

Probably the best way to understand what is and what is not valid syntax is to look at the examples and try the assembler. However, some simple rules are of assistance from the beginning. To ensure that the correct program syntax is used, the following suggestions are recommended:

No blank lines. A blank line is ignored by the assembler and removed from any formatted files. To keep a line, use a blank comment (a semicolon).

Comments. Any item on a line following a semi-colon (;) is ignored by the assembler. Concise comments should be used to keep the program manageable and make it easy to print out programs and log files.

Registers. All registers must be defined as the letter “s” immediately followed by a single hexadecimal character in the range of 0 to F. The assembler accepts any mixture of upper and lower case characters and automatically converts them to the “sX” format, where “X” is one of the following: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Constants. A constant is specified in the form of a two-digit hexadecimal value (range 00 to FF). The assembler accepts any mixture of upper and lower case characters and automatically converts them to upper case.

Labels. Labels are any user-defined text string and are *case sensitive* for additional flexibility. No spaces are allowed, but the underscore character is supported. Valid characters are 0 to 9, a to z, and A to Z. Labels should be reasonably concise to keep the program formatting clean. Labels which could be confused with hexadecimal addresses and constants or register specifications are rejected by the assembler.

Line Labels. A label used to identify a program line for reference in a JUMP or CALL instruction should be followed by a colon (:). **Figure 58** shows the use of a label to identify a program line and its use later in a JUMP instruction.

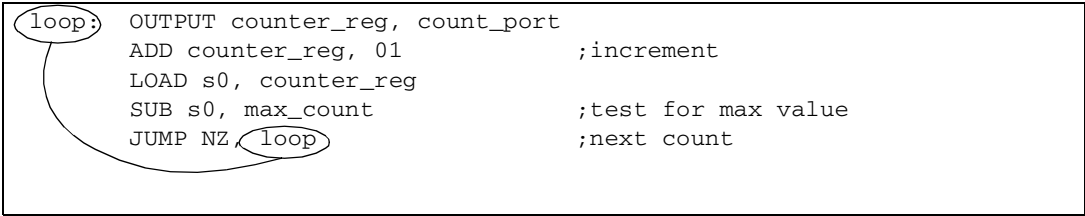


Figure 58: Line Label Example

Program
Instructions

The instructions should be as shown in **PicoBlaze Instruction Set**. The assembler is very forgiving over the use of spaces and <TAB> characters, but instructions and the first operand must be separated by at least one space. Instructions with two operands must ensure that a comma (,) separator is used.

The assembler accepts any mixture of upper and lower case characters for the instruction and automatically converts them to upper case. The following examples show acceptable instruction specifications, but the formatted output shows how it was expected.

load s5,7E	Assembler →	LOAD s5, 7E
AddCY s8,SE		ADDCY s8, sE
ENABLE interrupt		ENABLE INTERRUPT
Output S2, (S8)		OUTPUT s2, (s8)
jump Nz, 67		JUMP NZ, 67
ADD sF, step_value		ADD sF, step_value
INPUT S9,28		INPUT s9, 28
sl1 se		SL1 sE
RR S8		RR s8

Most other syntax problems are solved by reading the error messages provided by the assembler.

Assembler
Directives

The assembler supports three assembler directives. These commands are used purely by the assembly process and do not correspond to instructions executed by PicoBlaze module.

CONSTANT Directive

This directive provides a way to assign an 8-bit constant value to a label. In this way, the program can declare constants such as port addresses and particular data values needed in the program. By defining constant values in this way, it is often easier to understand their meaning in the program rather than as actual hexadecimal constant values in the program lines. **Figure 59** illustrates the directive syntax and its uses.

```

        CONSTANT max_count, 18           ;count to 24 hours
        NAMEREG s4, counter_reg         ;define register for counter
        CONSTANT count_port, 12
start:  LOAD counter_reg, 00             ;initialize counter
loop:   OUTPUT counter_reg, count_port
        ADD counter_reg, 01             ;increment
        LOAD s0, counter_reg
        SUB s0, max_count              ;test for max value
        JUMP NZ, loop                 ;next count
        JUMP start                    ;reset counter

```

Figure 59: **CONSTANT Directive**

Notes:

1. A constant is global. Even if a constant is defined as the end of the program file, it can be used in instructions anywhere in the program.
2. Constant names must not contain any spaces although the underscore character is supported. Valid characters are 0 to 9, a to z, and A to Z.

In Figure 59, “max_count” is being used to specify a data constant of 18 hex. In the program, this is used to test the value of a counter. By using a constant directive, the code is more readable. It would also be possible to change the constant value and its effect would be applied to multiple places in the program.

“count_port” is being used to specify a port address. In the program, the OUTPUT instruction refers to the port by name rather than absolute value. By using a constant directive, the code is more readable. It would also be possible to change the constant value once in the directive and its effect would be applied to multiple places in the program. This is particularly useful when defining the hardware interface. Indeed, the program can be developed before the I/O addresses are defined.

NAMEREG Directive

This directive provides a way to assign a new name to any of the 16 registers. In this way, the program refers to “variables” by name rather than as absolute register specifications. By naming registers in this way, it is easier to understand the meaning in the program without so many comments. It also helps to prevent inadvertent reuse of a register with associated data corruption. See Figure 60.

```

        CONSTANT max_count, 18           ;count to 24 hours
        NAMEREG s4, counter_reg         ;define register for counter
        CONSTANT count_port, 12
start:  LOAD counter_reg, 00             ;initialize counter
loop:   OUTPUT counter_reg, count_port
        ADD counter_reg, 01             ;increment
        LOAD s0, counter_reg
        SUB s0, max_count              ;test for max value
        JUMP NZ, loop                 ;next count
        JUMP start                    ;reset counter

```

Figure 60: **NAMEREG Directive**

Notes:

1. Register names must not contain any spaces although the underscore character is supported. Valid characters are 0 to 9, a to z, and A to Z.

In Figure 60, the register s4 has been renamed to be “counter_reg” and is then used in multiple instructions, making it clear what the meaning of the register contents actually are.

Important The NAMEREG directive is applied in-line with the code by the assembler. Before the NAMEREG directive, the register is named in the 'sX' style. Following the directive, only the new name applies.

It is also possible to rename a register again (i.e., NAMEREG counter_reg, hours) and only the new name applies in the subsequent program lines. This can be useful in making portable code and subroutines.

ADDRESS Directive

ADDRESS directive (Figure 61) provides a way to force the assembly of the following instructions commencing at a new address value. This is useful for separating subroutines into specific locations and vital for handling interrupts. The address must be specified as a two-digit hexadecimal value in the range 00 to FF.

```
JUMP NZ, inner_long
RETURN
;Interrupt Service Routine
ISR: LOAD wait_light, 01           ;register press of switch
OUTPUT wait_light, wait_light_port ;turn on light
RETURNI DISABLE                   ;continue light sequence but no more interrupts
ADDRESS FF                        ;Interrupt vector
JUMP ISR
;end of program
```

Figure 61: ADDRESS Directive

In Figure 62, the log file shows that the ADDRESS directive is used to force the last instruction into the highest memory location in the program RAM. This is the address to which the program counter is forced during an active interrupt.

```
E3 95E1      JUMP NZ, inner_long[E1]
E4 8080      RETURN
E5           ;Interrupt Service Routine
E5 0A01      ISR: LOAD wait_light[sA], 01           ;register press of switch
E6 EA10      OUTPUT wait_light[sA], wait_light_port[10] ;turn on light
E7 80D0      RETURNI DISABLE
FF           ADDRESS FF                        ;Interrupt vector
FF 81E5      JUMP ISR[E5]
FF           ;end of program
```

Figure 62: ADDRESS Directive Example

KCPSM2 Code Compatibility

KCPSM and KCPSM2 have many similarities. However, each has been tuned to specific device architecture so there are differences. Since KCPSM2 has more program space and more registers, it is more likely that a KCPSM design is migrated to a KCPSM2 rather than the other way round.

Registers

The key difference from a user perspective is that KCPSM has 16 registers with names "sX" rather than 32 registers with names "sXX." If KCPSM2 code is to be used with KCPSM code, the NAMEREG directive can make the code compatible.

Simply add the following lines before any active instructions:

- namereg s0,s00
- namereg s1,s01
- .
- .
- namereg sF,s0F

Now the 16 registers have the same identifiers as that for the lower 16 registers of KCPSM2. Of course, more than 16 registers cannot be supported.

However, the input code must use the “sXX” format correctly and not an upper case “S” or lower case hexadecimal digit. Therefore, the “<name>.fmt” output from KCPSM2 should be used where possible.

Address Range

Since the KCPSM module supports 256 program instructions and KCPSM2 module supports 1024, it is possible that the program simply will not fit. However, for programs less than 256 instructions, the different address range does make a difference to ADDRESS directives.

Any address directives used in the program code will need to be adjusted to specify the desired assembly address as two hexadecimal digits rather than three.

Interrupt Vector

During an active interrupt, the program counter of KCPSM2 is forced to the last memory location “3FF.” In a similar way, the program counter of KCPSM is also forced to the last memory location, but this is now “FF” due to the smaller program space. Therefore, it is vital that programs using interrupts adjust the location of the interrupt vector. This typically involves adjusting the associated ADDRESS directive from address 3FF to FF.

Label Validity

The assembler has slightly different rules concerning which labels for lines, constants, and registers are acceptable. For example, a constant label cannot be “se” because this can be confused with a default register name of the KCPSM macro. Therefore, it may be necessary to adjust some of the user names in the program code. Typically, labels are descriptive and this issue is not encountered.

Interrupt Handling

Effective interrupt handling and how and when an interrupt is used are not covered in this document. The information supplied should be adequate to assess the capability of the PicoBlaze module and to create interrupt-based systems.

Default State

By default, the interrupt input is disabled. This means that the entire 256 words of program space are used without any regard to interrupt handling or use of the interrupt instructions.

Enabling Interrupts

For an interrupt to take place, the ENABLE INTERRUPT command must be used. At critical stages of program execution where an interrupt is unacceptable, a DISABLE INTERRUPT is used. Since an active interrupt automatically disables the interrupt input, the interrupt service routine ends with a RETURNI instruction, which also includes the option to ENABLE or DISABLE the interrupt input as it returns to the main program.

During an interrupt (Figure 63), the program counter is pushed onto the stack and the values of the CARRY and ZERO flags are preserved (for restoration by the RETURNI instruction). The

interrupt input is automatically disabled. Finally, the program counter is forced to address FF (last program memory location) from which the next instruction is executed.

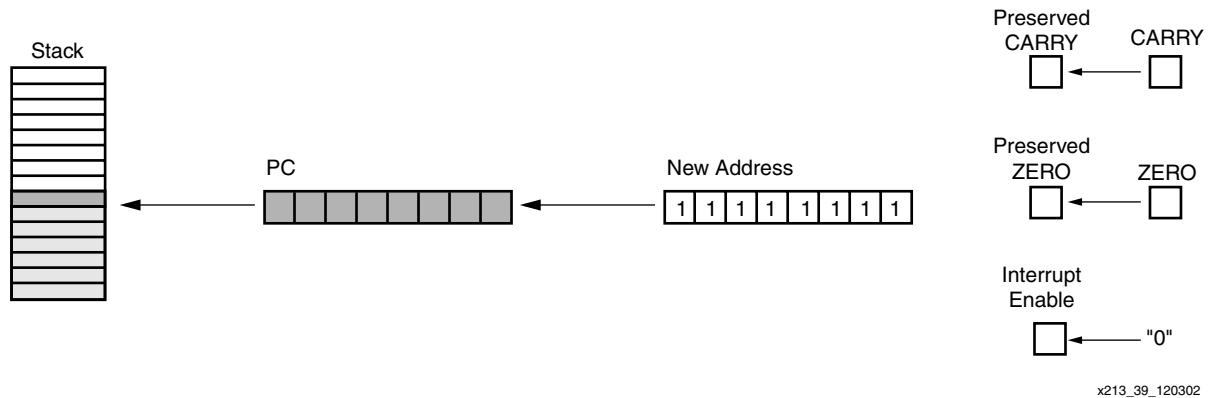


Figure 63: Effects of an Active Interrupt

Basics of interrupt Handling

Since the interrupt forces the program counter to address FF, it will generally be necessary to ensure that a jump vector to a suitable interrupt service routine is located at this address. Without a JUMP instruction, the program rolls over to address zero.

In typical cases, an interrupt service routine is provided. The routine can be located at any position in the program and jumped to by the interrupt vector located at the FF address. The service routine performs the required tasks and then ends in RETURNI with ENABLE or DISABLE.

Figure 64 illustrates a very simple interrupt handling routine. The PicoBlaze module generates waveforms to an output by writing the values 55 and AA to the waveform_port (port address 02). It does this at regular intervals by decrementing a register (s0) based counter seven times in a loop.

When an interrupt is asserted, the PicoBlaze module stops generating waveforms and simply increments a separate counter register (sA) and writes the counter value to the counter_port (port address 04).

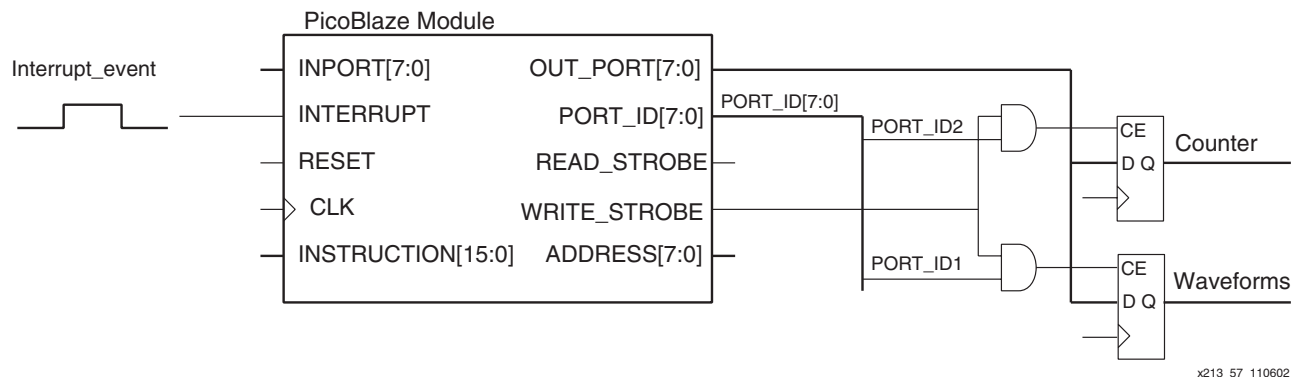


Figure 64: Interrupt Handling Example

Design VHDL Example

The following VHDL example (Figure 65) shows the addition of the data capture registers to the processor. Note the simplified port decoding logic through careful selection of port addresses. The complete VHDL file is supplied as `kcpsm_int_test.vhd`.

```
-- adding the output registers to the processor

IO_registers: process(clk)
begin

    -- waveform register at address 02

    if clk'event and clk='1' then
        if port_id(1)='1' and write_strobe='1' then
            waveforms <= out_port;
        end if;
    end if;

    -- Interrupt Counter register at address 04

    if clk'event and clk='1' then
        if port_id(2)='1' and write_strobe='1' then
            counter <= out_port;
        end if;
    end if;

end process IO_registers;
```

Figure 65: Design VHDL Example

Interrupt Service Routine

In the assembler log file for the example (Figure 66), it can be seen that the interrupt service routine has been forced to compile at address B0, and that the waveform generation is based in the normal lower addresses. This makes it easier to observe the interrupt in action in the operation waveforms. This program is supplied as `int_test.psm` for the user to compile.

```
00          ;Interrupt example
00          ;
00          CONSTANT waveform_port, 02                ;bit0 will be data
00          CONSTANT counter_port, 04
00          CONSTANT pattern_10101010, AA
00          NAMEREG sA, interrupt_counter
00          ;
00 0A00      start: LOAD interrupt_counter[sA], 00      ;reset int count
01 02AA      LOAD s2, pattern_10101010[AA]            ;start pattern
02 8030      ENABLE INTERRUPT
03          ;
03 E202      drive_wave: OUTPUT s2, waveform_port[02]
04 0007      LOAD s0, 07
05 6001      loop: SUB s0, 01
06 9505      JUMP NZ, loop[05]
07 32FF      XOR s2, FF
08 8103      JUMP drive_wave[03]
09          ;
09 B0        ; ADDRESS B0
B0 4A01      intRoutine : ADD interrupt_counter[sA], 01 ;increment counter
B1 EA04      OUTPUT interrupt_counter[sA], counter_port[04]
B2 80F0      RETURNI ENABLE
B3          ;
FF          ; ADDRESS FF
FF 81B0      JUMP intRoutine[B0] ;set interrupt vector
```

Main program delay loop where most time is spent

Interrupt service routine (here located at address B0 onwards)

Interrupt vector set at address FF and causing JUMP to service routine

x627_58_012703

Figure 66: Interrupt Example

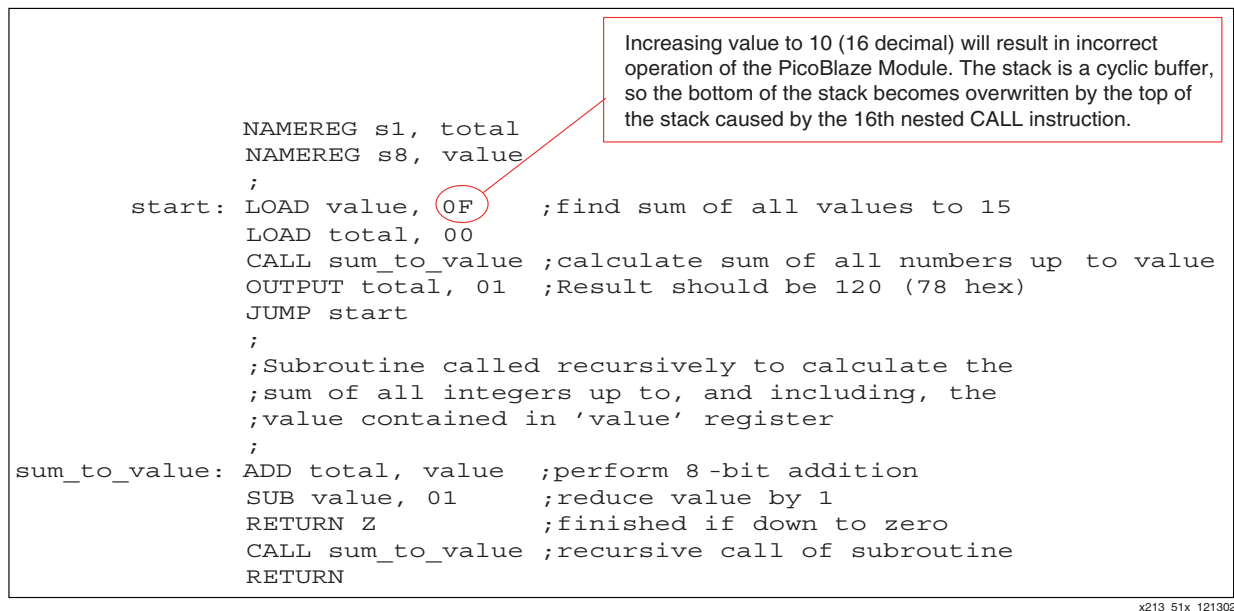
Therefore, it is advisable that an interrupt signal should be active for a minimum of two KCPSM rising clock cycle edges. An improvement would be for the interrupt service routine to acknowledge the interrupt to the external logic. There are three ways to achieve this:

- Service routine **writes** to a specific port to acknowledge interrupt and reset driving pulse (wasteful).
- **Read** a specific port to determine the reason for interrupt and use READ_STROBE as a register reset pulse.
- Decode the address bus to identify when the address FF has been forced by the active interrupt.

CALL/RETURN Stack

The PicoBlaze module contains an automatic embedded stack (Figure 69) which is used to store the program counter value during a CALL instruction (or interrupt) and restore the program counter value during a RETURN (or RETURNI) instruction. The stack does not need to be initialized or require any control by the user. However, the stack can only support nested subroutine calls to a depth of 15.

This simple program can calculate the sum of all integers up to a certain value, i.e. 'sum_of_value' when value = 5 is $1+2+3+4+5 = 15$. In this case, the sum of integers up to the value 15 (0F hex) is calculated to be 120 (78 hex). This is achieved by using a recursive call of a subroutine and results in the full depth of the call/return stack being utilized. Obviously, this is not an efficient implementation of this algorithm, but it does fully test the stack.



x213_51x_121302

Figure 69: CALL/RETURN Stack

Hints and Tips

Compare Operations

COMPARE Instruction

The PicoBlaze module does not directly support a compare instruction, so a combination of instructions based on a subtraction should be used. Here are three ideas:

Case 1 - A subtract instruction is destructive, so if the value in the register to be tested is valuable, then copy it to a temporary register before performing the test. In this example, the operation jumps to a routine if the value in sF is 27.

```
LOAD s0, sF
SUB s0, 27
JUMP Z, my_routine
```

Case 2 - The use of a temporary register may not be ideal, and the additional instructions to perform the compare can seem wasteful. Sometimes, the compare operation can be usefully combined with the operation being performed. In this example, a counter is being formed in the sF register and the next step is to jump to routine when the counter reaches the value 27.

```
Count_up: ADD sF, 01          ; Increment the counter
          SUB sF, 27          ; test if counter is 27
          JUMP Z, my_routine  ; Counter was value 27
          ADD sF, 27          ; Count value was not 27 so restore the value.
          JUMP count_up
```

Note that the act of testing the counter value in sF does destroy the value, and when the compare value is not 27, it needed to be restored using addition. However, when the count value was 27, the effect was also to reset the counter value which is probably what would have been required anyway.

Case 3 - It is possible to test if a register is zero without destroying the contents and only using a single instruction. In this example, sF reads the status of an input port and a test is made to see if any switches have been pressed (indicated by a 1).

```
INPUT sF, switch_port
AND sF, sF          ; test for zero
JUMP NZ, switch_routine
```

Increasing Program Space

The PicoBlaze module is designed to work with a single block RAM which provides 256 locations in the Spartan-II and Virtex-E devices. If the user requires more program memory, then here are some potential solutions to be considered:

Solution A - Consider the Virtex-II architecture. The block RAM of these devices provides 1024 locations. There is a KCPSM2 module specifically designed for this architecture (see [XAPP627](#)). The KCPSM2 module also provides 32 registers which is useful for the larger program.

Solution B - PSM stands for Programmable State Machine. If the user's program is bigger than 256 instructions, then it may simply be that the user is trying to do too much with a single PicoBlaze module. The user should consider dividing the whole process into several small processes, each under the control of a separate PicoBlaze module (and each with its own block RAM)([Figure 70](#)). At 76 slices per processor, additional processors do not necessarily take up that much space. Several users have found this an excellent solution because it simplifies the software and the hardware interface logic making operations work independently (even when interrupts occur), which is easier to test. It also simplifies the data input and output to the processor (less peripheral logic). Currently, the record is eight PicoBlaze modules in one device.

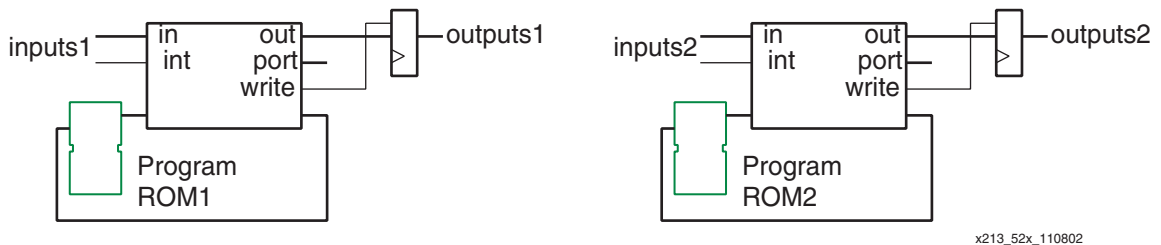


Figure 70: Increasing Program Space

In some cases, one PicoBlaze module can be the master device, with one or more PicoBlaze modules acting as coprocessors. Dual-port distributed memory is a good way to exchange data between processors.

Solution C - Divide the program between two or more block RAMs and use a *soft switch* to page between memories containing different routines (Figure 71). A block switching routine is located at the same memory location in each memory block (using the ADDRESS directive). The processor at some stage executes this routine (possibly because of an interrupt). The routine writes to an output port to execute the switch of block RAMs. Since the new block RAM contents are the same in this location of RAM, the processor does not see anything change until it performs a JUMP back into the main program area. Obviously, this technique requires that the program can be suitably divided.

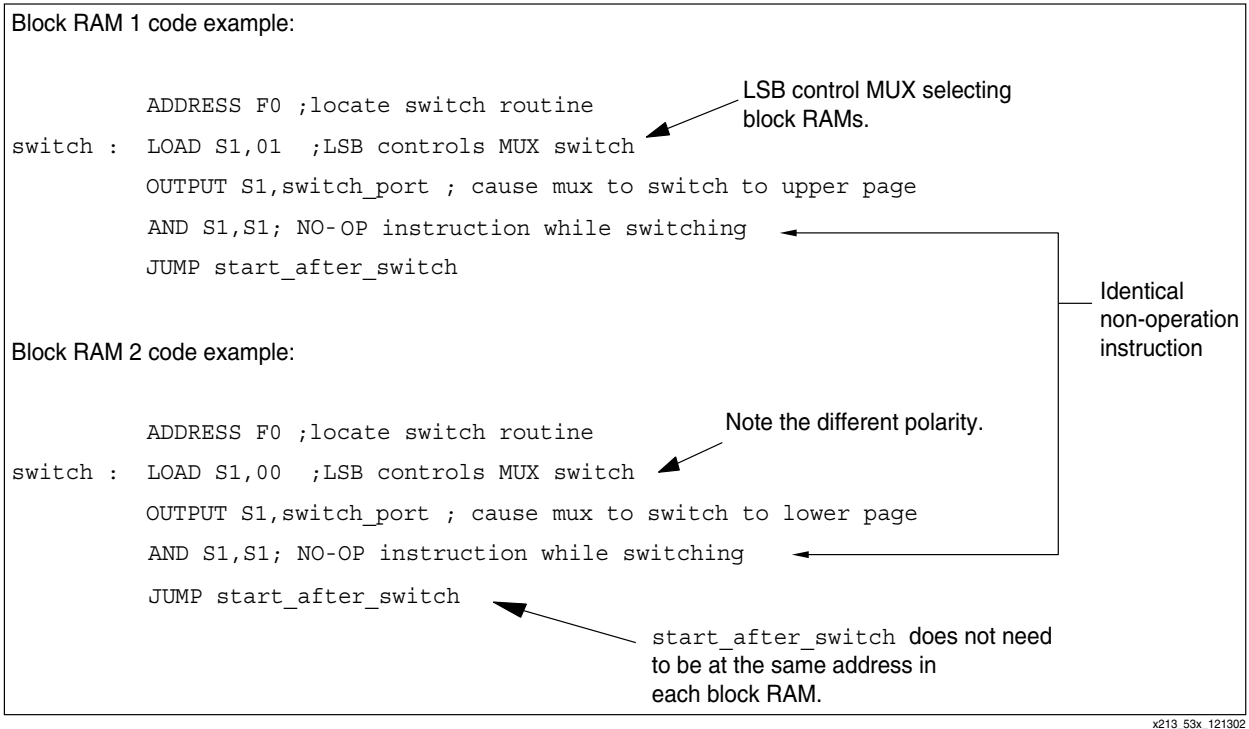


Figure 71: Block RAM Code Examples

The most obvious way to switch between block RAMs is to use a multiplexer with the select line controlled by a single flip-flop register mapped as an output port to the PicoBlaze module (Figure 72). This means that the assembler can be used to generate the two program ROM files as normal VHDL files and insert them into the design.

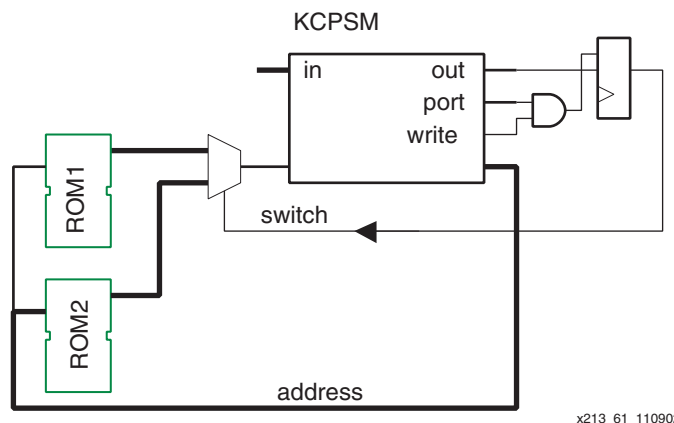


Figure 72: Switching Between Block RAMs (Method 1)

The disadvantages of this method is that the multiplexer requires eight slices to implement, and the effective increase in the clock-to-output delay of the program memory results in a lower maximum clock rate.

A more elegant solution (Figure 73) is to absorb the multiplexer back into the block RAMs by configuring each in 512×8 aspect ratio, and then using the select bit to address the MSB (9th) address bit. However, the instructions are then split across the RAM blocks, making programming the ROMs more challenging. See Figure 74 for details of a utility to automatically generate this structure

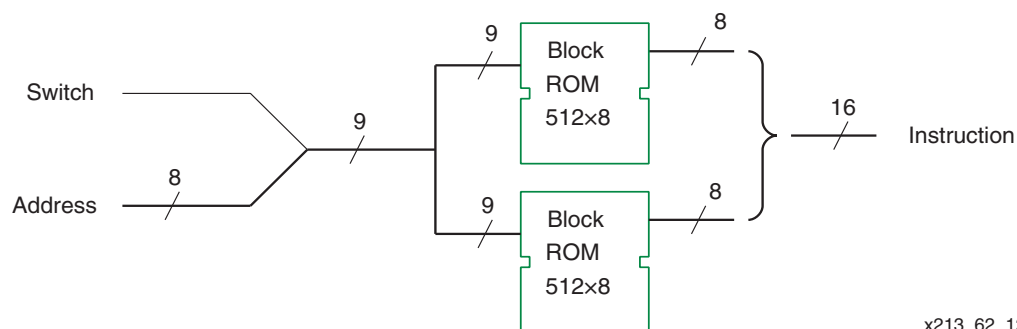


Figure 73: Switching Between Block RAMs (Method 2)

PSMSPLIT

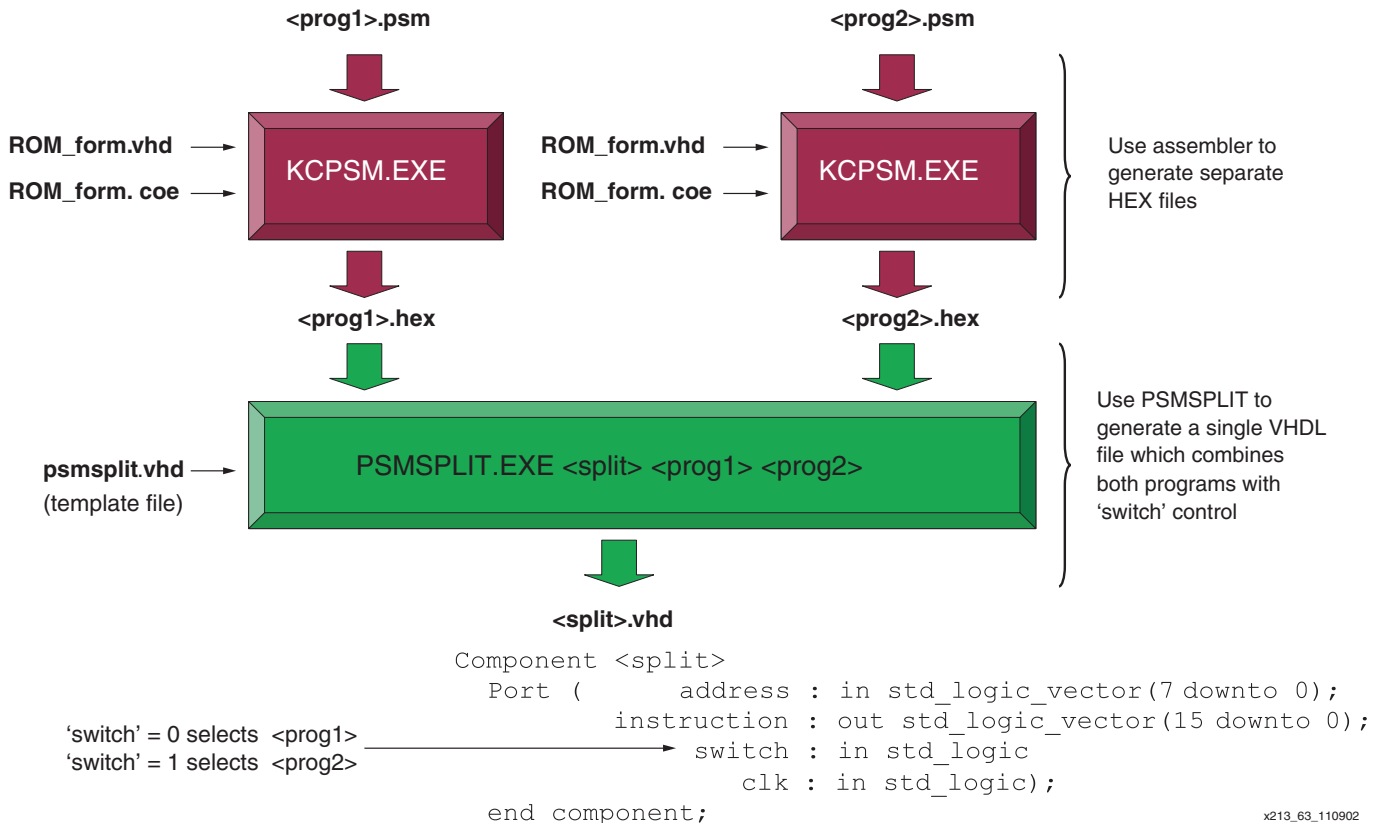


Figure 74: PSMSPLIT

PSMSPLIT Example Design

An example design (Figure 75) using PSMSPLIT is supplied and can be compiled as follows.

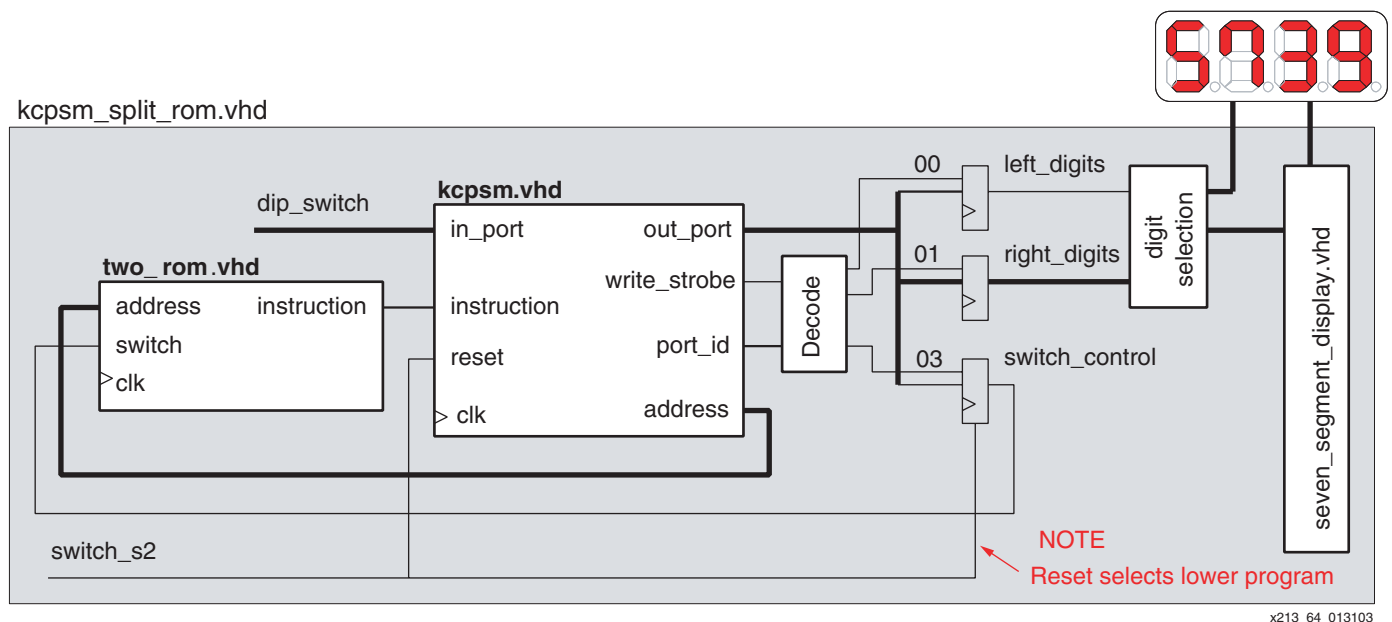


Figure 75: PSMSPLIT Example Design

The example design is based on a small demo board. The concept is to control the four 7-segment displays so that the left two digits form a decimal 0 to 99 counter and the right two digits form a second decimal 0 to 99 counter. Each counter increments at approximately 1-second intervals based on a software delay loop. However, only one counter is incrementing, depending on the setting of the least significant DIP switch input. In this simple case, there are two programs with each responsible for a different counter. The DIP switch setting is actually used to swap programs being executed.

- **lo_prog.psm** - This is the start program which clears counters and then controls the left digits only. Run **kcpsm lo_prog**.
- **hi_prog.psm** - This program controls the right digits only. Run **kcpsm hi_prog**.
- To form the VHDL file used in the design, run **psmsplit two_rom lo_prog hi_prog**.

Reference Design Files

All files described in this application note (plus some additional files) are available on the Xilinx Xilinx PicoBlaze Lounge site at:

http://www.xilinx.com/ipcenter/processor_central/picoblaze/index.htm

Conclusion

A microprocessor module does not have to be large or expensive when implemented in a Virtex or Spartan-II device. The Virtex architectural features (block memory, distributed memory, dedicated multiplexers, and carry logic) are ideal for the construction of fully embedded microprocessor modules.

The PicoBlaze module is a simple 8-bit processor with an instruction set for basic control functions and data manipulation. This is achieved with just 76 slices and one block RAM. Even with a silicon utilization over performance objective, over 40 MIPs of processing power shows the very high performance provided by Xilinx devices. Most typical applications do not exploit this performance, but simply benefit from the small size and the design methodology.

When a processor is completely embedded within an FPGA, no I/O resources are required to communicate with other modules in the same FPGA. Additionally, system design flexibility is included along with savings on PCB requirements, power consumption, and EMI. Whenever a special type of instruction is required, it can be created in hardware (other CLBs) and connected to the PicoBlaze solution as a kind of coprocessor. Indeed, there is nothing to prevent a coprocessor from being another PicoBlaze module. In this way, even the 256-instruction program length is not a limitation.

PicoBlaze has been used successfully by thousands of Xilinx customers. Many references to its use and alternative software development tools can be found when searching the web (search for KCPSM and PicoBlaze). The author welcomes any feedback from PicoBlaze users.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/25/00	1.0	Initial Xilinx release.
10/04/00	1.1	Minor text edits to make the copy more readable.
04/30/02	1.2	Replaced KCPSM with PicoBlaze globally.
12/17/02	2.0	Major revision for compatibility with ISE tools.
02/04/03	2.1	Minor edits done.