

8-Bit Pipelined Picoblaze Validation Plan

Jim Rowe, Victoria Van Gaasbeck, Danny Hale

ECE571-FALL 2021

12/07/2021

Introduction:	1
Design Description:	2
Coverage:	3
Assertions:	3
Unit Level Testing Plans	3
Reset	3
Instruction Fetch	4
Instruction Decode:	5
Instruction Execute:	7
Memory Access:	8
Writeback:	8
Results	10
Lessons Learned	12
Team Member Contributions	12

Introduction:

As the presence of increasingly complex and affordable microcontrollers and ASICs become ubiquitous in the modern world it is easy to get carried away in the notion of designing a new digital system. While system design is an important and fascinating part of engineering , these complex designs necessitate increased effort dedicated to designing verification and validation environments. Proper verification ensures the development of correct and maintainable designs and helps to prevent costly and time consuming errors surfacing post-silicon. Given the importance of verification and validation it is important that engineering students gain experience designing validation plans.

Design Description:

The overall goal of this project is to validate an 8-bit pipelined picoblaze. We intend to start our testing at the unit level, ensuring each unit functions properly on its own. Once we feel comfortable with our unit level testing we will increase the testing scope to encompass unit to unit interaction and the top level design.

First is to identify coverage points and stimulation. This includes important features of the design that must be tested for in order to finish testing by knowing all possible behaviors have been observed. Once cases have been covered in every phase we can move forward in creating assertions and testing that behavior is designed as intended in the design spec.

Each design unit's functionality will be tested utilizing black boxes and cones of influence. It is important to test the design units first before testing the interactions between units, so we can localize bugs at the unit level. The final goal for unit testing is to check for the completeness of features implemented in that unit.

Before the final goal the intended behavior of each phase must be verified to always be true through assertions. This includes opcodes are properly executed, interactions are fetched in sequential order and jumps to the proper location.

After independent unit verification, verify unit to unit interactions. Unit to unit interactions contain possible timing issues and require certain behaviors on these ports such as ALU writing back to registers or IR sending opcode, sources, and destinations correctly to ALU.

Final verification test is targeted at the top level to verify design wide functionality. Top level verification will check for things such as the correct execution of instruction types and verifying outputs to the register file and memory.

With the final stage there is also testing corner cases and logic outside the units and stages in the pipeline, this includes hazards, interrupts and interactions between the stages in the pipeline.

Coverage:

As described above, coverage should check functionality of each feature in the design. This includes covering ALU opcodes, register destinations and sources, memory destinations sources, pc increment and jump, FSM states, instruction register storage. We will also account for single iteration cases such as register write backs, memory data reads etc. Overall, coverage checks that basic functionality of the design is correct to the ISA.

Assertions:

After coverage conditions intended to be observed to check for initial completeness of the verification, assertions must prove what the design will and should do. Assertions will start with simple cases such as checking execution of the opcodes for the ALU and that the opcode is properly decoded from the instruction and finish with corner cases that test undefined functionalities such as unsupported opcodes and out of bounds FSM states. Final assertions check that each type of instruction and particular sequences of instructions can properly be handled at the top level.

Unit Level Testing Plans

Reset

Mostly straight forward assertion testing for validating that initial values are set.

Signal Name	Description	Good/Bad
program_counter	Asserts is reset to zero	Good
stack_pointer	Asserts is reset to top of memory range	Good
index_operation	Asserts is reset to LOAD	Good

index_reg_x_out	Asserts is reset to zero	Good
index_reg_y_out	Asserts is reset to zero	Good
index_dst	Asserts is reset to zero	Good
zero	Asserts is reset to zero	Good
carry	Asserts is reset to zero	Bad
interrupt_ack	Asserts is reset to zero	Good
interrupt_enable	Asserts is reset to zero	Good
interrupt_latch	Asserts is reset to zero	Good
write_strobe	Asserts is reset to zero	Good
read_strobe	Asserts is reset to zero	Good
register_write_enable	Asserts is reset to zero	Good
exwb_register_write	Asserts is reset to zero	Good
scratch_write_enable	Asserts is reset to zero	Good
stack_write_enable	Asserts is reset to zero	Good
zero_carry_write_enable	Asserts is reset to zero	Good

Instruction Fetch

Instruction fetch testing will be fairly concise, and consist of testing reset conditions and proper updating of the program counter under various circumstances. Jump, call and return operations require that the program counter update to a different address supplied by the idu/stack and it's important to verify the proper instruction is fetched.

The final implemented strategy involved utilizing a combination of programs with various branching instructions to provide stimuli and assertions to verify proper instruction fetching.

Program	Description
test_call.psm	Tests CALL, RETURN from subroutine

test.psm	Tests JUMPS based on different flags
jump_sub_test.psm	More JUMP tests
fetch_store_test.psm	Tests another subroutine

Assertion/Property	Description	PASS/FAIL
Property ifid	Checks that ifid_pcplus2 = program_counter + 2	PASS
Property pc_jump	Checks proper PC behavior on JUMP/CALL	PASS
Property pc_return	Checks proper PC behavior on RETURNS	PASS

Instruction Decode:

The decode unit takes the instruction fetched from the instruction ROM and decodes the information into its various parts for use in control signals and the subsequent pipeline stages.

The high level strategy for testing this unit will be to supply every sort of instr_t and check for the properly decoded output. Decoded outputs to verify for correctness include signals such as idu_operation, the x/y addresses, conditional flags, operand selection, etc. If, for instance, a conditional flag was improperly decoded the program counter might not be updated properly (inferring a JUMP or missing a JUMP operation) and the incorrect instruction could be fetched next.

We intended to utilize SystemVerilog's OOP functionality and constrained randomization to generate opcodes (including invalid options) to test the instruction decode unit's functionality. There will likely be overlap for code reusability between instruction decode and instruction execute tests.

Actual implementation differed a bit from the initial strategy. Constrained randomization and OOP were not implemented at this time, but remain good goals for future projects. The implemented strategy utilized a series of assembly programs to provide stimulus to the model. Programs were written to cover nearly all instruction types and opcodes. The programs were first tested on the working Picoblaze model to ensure functional correctness of the code and monitor what execution looks like in a

working model. Assertions were used on the outputs of the instruction decode unit to check for bugs.

The table below shows a sample of programs used to provide stimuli. Note, LOAD instruction is used in all of these programs and was not listed in each description.

Program	Description
logical_ops.psm	Tests logical operations : AND, OR, XOR
test.psm	Tests TEST
addcy_sub.psm	Tests ADDCY, including a subroutine
shift_test.psm	Tests Left and Right shifting operations with branching
jump_sub_test.psm	Tests SUB with branching and ADD with immediate
fetch_store_test.psm	Tests FETCH and STORE
reg_reg_justadd.psm	Test register to register ADD
shift.psm	Tests left shift, no branching

The following table lists the assertions used for the instruction decode stage of the pipeline. Each output of the IDU was compared against the instruction that was used as input to the unit.

Property/Assertion	Description	PASS/FAIL
idu_op	Checks idu operation	PASS
idu_implied	Checks idu implied value	FAIL
idu_flag	Checks idu conditional flags	PASS
idu_xaddr	Checks idu x address	PASS
idu_yaddr	Checks idu y address	PASS
idu_operand	Checks idu operand selection	PASS
idu_shiftdir	Checks idu shift direction	PASS
idu_shiftc	Checks idu shift constant	FAIL
idu_shiftop	Checks idu shift operation	PASS
idu_port	Checks idu port	PASS
idu_scratch	Checks idu scratch address	PASS

idu_codeaddr	Checks idu code address	PASS
idu_cond	Checks idu conditional selection	PASS

Instruction Execute:

The execute stage is responsible for enabling/disabling interrupts, modifying cpu flags, and setting the stack address. Tests with asserts will be conducted to verify that these control signals get set only when the situation is appropriate.

The ALU also falls within the domain of the execute stage and will be tested for coverage of all valid opcodes. Invalid opcodes will also be tested to ensure that the failure doesn't propagate and that the ALU is always operating within a defined manner.

Assertion SV File	Program File(s)	Assertions for...	Complete
AluAssert.sv	All	Test each rojoblaze enumeration in opcode_instr_t	Yes
ExAssert.sv	All	Assert that the EX stage control signals FSM get set in all 18 picoblaze cases	Yes except INPUT/OUT PUT
		Use randomization to test each picoblaze enumeration in opcode_t	No
AluAssert.sv	shift_test.h ex	Check all 10 shift and rotate cases (enum RS)	Yes
AluAssert.sv	test.hex	Assert that the carry flag gets set using both arithmetic and test instructions	Yes
AluAssert.sv	test.hex	Assert that the zero flag gets set using both arithmetic and test instructions	Yes
ExAssert.sv	test_call.he x	Check CALL opcode using carry and zero flags	Yes
ExAssert.sv	test_call.he x	Check RETURN opcode using carry and zero flags	Yes
		Check that RETURNI properly sets the both the return address and the interrupt flag on the same cycle	No
		Check control signals for enabling/disabling interrupts	Planned
		Check EX outputs when using an invalid opcode(s)	Planned
AluAssert.sv ExAssert.sv	EX	Check EX stage FSM for latches	Yes
		INPUT instruction only drives the read strobe	Needed an

			IO device
		OUTPUT only drives the write strobe	Needed an IO device
AluAssert.sv	logical.hex, add.hex, subtract.he x	Check that neither FETCH nor INPUT bits are set during ALU operations	
AluAssert.sv	EX	Check FETCH and INPUT bits in idex_reg_source are onehot	Planned Next

Memory Access:

Testing will validate that memory addresses are available when the writeback stage expects them. Instructions that do not require memory accesses are verified to ensure that registers are properly forwarded.

Assertion File	Programs	Test	Complete
		Check boundaries of the scratchpad (64, 8-bit entries by default)	No
PipelineAssert.sv	Any	Check that non-memory bound operations are simply forwarded	Yes

Writeback:

Testing that data created through the EXECUTE and MEM phases are properly written back to the register file. Cover all three possible sources propagated to output, input instruction, scratch pad and ALU result, as well as all register file destinations.

Covers:

Team Member	Design Unit	Test
Jim	WB	Check for Scratch Pad source
Jim	WB	Check for Input instruction source
Jim	WB	Check for ALU result source
Jim	WB	Check all registers written to

Asserations:

Team Member	Design Unit	Test
Jim	WB	WB destination and data always register file input
Jim	WB	If ALU source then must have had EX_enable asserted cycle prior
Jim	WB	Exwb_reg_source one hot
Jim	WB	If not enabled then no data is allowed to the ScratchPad

Hazards:

Due to the simplicity of the micro architecture the amount of hazards is reduced to no structural hazards, one data hazard and several control hazards to verify. No structural hazards exist as fanouts are utilized in multi output modules as well as a separated instruction and data memory acting as the ROM and Scratch Pad. the only data hazard exists in a WAR, stalling the prior instruction until the dependency is written back to the register file. Finally, many control hazards are located in the fetch as due to the multiple conditional instructions that cannot fetch the instruction until the condition is evaluated in the execute phase.

Covers:

Team Member	Design Unit	Test
Jim	HAZ	Check all conditionals caused an interrupt for FETCH
Jim	HAZ	Check RAW hazard is identified and sent interrupt
Jim	HAZ	Check RAW can occur on both operands
Jim	HAZ	Check RAW can occur on single source instructions

Asserations:

Team Member	Design Unit	Test
Jim	HAZ	On interrupt the PC is set to 3FF.
Jim	HAZ	When there is a RAW hazard data from WB is forwarded ALU.
Jim	HAZ	When there is a RAW hazard the correct register is used.

Results

Through our testing we were able to pick out four bugs in the broken model.

1. Carry bit set high on reset
2. idu_implied set incorrect
3. idu_shift_constant set incorrect
4. AND performed the incorrect operation

Assertions tested the reset vector and found that the carry bit was incorrectly initialized to ‘1’ on reset as opposed to ‘0’.

```
#      Time: 35 ns Started: 25 ns  Scope: alt_rojo_tb.dut.onReset File: N:/  
# ** Error:                      45 Simulation Failed Reset  
#   reset: 1  
#   program_counter: 000 should be 000  
#   stack_pointer: 1f should be 0x1f  
#   index_operation: LOAD should be LOAD  
#   index_reg_x_out: 00 should be 00  
#   index_reg_y_out: 00 should be 00  
#   index_dst: 0 should be 0  
#   zero: 0 should be 0  
#   carry: 1 should be 0  
#   interrupt_ack: 0 should be 0  
#   interrupt_enable: 0 should be 0  
#   interrupt_latch: 0 should be 0  
#   write_strobe: 0 should be 0  
#   read_strobe: 0 should be 0  
#   register_write_enable: 0 should be 0  
#   exwb_register_write: 0 should be 0  
#   scratch_write_enable: 0 should be 0  
#   stack_write_enable: 0 should be 0  
#   zero_carry_write_enable: 0 should be 0
```

Figure 1: Transcript showing failed reset

The instruction decode unit was also a source of bugs. The outputs to the IDU were checked with assertions comparing the decoded output to the instruction input. Our assertions flagged two errors, the idu_implied_value was wrong and the idu_shift_constant was improperly set. The assertions labeled “idu_implied” and “idu_shiftc” flagged these errors. The nature of these bugs caused issues on nearly every instruction and the transcripts were so long that the QuestaSim transcript eventually cut off the upper part.

```
Time: 4815 ns Started: 4815 ns  Scope: alt_rojo_tb.dut File: N:/ECE571/ece571f21/vvan/FinalProject/picoblaze_project_bugs/kcpsm_rojo.svp Line: 251  
4815      LOAD sA,sA z = 0 c = 0 || shiftbit = x || shiftop = RL_SRX || reg cont = 0a  
* Error: 4825$time, idu_shift_constant wrong  
Time: 4825 ns Started: 4825 ns  Scope: alt_rojo_tb.dut File: N:/ECE571/ece571f21/vvan/FinalProject/picoblaze_project_bugs/kcpsm_rojo.svp Line: 277  
* Error: 4825$*idu_implied_value: 01, instruction.instr_type.reg const.constant: 00  
Time: 4825 ns Started: 4825 ns  Scope: alt_rojo_tb.dut File: N:/ECE571/ece571f21/vvan/FinalProject/picoblaze_project_bugs/kcpsm_rojo.svp Line: 251  
4825      LOAD s1,00 z = 0 c = 0 || shiftbit = 0 || shiftop = SA || reg cont = 0c  
* Error: 4835$time, idu_shift_constant wrong  
Time: 4835 ns Started: 4835 ns  Scope: alt_rojo_tb.dut File: N:/ECE571/ece571f21/vvan/FinalProject/picoblaze_project_bugs/kcpsm_rojo.svp Line: 277  
* Error: 4835$*idu_implied_value: 03, instruction.instr_type.reg const.constant: 01  
Time: 4835 ns Started: 4835 ns  Scope: alt_rojo_tb.dut File: N:/ECE571/ece571f21/vvan/FinalProject/picoblaze_project_bugs/kcpsm_rojo.svp Line: 251  
4835      SUB s0,01 z = 0 c = 0 || shiftbit = 0 || shiftop = SA || reg cont = 00  
* Error: 4845$*idu_implied_value: 03, instruction.instr_type.reg const.constant: 01
```

Figure 2: Transcript showing repeated IDU error messages

ALU operations mostly passed except for AND. By using force to make idu_implied correct, it was discovered that AND was performing a negation on operand_b. Only one sample is shown on the images below but the negation of B was repeated over and over in the broken model.

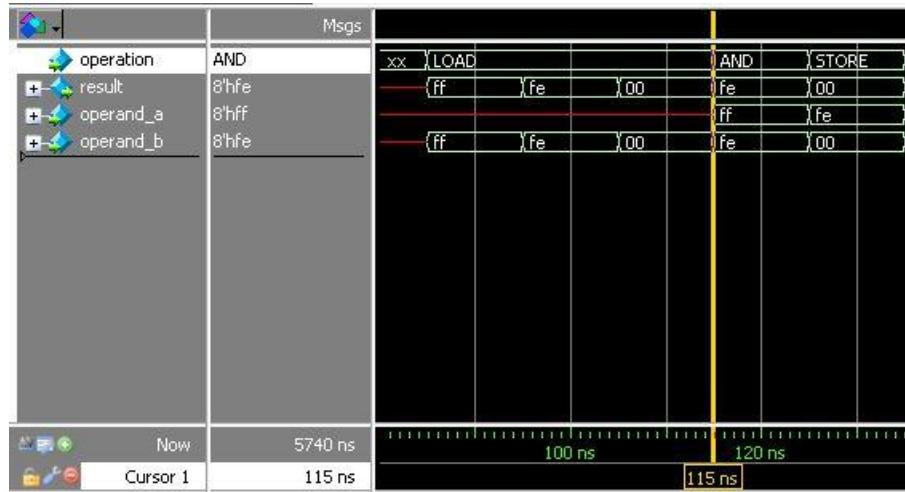


Figure 3: Unbroken model showing the result of and'ing 0xff and 0xfe

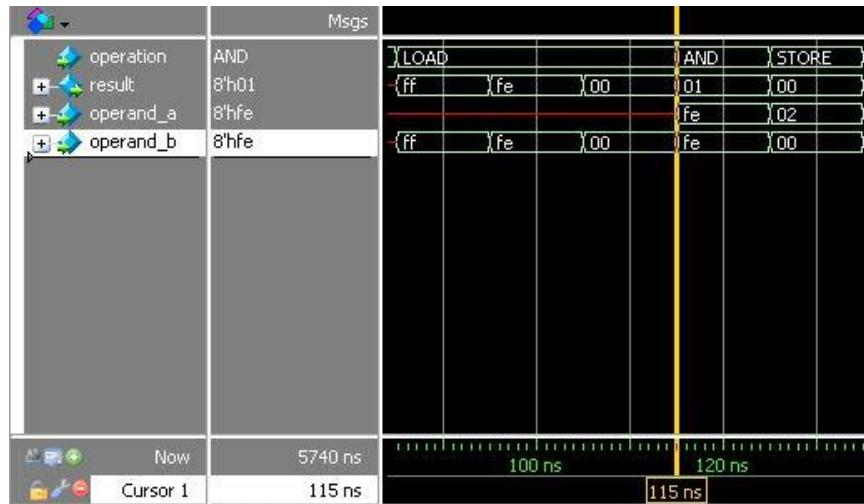


Figure 4: Broken model showing the result holding a negation of operand_b

In all we had 43 assertions and our testing hit 90.70% of those assertions.

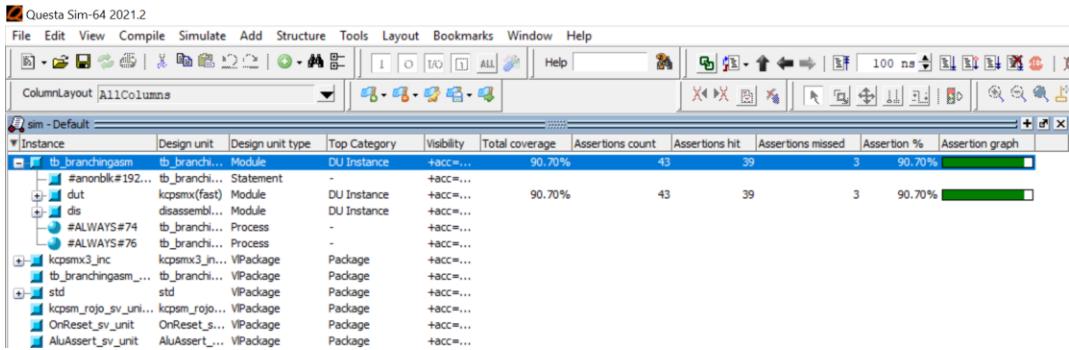


Figure 5: Questasim Assertion Coverage

Lessons Learned

We are pleased with the bugs we were able to find in the design, but there were many lessons learned and things we'd do differently in the future. First off, project organization strategies could be improved upon. Two of the three team members are relatively “green” at GitHub so that tool was not used to its full potential. Furthermore, a more consistent coding style guide was something we discussed at the beginning of the project but failed to successfully implement as the term got busier.

A big lesson learned in regards to verification strategy is, keep it simple with the test stimulus, at least at first. In a misguided attempt to be “efficient” most of the initial assembly programs were overly complicated. Too many different instructions were used together in any single program which was not particularly helpful with debugging. In the future we would opt for a strategy of more programs that handle simple things as opposed to fewer programs that test many things at once. Once the simple tests are worked out, then there's room for adding complexity.

Our other biggest hurdle was underestimating the time requirements and complexity of learning new topics. Certain assertions proved harder to code than initially presumed. We also failed to implement constrained randomization like we intended. Going forward it's something we would like to implement now that we've got a better understanding of the tools and processes.

Team Member Contributions

Victoria: Wrote the assembly programs used to provide stimulus to the model. Modified the provided test bench to work for our project. Responsible for unit testing Instruction Fetch and Instruction Decode units.

Danny: Figured out how to make assertions work without heavy modification to the original files. Figured out how to organize a single project to use two sets of files describing the same module without conflict. Wrote assertions for the ALU, reset, execute stage control signals.

Jim: Organized our initial verification plan. Took on the difficult assertions for writeback and pipeline hazards.

Jim Rowe, Victoria Van Gaasbeck, Danny Hale

ECE571-FALL 2021

11/25/2021

8-Bit Pipelined Picoblaze Validation Plan

Introduction:

As the presence of increasingly complex and affordable microcontrollers and ASICs become ubiquitous in the modern world it is easy to get carried away in the notion of design. While design is an important and fascinating part of engineering , these complex designs necessitate increased effort dedicated to verification and validation. Proper verification ensures the development of correct and maintainable designs and helps to prevent costly and time consuming errors surfacing post-silicon. Given the importance of verification and validation it is important that engineering students gain experience designing validation plans.

Design Description:

The overall goal of this project is to validate an 8-bit pipelined picoblaze. We intend to start our testing at the unit level, ensuring each unit functions properly on its own. Once we feel comfortable with our unit level testing we will increase the testing scope to encompass unit to unit interaction and the top level design.

First is to identify coverage points and stimulation. This includes important features of the design that must be tested for in order to finish testing by knowing all possible behaviors have been observed. Once cases have been covered in every phase we can move forward in creating assertions and testing that behavior is designed as intended in the design spec.

Each design unit's functionality will be tested utilizing black boxes and cones of influence. It is important to test the design units first before testing the interactions between units, so we can localize bugs at the unit level. The final goal for unit testing is to check for the completeness of features implemented in that unit.

Before the final goal the intended behavior of each phase must be verified to always be true through assertions. This includes opcodes are properly executed, interactions are fetched in sequential order and jumps to the proper location.

After independent unit verification, verify unit to unit interactions. Unit to unit interactions contain possible timing issues and require certain behaviors on these ports such as ALU writing back to registers or IR sending opcode, sources, and destinations correctly to ALU.

Final verification test is targeted at the top level to verify design wide functionality. Top level verification will check for things such as the correct execution of instruction types and verifying outputs to the register file and memory.

With the final stage there is also testing corner cases and logic outside the units and stages in the pipeline, this includes hazards, interrupts and interactions between the stages in the pipeline.

Coverage:

As described above, coverage should check functionality of each feature in the design. This includes covering ALU opcodes, register destinations and sources, memory destinations sources, pc increment and jump, FSM states, instruction register storage. We will also account for single iteration cases such as register write backs, memory data reads etc. Overall, coverage checks that basic functionality of the design is correct to the ISA.

Assertions:

After coverage conditions intended to be observed to check for initial completeness of the verification, assertions must prove what the design will and should do. Assertions will start with simple cases such as checking execution of the opcodes for the ALU and that the opcode is properly decoded from the instruction and finish with corner cases that test undefined functionalities such as unsupported opcodes and out of bounds FSM states. Final assertions check that each type of instruction and particular sequences of instructions can properly be handled at the top level.

Unit Level Testing Plans

Instruction Fetch:

Victoria will be the point person for testing the instruction fetch unit. Instruction fetch testing will be fairly concise, and consist of testing reset conditions and proper updating of the program counter under various circumstances. Jump, call and return operations require that the program counter update to a different address supplied by the idu/stack and it's important to verify the proper instruction is fetched.

Team Member	Design Unit	Test
Victoria	IF	Test PC = reset_vector when internal reset signal asserted
Victoria	IF	Check correct instruction fetched (PC = idu_code_address) on JUMP/CALL operation
Victoria	IF	Check correct instruction fetched (PC = stack_data_out) on RETURN/RETURNI operation

Instruction Decode:

Victoria will also be the point person for testing the instruction decode unit. The decode unit takes the instruction fetched from the instruction ROM and decodes the information into its various parts for use in control signals and the subsequent pipeline stages.

The high level strategy for testing this unit will be to supply every sort of instr_t and check for the properly decoded output. Decoded outputs to verify for correctness include signals such as idu_operation, the x/y addresses, conditional flags, operand selection, etc. If, for instance, a conditional flag was improperly decoded the program counter might not be updated properly (inferring a JUMP or missing a JUMP operation) and the incorrect instruction could be fetched next.

We intend to utilize SystemVerilog's OOP functionality and constrained randomization to generate opcodes (including invalid options) to test the instruction decode unit's functionality. There will likely be overlap for code reusability between instruction decode and instruction execute tests.

Team Member	Design Unit	Test
Victoria	ID	Test idex_reg's/idex_operation reset properly when internal_reset asserted
Victoria	ID	Test correct opcode decoded using constrained randomization of instr_t

Victoria	ID	Test outputs from IDU when provided invalid instructions
Victoria	ID	Test for proper condition_flag assertion on conditional instructions

Initial plans are discussed here, but Victoria expects that her plans may grow and change as she progresses through the project and gets more familiar with the process.

Instruction Execute:

The execute stage is responsible for enabling/disabling interrupts, modifying cpu flags, and setting the stack address. Tests with asserts will be conducted to verify that these control signals get set only when the situation is appropriate.

The ALU also falls within the domain of the execute stage and will be tested for coverage of all valid opcodes. Invalid opcodes will also be tested to ensure that the failure doesn't propagate and that the ALU is always operating within a defined manner.

Team Member	Design Unit	Test
Danny	EX	Check that the ALU FSM can go through all 37 rojoblaze opcodes
Danny	EX	Use randomization to test each rojoblaze enumeration in opcode_instr_t
Danny	EX	Assert that the EX stage control signals FSM get set in all 18 picoblaze cases
Danny	EX	Use randomization to test each picoblaze enumeration in opcode_t
Danny	EX	Check EX output signals when using a valid opcode
Danny	EX	Check all 10 shift and rotate cases (enum RS)
Danny	EX	Assert that the carry flag gets set using both arithmetic and test instructions
Danny	EX	Assert that the zero flag gets set using both arithmetic and test instructions
Danny	EX	Check that carry and zero can both be set under the right condition i.e. 128+128
Danny	EX	Check CALL opcode using carry and zero flags
Danny	EX	Check RETURN opcode using carry and zero flags
Danny	EX	Check that RETURNI properly sets the both the return address and the interrupt flag on the same cycle
Danny	EX	Check control signals for enabling/disabling interrupts
Danny	EX	Check EX outputs when using an invalid opcode(s)
Danny	EX	Check EX stage FSM for latches
Danny	EX	INPUT instruction only drives the read strobe
Danny	EX	OUTPUT only drives the write strobe

Danny	EX	Check that neither FETCH nor INPUT bits are set during ALU operations
Danny	EX	Check FETCH and INPUT bits in idx_reg_source are onehot

Memory Access:

Testing will validate that memory addresses are available when the writeback stage expects them. Instructions that do not require memory accesses are verified to ensure that registers are properly forwarded.

Team Member	Design Unit	Test
Danny	MEM	Check boundaries of the scratchpad (64, 8-bit entries by default)
Danny	MEM	Check that non-memory bound operations are simply forwarded

Writeback:

Testing that data created through the EXECUTE and MEM phases are properly written back to the register file. Cover all three possible sources propagated to output, input instruction, scratch pad and ALU result, as well as all register file destinations.

Covers:

Team Member	Design Unit	Test
Jim	WB	Check for Scratch Pad source
Jim	WB	Check for Input instruction source
Jim	WB	Check for ALU result source
Jim	WB	Check all registers written to

Asserations:

Team Member	Design Unit	Test
Jim	WB	WB destination and data always register file input
Jim	WB	If ALU source then must have had EX_enable asserted cycle prior
Jim	WB	Exwb_reg_source one hot

Jim	WB	If not enabled then not data is allowed to the register file
-----	----	--

Hazards:

Due to the simplicity of the micro architecture the amount of hazards is reduced to no structural hazards, one data hazard and several control hazards to verify. No structural hazards exist as fanouts are utilized in multi output modules as well as a separated instruction and data memory acting as the ROM and Scratch Pad. the only data hazard exists in a WAR, stalling the prior instruction until the dependency is written back to the register file. Finally, many control hazards are located in the fetch as due to the multiple conditional instructions that cannot fetch the instruction until the condition is evaluated in the execute phase.

Covers:

Team Member	Design Unit	Test
Jim	HAZ	Check all conditionals caused an interrupt for FETCH
Jim	HAZ	Check RAW hazard is identified and sent interrupt
Jim	HAZ	Check RAW can occur on both operands
Jim	HAZ	Check RAW can occur on single source instructions

Asserations:

Team Member	Design Unit	Test
Jim	HAZ	On interrupt the state of machine is returned to previous state
Jim	HAZ	No further interrupts after first has occurred, no nested interrupts
Jim	HAZ	After WB phase the WAR hazard is always solved

Exit Criteria:

Testing can be completed when individual units, transactions between units and top level are proven to behave as expected through formal assertions. Each unit in the DUT is proven through assertions guaranteeing behavior is as expected for all possible states of the DUT. Not only must each unit be proven to functional work as expected but also the transactions to other designs and top-level behavior. To prove transaction and communication to other units in the DUT assertions must be checked to verify timing

and data sent and received is valid. Finally top-level assertions must prove that each instruction to the microprocessor is stored, loaded, decoded and executed correctly before verification is complete.

Show stoppers to failure completion would be unable to verify each type of instruction, that being a store, load, arithmetic, jump or comparison. These are the fundamental operations of the design that must be verified.

ECE 571 Final Project – Verify a pipelined Picoblaze model and then find the defects we injected into the design using your verification environment

This assignment is worth 100 points. Final project presentations will be conducted live on either Tue, 06-Dec or Wed, 07-Dec . Presentation slots can be reserved via an online calendar. Deliverables are due on Thu, 08-Dec by 10:00 PM. NO LATE SUBMISSIONS WITHOUT PERMISSION

We will be using GitHub classroom for this assignment. You should submit your assignment before the deadline by pushing your final code and other deliverables to your team's GitHub repository for the assignment.

After completing this assignment students should have:

- Gained experience w/ unit level and system level design verification
- Gained experience w/ SystemVerilog verification techniques using checkers, assertions, randomization, etc
- Gained experience making a technical presentation

Final Project write-up:

[Final Project Assignment](#)

Summary

The final project is a chance to put what you learned this term into practice. Each team will perform unit-level and full-core testing on a pipelined Picoblaze model. You will implement a testbench (or testbenches) for simulation.

The pipelined Picoblaze (also called Rojoblaze) is a pipelined version of the Xilinx Picoblaze, an 8-bit microcontroller for Spartan-3, Virtex-II and Virtex-II Pro and Series 7 FPGAs. Rojoblaze was created by John D. Lynch and Roy Kravitz from Pablo Bleyer Kocik's Pacoblaze code base. PacoBlaze is a from-scratch synthesizable & behavioral Verilog clone of Ken Chapman's popular Picoblaze embedded microcontroller We are providing a SystemVerilog implementation of the Rojoblaze that was written by SethR, MilesS, ShubhankaSPM, and Supraj Vastrad for their ECE 571 Winter 2020 final project.

We envision that one of the keys to verifying this design will be writing and executing a suite of test programs using the KCASM (written in Java) or kcpsm3 (Windows) Assembler. Assertions will be the primary mechanism for checking that the pipelined Picoblaze works. Teams that have done this project in the past have made heavy use (often over 100 assertions) of assertion-based checking to flag defects in the implementation. The verification strategy and its implementation, however, is up to the team.

The final project will culminate in a technical presentation for the instructor and T/A. The teams will present their verification environment, verification strategy, verification approach, test results, and lessons learned during this presentation. Each team will have ~20 minutes to make their presentation. Depending on the size of the conference room for the final project presentations it may be possible for other teams to attend a presentation - we hope so.

We will be using the group project support in GitHub classroom for this assignment. This means that your team will share a private repository on GitHub that can also be accessed by the instructor and T/A for the course. You will submit your work via that repository and we would also like you to submit a .zip version of your repository to the team's Final Project dropbox on Canvas. We will review your work and provide feedback based on what your final submission was.

Where to submit your deliverables for the project:

- Verification Plan: Submit your Verification Plan to the appropriate dropbox on Canvas. Include a .pdf of the Verification Plan in your GitHub repository
- Verification Report: Submit your verification report with your final deliverables. The Verification Report should be a completed (test results included) version of your Verification plan. The Verification report should include the coverage

statistics that can be provided by QuestaSim, conclusions, lessons learned, next steps, and a work breakdown (which team member did what?)

- Demo presentation slides - Include a .pdf of your final project presentation in your final deliverables.
- Source code, makefiles, etc.: Include all of the source code you wrote for the project. This code should naturally be in your GitHub repository since you are committing your changes, merging your source code, etc. to GitHub.

Grading Rubric

This project is worth 100 points. There is the possibility of extra credit for projects and project presentations that stand out (in a good way)

- 30 pts: Verification Plan
- 40 pts: Final project presentation
- 25 pts: Quality and contents of your Design Report
- 5 pts: Quality/readability of your source code
- (up to) 5 pts: Extra Credit. Extra credit is just that - extra. Your documentation and presentation must be prepared, well-written, and well presented for the project to be eligible for extra credit

Broken model

We will provide a pipelined Picoblaze model that we've injected "subtle" bugs into as we get closer to the project due date. This code in this model will be encrypted so that you cannot perform a diff or code review between the working model and the broken model. How do you find the bugs? If you have a good verification strategy and good test cases they should identify the problems. If, not, you may need to add additional test cases.

Important Dates

- Wed, 09-Nov: Final project assigned during class
- Tue, 15-Nov: Pipelined Picoblaze project released to Canvas and GitHub Classroom
- Sat, 26-Nov: Verification plan submitted to Canvas by 10:00 PM
- Tue, 29-Nov: "Broken" model(s) released. The model(s) will be encrypted.
- Tue, 06-Dec: Final project presentations - day 1
- Wed, 07-Dec: Final project presentations - day 2
- Thu, 08-Dec: Final project deliverables due to Canvas and GitHub by 10:00pm

ECE 571 – Introduction to SystemVerilog

Fall 2022 Pipelined Picoblaze final project

Problem Statement:

You and your teammates (teams of 3) will test and verify a pipelined Picoblaze RTL model. The pipelined Picoblaze was written by John D. Lynch and Roy Kravitz in 2006 for a course they taught at OGI (the now defunct Oregon Graduate Institute) as the final project in a computer architecture course.

The Pipelined Picoblaze is based on the Pacoblaze model written by Pablo Bleyer Kocik. Pacoblaze is an RTL implementation of the Picoblaze Instruction Set Architecture. The Xilinx Picoblaze architecture was originally created by Ken Chapman, a Xilinx FAE, in 2003 and has been ported to many Xilinx FPGA families including the Series 7 FPGA used in ECE 540 and ECE 544.

For this project you will define, implement, debug, and make use of a verification environment that explores verification techniques such as constrained randomization, checkers, assertions, stimulus generators, etc. that we will discuss in ECE 571. You will write and execute a Verification plan using a SystemVerilog simulator (most likely QuestaSim).

Your team will be provided with a working (thoroughly verified by an ECE 571 final project teams in Fall 2021) SystemVerilog model of the Pipelined Picoblaze. Any testbenches or additional capability you add to the module must be written in SystemVerilog.

The focus for the final project is on verification, not on design.

You will make a 20-minute technical presentation of your project during Final's week which includes:

- Your verification environment, the approach you followed, and the techniques you applied
- The results you achieved, bugs found in the “broken” model and which test(s) found them.
- Challenges you encountered
- What additional work would do if you had more time.
- Pre-existing code/ideas leveraged (besides the Pipelined Picoblaze model)
- Assessment of the code coverage of your test suite (QuestaSim can help with that)

You will describe your verification strategy, including an outline of tests, assertions, and verification environment in a Verification plan. You will submit a Verification report based on your Verification plan summarizing your results, changes/enhancements from your original Verification plan and conclusions at the end of the project. The report should list the work done by each of the team members.

Hints for your final project team:

- From experience, the most significant challenge in forming and managing a project team is to understand how much time and effort each of the team members is willing, and able, to put into the project. Some students are extremely interested in the course and in mastering SystemVerilog and are willing to put in an extraordinary effort. Others are not. This is the first, and most important, discussion to have with your team before start on the project.
- Also of great interest is the whereabouts of the members of your team; that is, one or more of your team members may be taking the course remotely. If this is the case the team needs to agree on how they will communicate and coordinate the work. The project is too big for a single team member and all team members are expected to contribute a similar amount to the results of the project.
- You need to have a good project plan. This is not the Verification plan - it is how you are going to manage the project team to ensure that the project successfully meets the goals you set out for the team. A key element of planning is to discuss schedules and the work process. Your team will be using GitHub for this project, but how will you communicate? Slack? Discord? Text message? emails? How often will your team meet, and how? Will you use Zoom (all PSU students and staff have a Zoom license) or Google meetings? Will you meet in person with proper social distancing? Having a good plan is important for all projects, but it is even more important for a project like this with a limited, but somewhat undefined, scope and a fixed amount of time to complete the work.
- You will need to have a good schedule. List milestones (I like to list milestones with 3 – 5 day activities). Check your schedule for what I call “touch points” and what project management tools call “dependencies”. So often I see Gantt charts that have bars for each task and each individual but there are very few interdependencies. Most projects fail because the team underestimates the time it takes to integrate the test environment and/or because there is no time in the schedule to recover from problems. Consider allowing time for a 2nd pass through your Verification plan.
- The third thing to discuss is skill set of the team members. Look for complementary skill sets in your team. Who is good at hardware design? Who is good at coding? Who has an interest and a desire to find problems in code they didn’t write? Assign tasks based on skill set and interest as much as you can. In project management-speak this, and your schedule, is sometimes called the WBR...The Work Breakdown.

I can visualize you rolling your eyes when you read through this. After all, we’re talking about a 3-week project. We’re talking a single verification effort, not an entire chip design. However, you are going to put a lot of effort (at least I hope you will) into this project and planning for success is more likely to end up in a success (and a better final project grade) than doing things in an ad-hoc manner.

Presentation:

You will make a 20-minute presentation to the T/A and instructor during the Final's week. Each member of the team must make part of the presentation. Your presentation should briefly describe your design (block diagram, etc.) but delve more deeply into the verification strategy your results. *Of particular interest is the assertions you have placed in your design that found bugs.*

Deliverables:

- Your Verification Plan
- Your final project presentation materials (slides, etc.)
- A completed Verification Report. Start with your Verification plan and fill in the results of your testing. Which assertions were checked? Did your verification environment identify the bugs that we placed in the design? If so, how? If not, why not? What were the bugs? Your report should summarize your test coverage. Include conclusions, lessons learned, next steps if you had more time. List the contributions of each team member.
- All your source code including any scripts, programs, etc. you may have used to produce stimulus/result data for your verification effort. Source code should be structured with meaningful signal names and comments as appropriate. Each file should start with a header including the author and a short description of what the code in the files does. The header should acknowledge the source of any work that is not your own. You do not need to submit the Pipelined Picoblaze .sv (and .svp) files unless you made changes to them.
- If you have written Makefiles, scripts, etc. please include them so that somebody could reproduce your testing and your results. For example, if you use Assembly language test programs for unit level test/CPU level tests they could/should have their own Makefile.

Grading:

- (30 pts) Your Verification Plan. This plan will most likely be incomplete since it needs to be in place to implement your verification strategy.
- (40 pts) The final project presentation. Be sure to walk through your project, describing your test strategy and verification environment. Discuss bugs your testing caught and the way by which they are caught.

Make sure you practice your presentation before you present. 20 minutes may seem like forever, but it is a short time for a project of this magnitude presented by 3 verification engineers (that's you!). Make sure all team members present and all speak about their portion(s) of the project. Encourage questions.

- (25 pts) The quality of your Verification Report. Your report should effectively describe your verification methodology and summarize your results. Draw conclusions on what you

have accomplished and learned about your verification strategy and implementation. Start with your Verification Plan.

- (5 pts) Your source code should be organized with meaningful signal names. Make use of comments to describe aspects of your code that would be hard to understand from just reading the code. Make use of SystemVerilog constructs such as typedefs and structs, packages, assertions, and SystemVerilog verification constructs. Much of this grade will be based on the readability of your code; that is, can we get the gist of what you did by reading through, not studying the code in detail.
- (Up to 5 extra credit points) You can get extra credit by doing more than we asked for and/or for doing an excellent job. Extra credit is just that, though- Extra. You need to have a good verification strategy, a good plan, and a good presentation to be eligible for extra credit.

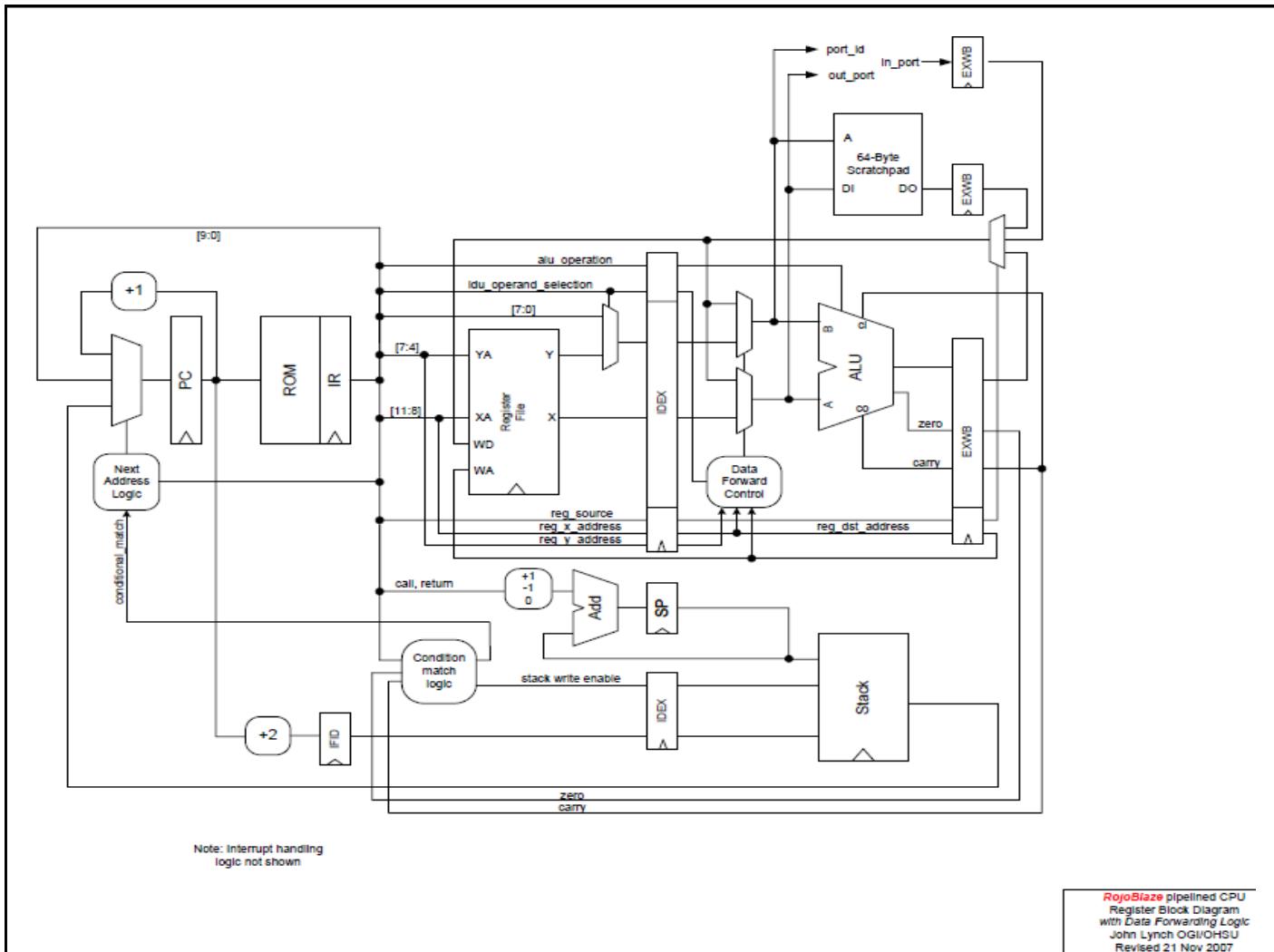
Important dates:

- Wed, 09-Nov: Final project assigned during class
- Tue, 15-Nov: Pipelined Picoblaze final project released to Canvas and GitHub
- Sun, 26-Nov: Verification plan submitted to Canvas by 10:00pm
- Tue, 29-Nov: Release of the “broken” Pipelined Picoblaze encrypted model(s)
- Tue, 06-Dec: Day 1 - Final project presentations (sign up for a time slot)
- Wed, 07-Dec: Day 2 - Final project presentations (sign up for a time slot)
- Thu, 08-Dec: Final project deliverables due to GitHub and D2L by 10:00pm

Introduce final project

What is the pipelined Picoblaze?

- Picoblaze is an 8-bit microcontroller “soft core” that can be incorporated into an FPGA-based SoC
 - Conceived and implemented by Ken Chapman, a Xilinx FAE, Circa 2010 in its current incarnation...but much older than that
 - Supported on many Xilinx FPGA families including the Series 7 FPGA families we target in ECE 540 and ECE 544
 - Synthesizable but not Verilog as we think of it...specified at the LUT-level
- Pacoblaze is a clone of Picoblaze but written as a synthesizable RTL model
 - Written and (was) supported by Pablo Bleyer Kocik (Last update was in 2007)
- Rojoblaze (Roy-John Picoblaze) is a pipelined version of the Pacoblaze written by John D. Lynch (<https://directory.vancouver.wsu.edu/people/john-lynch>) and Roy Kravitz, Circa 2007



Expectations

- Understand the ISA, write test cases, assemble the code using KCPSM and run the test cases on the core
- Understand the core and co-relate it with the ISA
- Write and execute a verification plan
 - Assertions, checkers, randomization, etc.
- Find bugs in a “broken” core.
- Produce a verification report based on your Verification plan
 - Should include coverage statistics from QuestaSim
- Present your findings in a 20-minute presentation on Tue, 06-Dec or Wed, 07-Dec
 - Signup calendar will be posted as we get closer to Final’s week

Self-enrolled teams of 3

Things to remember

5

- Teams of 3 - Github Classroom and Canvas
- *Please make sure all previously leveraged code is compiled and error free before executing your verification plan.*
- Grading:
 - (30 pts) Verification plan
 - (40 pts) Final project presentation
 - (25 pts) The quality of your design report and completed verification plan
 - (5 pts) The quality/readability of your source code
 - (up to 5) extra credit points

Important dates

- Wed, 09-Nov: Final project assigned during class
- Tue, 15-Nov: Pipelined Picoblaze project released to Canvas and GitHub Classroom
- Sat, 26-Nov: Verification plan submitted to Canvas by 10:00pm
- Tue, 29-Nov: “Broken” Picoblaze model released
- **Wed, 30-Nov: Final Exam (during class time)**
- Tue, 06-Dec, Wed, 07-Dec: Final Project Presentations
- Thu, 08-Dec: Final project deliverables due to GitHub and Canvas by 10:00pm

MIPS16 verification Draft

Person 1: Sai Bindu Konuri
Person 2: Monika Mullapudi
Person 3: Manjusha Ghanta
Person 4: Dhakshayini Koppad
Person 5: Abhishek Puttaswamy

Work Split up:

Person 1-5: Unit level testing, Writing assertions at unit level.
Person 1, 2, 5: CPU level testing, assemble code, log expected result
Person 3, 4: Emulation
Person 1-5: Makefile, Environment setup.
Person 5: Packaging the RTL, Run unit level and CPU level test.

Unit level Testing and the testcases and assertions for Unit level

We will be testing all 8 units naming IF, ID, EXE, MEM, WB (involved in the 5 stage MIPS16 pipeline), Hazard, register_file and CPU with 8 test benches. For unit level testing, we will be writing directed test cases by simulating the behavior of the neighboring blocks to cover functionality of each unit. Assertions will be used as checkers for unit level testing. A brief test plan for each unit is provided below.

Instruction Fetch:

Strategy: - Load the instruction memory through a memory map file, simulate the output behavior from the ID stage (branch, imm offset) and Hazard (fetch enable), check for valid output.

Test	Description
Test_reset	Assert reset, check PC=0 (assertions)
Test_instruction_enable	De assert instruction fetch enable, simulate for branch taken and branch not taken
Test_branch_taken	Assert instruction fetch enable and branch, test for + and – immediate displacement, zero displacement, +/- max displacement (assertions)
Test_branch_not_taken	Assert instruction fetch enable, de assert branch
Test_pc_rollover	Assert instruction fetch enable, assert branch, keep incrementing PC by giving + immediate displacement till it roll over from last address

Assertions

Assertion	Description
IF_inst_nx_assert	Check if the instruction from Instruction memory is not unknown, and instruction[15:12] is a valid code
IF_inst_valid_nop_assert	Check if the NOP instruction is valid, if instruction[15:12] = 0 then instruction = 0
IF_inst_valid_rtype_assert	Check for valid R- Type instruction, instruction [2:0]=0
IF_inst_valid_bz_assert	Check for valid Branch instruction, instruction [11:9]=0
IF_pc_inc_assert	Check if the PC is incremented, based on branch taken/immediate offset

Instruction Decode:

Strategy: - Send a valid instruction, valid data for the register read access request to simulate the behavior of IF (instruction), Hazard (decode_enable) and register file (reg_read_addr1/2). Check for the expected decoded output (write_back_enable, source/destination address, immediate field value, ALU command).

Test	Description
Test_reset	Assert reset, check instruction_reg = 0 (assertions)
Test_decode_en	De assert decode enable, send a valid instruction and check if it is executed or not
Test_register_instruction_decode	Assert decode enable, send valid R-type instructions and check a valid/corresponding ALU command and register address is decoded at the output (assertions)
Test_immediate_instruction_decode	Assert decode enable, send valid I-type instructions and check a valid/corresponding ALU command and register address and immediate field is decoded at the output
Test_branch_instruction_decode	Assert decode enable, send a branch instruction, test for branch taken/not taken
Test_invalid_instructions	Assert decode enable, send an invalid instruction and check if its not decoded.

Assertions:

Assertion	Description
ID_alu_cmd_nop_assert	To check if instruction is OP_NOP and alu_cmd is ALU_NOP and write_back_enable is 0 and mux_sel is X.
ID_alu_cmd_add_assert	To check if instruction is OP_ADD and alu_cmd is ALU_ADD and write_back_enable is 1 and mux_sel is 0.
ID_alu_cmd_sub_assert	To check if instruction is OP_SUB and alu_cmd is ALU_SUB and write_back_enable is 1 and mux_sel is 0.
ID_alu_cmd_and_assert	To check if instruction is OP_AND and alu_cmd is ALU_AND and write_back_enable is 1 and mux_sel is 0.

ID_alu_cmd_or_assert	To check if instruction is OP_OR and alu_cmd is ALU_OR and write_back_enable is 1 and mux_sel is 0.
ID_alu_cmd_xor_assert	To check if instruction is OP_XOR and alu_cmd is ALU_XOR and write_back_enable is 1 and mux_sel is 0. a
ID_alu_cmd_sl_assert	To check if instruction is OP_SL and alu_cmd is ALU_SL and write_back_enable is 1 and mux_sel is 0.
ID_alu_cmd_sr_assert	To check if instruction is OP_SR and alu_cmd is ALU_SR and write_back_enable is 1 and mux_sel is 0.
ID_alu_cmd_sru_assert	To check if instruction is OP_SRU and alu_cmd is ALU_SRU and write_back_enable is 1 and mux_sel is 0.
ID_alu_cmd_addi_assert	To check if instruction is OP_ADDI and alu_cmd is ALU_ADDI and write_back_enable is 1 and mux_sel is 0.
ID_alu_cmd_ld_assert	To check if instruction is OP_LD and alu_cmd is ALU_LD and write_back_enable is 1 and mux_sel is 1.
ID_alu_cmd_st_assert	To check if instruction is OP_ST and alu_cmd is ALU_ST and write_back_enable is 0 and mux_sel is x.
ID_alu_cmd_bz_assert	To check if instruction is OP_BZ and alu_cmd is ALU_BZ and write_back_enable is 0 and mux_sel is x.
ID_forward_rd1_op1_assert	Check read_data_1 is forwarded to operand1
ID_forward_rd2_mem_wr_data_assert	Check read_data_2 is forwarded to memory_write_data
ID_mem_wr_en_assert	Check if mem_write_en is asserted for ST instruction
ID_reg_addr_1_assert	Check if reg_read_address_1 is extracted from instruction
ID_reg_addr_2_assert	Check if reg_read_address_2 is extracted from instruction

Execute:

Strategy: This unit will be tested in conjunction with the ID (after ID unit level is clean), we will use the ID test stimulus to get a valid input to the EXE stage. Check for expected ALU operation result, and rest of the signals like write_back, memory write are forwarded to the next stage. We will be using assertions to check the ALU results.

Test	Description
Test_reset	Assert reset, check pipeline_reg_out = 0 (assertions)
Test_alu_operation	Send valid ALU command and operands, check for expected results (Assertions)
Test_invalid_alu_operation	Send invalid ALU command and check results

Assertions for Execute Stage:

Assertion	Description
alu_result_assert	To check if selected part of output is evaluated one clock tick after the ALU result ends when reset is disabled.
pipeline_reg_write_data_assert	To check if selected part of output is evaluated one clock tick after the pipeline_reg_in result ends when reset is disabled.

Assertions for ALU:

Assertion	Description
alu_nc_assert	Checking the result for the corresponding command ALU_NC and checking the bits a and b contain X or Z.
alu_add_assert	Checking the result for the corresponding command ALU_ADD and checking the bits a and b contain X or Z.
alu_sub_assert	Checking the result for the corresponding command ALU_SUB and checking the bits a and b contain X or Z.
alu_and_assert	Checking the result for the corresponding command ALU_AND and checking the bits a and b contain X or Z.
alu_or_assert	Checking the result for the corresponding command ALU_OR and checking the bits a and b contain X or Z.
alu_xor_assert	Checking the result for the corresponding command ALU_XOR and checking the bits a and b contain X or Z.
alu_sl_assert	Checking the result for the corresponding command ALU_SL and checking the bits a and b contain X or Z.
alu_sr_assert	Checking the result for the corresponding command ALU_SR and checking the bits a and b contain X or Z.
alu_sru_assert	Checking the result for the corresponding command ALU_SRU and checking the bits a and b contain X or Z.

Memory Stage:

Strategy: - We are going to test the data memory in this unit testing. Load the data memory with random values/memory map file (same memory image will be loaded to checker memory), drive valid address, toggle write enable and perform read/write. Check the read data with the checker memory copy. Assertion for checking data_in [4:0] == data_out[4:0]

Test	Description
Test_reset	Assert reset, check pipeline_reg_out = 0 (assertions)
Test_write_operation	Assert write enable, send valid address and data, check the data_out is same as data provided for write
Test_read_operation	De-assert write enable, send valid address and check whether data out is same as checker memory copy.

Assertions:

Assertion	Description
pipeline_reg_addr_assert	To check if selected part of pipeline_reg_out is evaluated one clock tick after the selected part of pipeline_reg_in ends and when reset is disabled.
pipeline_reg_write_data_assert	To check if selected part of pipeline_reg_out is evaluated one clock tick after the selected part of pipeline_reg_in ends and when reset is disabled.
pipeline_reg_in_out_assert	To check if selected part of pipeline_reg_out is evaluated one clock tick after the selected part of pipeline_reg_in ends and when reset is disabled.

Write Back Stage:

Strategy: - In this unit we will be testing the ALU result/Data write back mux. Toggle write back mux control to switch between data to be written to register file, drive valid alu result/memory read data, drive destination register address and check corresponding data/address/write_en appears at the output. Use assertions to check all the outputs.

Test	Description
Test_toggle_mux	Toggle write back mux control and check if reg_write_data is selected based on selection input Write to all destination registers

Assertions:

Assertion	Description
Write_back_result_mux1_assert	To check if pipeline_reg_in[20:5] is taken by reg_write_data when write_back_result_mux is 1.
Write_back_result_mux0_assert	To check if pipeline_reg_in[36:21] is taken by reg_write_data when write_back_result_mux is 0.

Register file:

Strategy: - We are testing the behavior of simultaneous single write, dual read memory in this unit. Drive the inputs to valid address/data check for read/write behavior. A checker memory array will be used to compare the results of read/write.

Test	Description
Test_reset	Assert reset, check all register values = 0 (assertions)
Test_write_operation	Assert write enable, send valid address and data, do a read to same address and check if the data was written successfully
Test_read_operation	<p>De-assert write enable, send valid address to both the read port, check if the data received matches the checker memory</p> <p>De-assert write enable, send same address to both the read port, check if the data received matches the checker memory</p> <p>De assert write enable, read to destination register 000 and check the result of read is zero</p>
Test_read_write_same_address	Assert write enable, make read and write address same. Check data written appears on read port.

Assertions:

Assertion	Description
Wb_reset_assert	Check all the registers are reset to 0
Wb_write_nx	Check if the data written to register on write_en is valid (no x's)
wb_read_r0	Check if the read to R0 always returns 0

Hazard:

Strategy: - Drive valid source register address (ID stage) and destination register address (EXE/MEM/WB). Check for pipeline stall condition. Assertions will be used to monitor active low pipeline stall signal.

Test	Description
Test_source_zero	Source1/source2 register address = 0
Test_source_non_zero_stall	<p>Source1/source2 register address!=0, but source1 = EXE/MEM/WB destination address</p> <p>Source1/source2 register address!=0, but source2 = EXE/MEM/WB destination address</p>
Test_source_non_zero_no_stall	<p>Source1/source2 register address!=0, but source1 != EXE/MEM/WB destination address</p> <p>Source1/source2 register address!=0, but source2 != EXE/MEM/WB destination address</p>

Assertions:

Assertion	Description
Hz_src1_hazard	Check if the pipeline_stall_n is asserted for RAW hazard from operand 1
Hz_src2_hazard	Check if the pipeline_stall_n is asserted for RAW hazard from operand 2

CPU level verification and Test plan:

For top level testing we will be writing code snippets in MIPS16 assembly, use the software model to get the instruction code. Load the Instruction memory and monitor the Registers.

The expected result (Register values) for each test will be logged in a file, and the results after each simulation will be compared against it

Test	Description
Test_reset	Assert reset, check registers
Test_ISA	<p>Test each instruction individually, checks expected result from the register file.</p> <p>R-type instruction: Test involving access to all 8 registers, for each instruction Test the access to R0 special register.</p> <p>I-type Instruction: Test LD/ST instruction Test for branch instructions, +/- displacement, rollover from last accessible address. Test for +/-0 immediate values</p>
Test_snippets	Write a MIPS16 assembly code snippets and test for expected behavior
Test_Hazard	Write a assembly code with RAW pipeline hazards and check for pipeline stalls.

Unit level assertions will be binded for top CPU level verification.

Emulation:

We have to setup the design for emulation. For this setup, we should create a design directory on our compile host or file server. After adding HDL, HVL and source files, we should setup and map analysis libraries and create veloce.config file. Now, when the veloce.config file with all the compilation options is in the current design directory, we should run the velocomp from the top level of our design directory.

Packaging of RTL and Verify:

Replace the interconnections between the pipeline stages with interface/mod ports. Individually test each unit using the test bench created for the RTL released. The RTL from git will be used as a reference model for the testing. Stitch all the units together and run CPU level testing on the top module.

Find any chance of parameterizing/encapsulation in the RTL

Makefile, Environment Setup, project package:

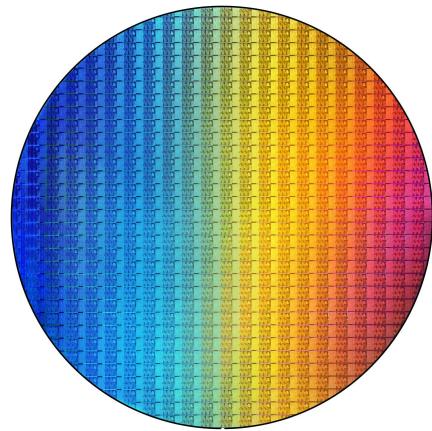
All the unit level test/CPU level testing will have its own Makefile, the make file is capable of reading the Assembly code, load the instruction memory and chose test to be run. Individual who owns the unit level test will create their own Makefile and environment setup.

Setup for Emulator is done by person who owns emulation part of the project.

Portland State University

ECE571 - SystemVerilog

Winter 2019



MIPS16 ISA Verification Plan

R. Ignacio Genovese
Chenyang Li
Shouvik Rakshit
Aishwarya Doosa

Introduction

As digital designs get bigger, the effort in verification can scale up to 70% or even 80% of the total design effort, thereby making it the most expensive step in the entire IC design flow. Besides, any behavioral or functional bugs escaping this phase will surface only after the silicon is integrated into the target system, resulting in even more costly design and silicon iterations. For this reason, nowadays it is essential for engineers to learn different approaches to overcome these bugs, getting to successfully implement verification environments aimed to reduce these costs.

Therefore, as final project for the course *ECE571 - Introduction to SystemVerilog for Design and Verification* at Portland State University, the team will develop and implement a Verification Plan for a MIPS16 ISA.

Verification Approach

Instead of using a set of functional specifications as base for our Verification Plan, we will learn the architecture and requirements from a completely functional MIPS 16 ISA design. This will be used to define a set of **unit tests** (by assembly code) that will serve as stimulus for the processor core. After running these tests, the QuestaSim **coverage tool** will be used to determine if this set of testcases is enough to get a good (branch, statement, fsm, toggle) coverage. Based on the results from the coverage tool, we will develop new testcases to get a better coverage in case it's necessary.

To assure the functional correctness of these testcases (besides taking it for granted as the design is supposed to be correct), the final state of the register file and memory will be checked, as each unit test should be simple enough to know this result.

Once we get a good set of tests, we will proceed to save into files the inputs and outputs of each of the 5 pipeline stages (Instruction Fetch, Instruction Decode, Execute, Memory and Writeback), in order to carry on with the verification of each of these stages independently. These files will then be used to stimulate each stage and compare its outputs.

Along with the unit tests, a set of **assertions** for each pipeline stage will be defined in a separate file and bound to the design.

Finally, this coverage and assertion based verification environment will be used to verify a “broken” design, in order to prove its effectiveness and implementation to find bugs.

Unit Tests

After reviewing and understanding the implemented MIPS16 ISA, we proceeded to consider each stage inputs to be stimulated and defined the following basic unit tests to implement:

- IF:
 - Implement branch taken and branch not taken.
 - Try the instruction memory boundaries using different offsets.
 - Produce stalls in the hazard detection unit.
- ID:
 - Implement all type of instructions using every register as destination, source operand 1 and source operand 2. There are 13 types of instructions:
 - OP_NOP
 - OP_ADD
 - OP_SUB
 - OP_AND
 - OP_OR
 - OP_XOR
 - OP_SL
 - OP_SR
 - OP_SRU
 - OP_ADDI
 - OP_LD
 - OP_ST
 - OP_BZ
- EX:
 - Test all ALU operations using every register as destination, source operand 1 and source operand 2. There are 8 ALU operations:
 - ADD
 - SUB
 - AND
 - OR
 - XOR
 - SL (shift left)
 - SR (shift right, preserving sign bit)
 - SRU (shift right unsigned)

- MEM:
 - Implement load and store operations to every memory position, using every register as source operand 1 (base), source operand 2 (offset) and destination register.
 - Consider that writes to register 0 (R0) will always produce a zero.
- WB:
 - Write back every register with results from the ALU and values read from the memory.
- Hazard detection unit:
 - Produce every possible stall using every register as destination, source operand 1 and source operand 2. There are 3 possible stalls:
 - When a source operand for the current instruction is the destination operand for the instruction in the EX stage.
 - When a source operand for the current instruction is the destination operand for the instruction in the MEM stage.
 - When a source operand for the current instruction is the destination operand for the instruction in the WB stage.

Testcases

Name	Stage	Description	Owner
hazard_r0.asm	Hazard Detection	Produce every possible stall using R0 as destination, source operand 1 and source operand 2.	Ignacio Genovese
hazard_r1.asm	Hazard Detection	Produce every possible stall using R1 as destination, source operand 1 and source operand 2.	Ignacio Genovese
hazard_r2.asm	Hazard Detection	Produce every possible stall using R2 as destination, source operand 1 and source operand 2.	Ignacio Genovese
hazard_r3.asm	Hazard Detection	Produce every possible stall using R3 as destination, source operand 1 and source operand 2.	Ignacio Genovese

hazard_r4.asm	Hazard Detection	Produce every possible stall using R4 as destination, source operand 1 and source operand 2.	Ignacio Genovese
hazard_r5.asm	Hazard Detection	Produce every possible stall using R5 as destination, source operand 1 and source operand 2.	Ignacio Genovese
hazard_r6.asm	Hazard Detection	Produce every possible stall using R6 as destination, source operand 1 and source operand 2.	Ignacio Genovese
hazard_r7.asm	Hazard Detection	Produce every possible stall using R7 as destination, source operand 1 and source operand 2.	Ignacio Genovese
hazard_detection.asm	Hazard Detection	Normal Operation test, producing every possible hazard.	Ignacio Genovese
R0_load_store	Memory, Write Back & Instruction Decode	Perform memory store from register R0 to 256 locations of the memory. Performs a load back to the register from memory.	Shouvik Rakshit
R1_load_store	Memory, Write Back & Instruction Decode	Perform memory store from register R1 to 256 locations of the memory. Performs a load back to the register from memory.	Shouvik Rakshit
R2_load_store	Memory, Write Back & Instruction Decode	Perform memory store from register R2 to 256 locations of the memory. Performs a load back to the register from memory.	Shouvik Rakshit
R3_load_store	Memory, Write Back & Instruction Decode	Perform memory store from register R3 to 256 locations of the memory. Performs a load back to the register from memory.	Shouvik Rakshit
R4_load_store	Memory, Write Back & Instruction Decode	Perform memory store from register R4 to 256 locations of the memory. Performs a load back to the register from memory.	Shouvik Rakshit
R5_load_store	Memory, Write Back & Instruction	Perform memory store from register R5 to 256 locations of the memory. Performs a load back to the register from	Shouvik Rakshit

	Decode	memory.	
R6_load_store	Memory, Write Back & Instruction Decode	Perform memory store from register R6 to 256 locations of the memory. Performs a load back to the register from memory.	Shouvik Rakshit
R7_load_store	Memory, Write Back & Instruction Decode	Perform memory store from register R7 to 256 locations of the memory. Performs a load back to the register from memory.	Shouvik Rakshit
add.asm	Execution	Performs addition of two operands using every register as a destination and source.	Aishwarya Doosa
sub.asm	Execution	Performs subtraction of two registers using every register as a destination and source.	Aishwarya Doosa
and.asm	Execution	Performs bitwise and operation between two registers using every register as a destination and source.	Aishwarya Doosa
or.asm	Execution	Performs bitwise or operation between two registers using every register as a destination and source.	Aishwarya Doosa
xor.asm	Execution	Performs bitwise exor operation between two registers using every register as a destination and source.	Aishwarya Doosa
sl.asm	Execution	Performs logical shift left operation using every register as a destination and source.	Aishwarya Doosa
sr.asm	Execution	Performs logical shift right operation using every register as a destination and source.	Aishwarya Doosa
sru.asm	Execution	Performs unsigned shift right operation using every register as a destination and source.	Aishwarya Doosa

branch_taken.asm	Instruction Fetch & Instruction Decode	Branch operation test, test branch taken	Chenyang Li
branch_not_taken.asm	Instruction Fetch & Instruction Decode	Branch operation test, test branch not taken	Chenyang Li
nop.asm	Instruction Fetch & Instruction Decode	NOP operation test, test nop instruction	Chenyang Li
add_r1.asm	Instruction Fetch & Instruction Decode	Single instruction test for addition of two operands using r1 register as a destination and source operand 1 and source operand 2.	Chenyang Li
add_r2.asm	Instruction Fetch & Instruction Decode	Single instruction test for addition of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
add_r3.asm	Instruction Fetch & Instruction Decode	Single instruction test for addition of two operands using r3 register as a destination and source operand 1 and source operand 2.	Chenyang Li
add_r4.asm	Instruction Fetch & Instruction Decode	Single instruction test for addition of two operands using r4 register as a destination and source operand 1 and source operand 2.	Chenyang Li
add_r5.asm	Instruction Fetch & Instruction Decode	Single instruction test for addition of two operands using r5 register as a destination and source operand 1 and source operand 2.	Chenyang Li
add_r6.asm	Instruction Fetch & Instruction Decode	Single instruction test for addition of two operands using r6 register as a destination and source operand 1 and source operand 2.	Chenyang Li
add_r7.asm	Instruction Fetch & Instruction Decode	Single instruction test for addition of two operands using r7 register as a destination and source operand 1 and source operand 2.	Chenyang Li

sub_r1.asm	Instruction Fetch & Instruction Decode	Single instruction test for subtraction of two operands using r1 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sub_r2.asm	Instruction Fetch & Instruction Decode	Single instruction test for subtraction of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sub_r3.asm	Instruction Fetch & Instruction Decode	Single instruction test for subtraction of two operands using r3 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sub_r4.asm	Instruction Fetch & Instruction Decode	Single instruction test for subtraction of two operands using r4 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sub_r5.asm	Instruction Fetch & Instruction Decode	Single instruction test for subtraction of two operands using r5 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sub_r6.asm	Instruction Fetch & Instruction Decode	Single instruction test for subtraction of two operands using r6 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sub_r7.asm	Instruction Fetch & Instruction Decode	Single instruction test for subtraction of two operands using r7 register as a destination and source operand 1 and source operand 2.	Chenyang Li
and_r1.asm	Instruction Fetch & Instruction Decode	Single instruction test for AND of two operands using r1 register as a destination and source operand 1 and source operand 2.	Chenyang Li
and_r2.asm	Instruction Fetch & Instruction Decode	Single instruction test for AND of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
and_r3.asm	Instruction Fetch & Instruction Decode	Single instruction test for AND of two operands using r3 register as a destination and source operand 1 and source operand 2.	Chenyang Li

and_r4.asm	Instruction Fetch & Instruction Decode	Single instruction test for AND of two operands using r4 register as a destination and source operand 1 and source operand 2.	Chenyang Li
and_r5.asm	Instruction Fetch & Instruction Decode	Single instruction test for AND of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
and_r6.asm	Instruction Fetch & Instruction Decode	Single instruction test for AND of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
and_r7.asm	Instruction Fetch & Instruction Decode	Single instruction test for AND of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
or_r1.asm	Instruction Fetch & Instruction Decode	Single instruction test for OR of two operands using r1 register as a destination and source operand 1 and source operand 2.	Chenyang Li
or_r2.asm	Instruction Fetch & Instruction Decode	Single instruction test for OR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
or_r3.asm	Instruction Fetch & Instruction Decode	Single instruction test for OR of two operands using r3 register as a destination and source operand 1 and source operand 2.	Chenyang Li
or_r4.asm	Instruction Fetch & Instruction Decode	Single instruction test for OR of two operands using r4 register as a destination and source operand 1 and source operand 2.	Chenyang Li
or_r5.asm	Instruction Fetch & Instruction Decode	Single instruction test for OR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
or_r6.asm	Instruction Fetch & Instruction Decode	Single instruction test for OR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li

or_r7.asm	Instruction Fetch & Instruction Decode	Single instruction test for OR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
xor_r1.asm	Instruction Fetch & Instruction Decode	Single instruction test for XOR of two operands using r1 register as a destination and source operand 1 and source operand 2.	Chenyang Li
xor_r2.asm	Instruction Fetch & Instruction Decode	Single instruction test for XOR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
xor_r3.asm	Instruction Fetch & Instruction Decode	Single instruction test for XOR of two operands using r3 register as a destination and source operand 1 and source operand 2.	Chenyang Li
xor_r4.asm	Instruction Fetch & Instruction Decode	Single instruction test for XOR of two operands using r4 register as a destination and source operand 1 and source operand 2.	Chenyang Li
xor_r5.asm	Instruction Fetch & Instruction Decode	Single instruction test for XOR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
xor_r6.asm	Instruction Fetch & Instruction Decode	Single instruction test for XOR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
xor_r7.asm	Instruction Fetch & Instruction Decode	Single instruction test for XOR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sl_r1.asm	Instruction Fetch & Instruction Decode	Single instruction test for SL of two operands using r1 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sl_r2.asm	Instruction Fetch & Instruction Decode	Single instruction test for SL of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li

sl_r3.asm	Instruction Fetch & Instruction Decode	Single instruction test for SL of two operands using r3 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sl_r4.asm	Instruction Fetch & Instruction Decode	Single instruction test for SL of two operands using r4 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sl_r5.asm	Instruction Fetch & Instruction Decode	Single instruction test for SL of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sl_r6.asm	Instruction Fetch & Instruction Decode	Single instruction test for SL of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sl_r7.asm	Instruction Fetch & Instruction Decode	Single instruction test for SL of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sr_r1.asm	Instruction Fetch & Instruction Decode	Single instruction test for SR of two operands using r1 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sr_r2.asm	Instruction Fetch & Instruction Decode	Single instruction test for SR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sr_r3.asm	Instruction Fetch & Instruction Decode	Single instruction test for SR of two operands using r3 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sr_r4.asm	Instruction Fetch & Instruction Decode	Single instruction test for SR of two operands using r4 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sr_r5.asm	Instruction Fetch & Instruction Decode	Single instruction test for SR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li

sr_r6.asm	Instruction Fetch & Instruction Decode	Single instruction test for SR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sr_r7.asm	Instruction Fetch & Instruction Decode	Single instruction test for SR of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sru_r1.asm	Instruction Fetch & Instruction Decode	Single instruction test for XOR of two operands using r1 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sru_r2.asm	Instruction Fetch & Instruction Decode	Single instruction test for SRU of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sru_r3.asm	Instruction Fetch & Instruction Decode	Single instruction test for SRU of two operands using r3 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sru_r4.asm	Instruction Fetch & Instruction Decode	Single instruction test for SRU of two operands using r4 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sru_r5.asm	Instruction Fetch & Instruction Decode	Single instruction test for SRU of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sru_r6.asm	Instruction Fetch & Instruction Decode	Single instruction test for SRU of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
sru_r7.asm	Instruction Fetch & Instruction Decode	Single instruction test for SRU of two operands using r2 register as a destination and source operand 1 and source operand 2.	Chenyang Li
addi.asm	Instruction Fetch & Instruction Decode	Single instruction test for ADDI of two operands using r1 register as a destination and r2 as operand 1 and an immediate number.	Chenyang Li

Assertions

Property Name	Stage	Description	Owner
stall_length	Hazard Detection	Stall shouldn't last more than three cycles	Ignacio Genovese
stall_operation	Hazard Detection	Stall should be produced if a source operand is the destination operand for the current instruction in the EX, MEM or WB stage	Ignacio Genovese
R0_operation	Register File	Every time register 0 is read, the output should be 0	Ignacio Genovese
RF_write	Register File	If register write is enabled, then the reg_write_data value should be written to the register destination	Ignacio Genovese
RF_read	Register File	For each input register address, the output should be the content of that register	Ignacio Genovese
Mem_write	Memory	If write mem is enabled, then the mem_write_data value should be written to the memory address	Ignacio Genovese
Mem_read	Memory	The output of the memory should be the value stored in the indicated memory address	Ignacio Genovese
pc_increment	Instruction Fetch	If there's no stall and no branch is taken, the PC should increment on each cycle	Chenyang Li
pc_stall	Instruction Fetch	If there's a stall, the PC should remain stable	Chenyang Li
pc_branch	Instruction Fetch	If there's no stall and a branch is taken, the PC should change to the correct offset	Chenyang Li
id_stall	Instruction Decode	If there's a stall, the instruction register shouldn't change	Chenyang Li
id_op_stall	Instruction Decode	If there's a stall, the opcode should be 0	Chenyang Li
id_dest_stall	Instruction Decode	If there's a stall, the destination register should be 0	Chenyang Li
id_wb_en	Instruction Decode	Check that the correct value is given to the write_back_en according to the current opcode	Chenyang Li
id_wb_mux	Instruction Decode	Check that the correct value is given to the write_back_result_mux according to the current opcode	Chenyang Li
id_alu_cmd	Instruction	Check that the correct value is given to the	Chenyang

	Decode	ex_alu_cmd according to the current opcode	Li
id_alu_src2_mux	Instruction Decode	Check that the correct value is given to the alu_src2_mux according to the current opcode	Chenyang Li
id_op_src2	Instruction Decode	Check that the correct value is given to the decoding_op_src2 according to the current opcode	Chenyang Li
alu_add	Execution	Check that the output corresponds to the sum of the inputs	Aishwayra Doosa
alu_sub	Execution	Check that the output corresponds to the subtraction of the inputs	Aishwayra Doosa
alu_and	Execution	Check that the output corresponds to the and of the inputs	Aishwayra Doosa
alu_or	Execution	Check that the output corresponds to the or of the inputs	Aishwayra Doosa
alu_xor	Execution	Check that the output corresponds to the xor of the inputs	Aishwayra Doosa
alu_sl	Execution	Check that the output corresponds to the correct shift left of the input	Aishwayra Doosa
alu_sr	Execution	Check that the output corresponds to the correct shift right of the input	Aishwayra Doosa
alu_sru	Execution	Check that the output corresponds to the correct unsigned shift right of the input	Aishwayra Doosa
wb_data	Writeback	Check that the correct data is selected to be written back	Shouvik Rakshit

Verification Environment

For the first phase of the verification flow (top level verification) there will be a testcase folder containing each stage testcases (bench\top\testcases). Some of these testcases will be shared between stages, as they are the same (for example, every instruction decoding will include ALU instructions, or stalls for the IF stage will be produced in the hazard detection unit testcases). A top level testbench will be used, which will have a string array containing every testcase file path and the expected results (register file and memory contents) for every unit test (used at the end of each test for proving functional correctness).

For the second phase (testing each pipeline stage separately), there will be a folder for each pipeline stage containing the input/output files for stimulating and comparing results (results\saved_regs). Each independent stage will have its own separate testbench (inside the bench folder).

For both of these phases there will be assertions that run along each testbench.

Also, a Makefile will be used for compiling and running each testbench (in the sim folder).

Along with this, there will be a results folder containing the coverage, simulation and assertions results for each testbench, and the saved inputs and outputs of each stage.

Finally, for the third stage, this whole structure will be reproduced to test the “broken” design.

File structure

Folder	Contents
bench	
└── EX	Execution stage testbench
└── ID	Instruction Decode stage testbench
└── IF	Instruction Fetch stage testbench
└── MEM	Memory stage testbench
└── WB	Writeback stage testbench
└── hazard_detection	Hazard Detection Unit testbench
└── register_file	Register File testbench

└── top	Top level testbench
└── testcases	Assembly testcases and expected outputs
└── docs	Documents folder
└── mips_16	
└── bench	Example testbenches
└── doc	MIPS 16 documentation
└── rtl	DUT source code
└── sw	Software tools, like java assembler
└── results	
└── EX	Execution stage coverage reports
└── ID	Instruction Decode stage coverage reports
└── IF	Instruction Fetch stage coverage reports
└── MEM	Memory stage coverage reports
└── WB	Writeback stage coverage reports
└── hazard_detection	Hazard Detection Unit coverage reports
└── saved_regs	Files containing the output of each stage for every testcase
└── top	Top level coverage reports
└── sim	
└── EX	Execution stage Makefile and simulation scripts
└── ID	Instruction Decode stage Makefile and simulation scripts
└── IF	Instruction Fetch Makefile and simulation scripts
└── MEM	Memory stage Makefile and simulation scripts
└── WB	Writeback stage Makefile and simulation scripts
└── hazard_detection	Hazard Detection Unit Makefile and simulation scripts
└── register_file	Register File Makefile and simulation scripts

<pre> └── top └── veloce </pre>	Top level Makefile and simulation scripts Veloce Makefile and simulation scripts
-------------------------------------	---

Testbenches

- *Top Level Testbench*: this testbench runs all of the testcases (loading the output of the java assembler into the instruction memory), compares the final state of the register file and memory with the expected results for each test and saves the inputs and the outputs of each stage for every test. Owner: All team members
- *Instruction Fetch Testbench*: this testbench uses the saved inputs of the IF testcases to stimulate the instruction fetch stage and compares the obtained outputs with the saved outputs of the IF testcases. Owner: Chenyang Li
- *Instruction Decode Testbench*: this testbench uses the saved inputs of the ID testcases to stimulate the instruction fetch stage and compares the obtained outputs with the saved outputs of the IF testcases. Owner: Chenyang Li
- *Execution Testbench*: this testbench uses the saved inputs of the IF testcases to stimulate the instruction fetch stage and compares the obtained outputs with the saved outputs of the IF testcases. Owner: Aishwarya Doosa
- *Memory Testbench*: this testbench uses the saved inputs of the IF testcases to stimulate the instruction fetch stage and compares the obtained outputs with the saved outputs of the IF testcases. Owner: Shouvik Rakshit
- *Writeback Testbench*: this testbench uses the saved inputs of the IF testcases to stimulate the instruction fetch stage and compares the obtained outputs with the saved outputs of the IF testcases. Owner: Shouvik Rakshit
- *Hazard Detection Unit Testbench*: this testbench uses the saved inputs of the IF testcases to stimulate the instruction fetch stage and compares the obtained outputs with the saved outputs of the IF testcases. Owner: Ignacio Genovese
- *Register File Testbench*: this testbench uses the saved inputs of the IF testcases to stimulate the instruction fetch stage and compares the obtained outputs with the saved outputs of the IF testcases. Owner: Ignacio Genovese

Team members responsibilities

For each stage, the designated member will create the assembly code, produce the prog file using the java assembler, create the expected register file and memory results and create and run each independent testbench.

- ID: Chenyang Li
- IF: Chenyang Li
- EX: Aishwarya Doosa

- MEM: Shouvik Rakshit
- WB: Shouvik Rakshit
- Hazard Detection Unit: R. Ignacio Genovese
- Register File: R. Ignacio Genovese
- Top level integration, makefiles, coverage and assertions results: R. Ignacio Genovese
- Veloce (set environment, run examples, run MIPS16 ISA): Aishwarya Doosa/R. Ignacio Genovese

MIPS16 verification Draft

Person 1: Sai Bindu Konuri
Person 2: Monika Mullapudi
Person 3: Manjusha Ghanta
Person 4: Dhakshayini Koppad
Person 5: Abhishek Puttaswamy

Work Split up:

Person 1-5: Unit level testing, Writing assertions at unit level.
Person 1, 2, 5: CPU level testing, assemble code, log expected result
Person 3, 4: Emulation
Person 1-5: Makefile, Environment setup.
Person 5: Packaging the RTL, Run unit level and CPU level test.

Unit level Testing and the testcase for Unit level

We will be testing all 8 units naming IF, ID, EXE, MEM, WB (involved in the 5 stage MIPS16 pipeline), Hazard, register_file and CPU with 8 test benches. For unit level testing, we will be writing directed test cases by simulating the behavior of the neighboring blocks to cover functionality of each unit. Assertions will be used as checkers for unit level testing. A brief test plan for each unit is provided below.

Instruction Fetch:

Strategy: - Load the instruction memory through a memory map file, simulate the output behavior from the ID stage (branch, imm offset) and Hazard (fetch enable), check for valid output.

Test	Description
Test_reset	Assert reset, check PC=0 (assertions)
Test_instruction_enable	De assert instruction fetch enable, simulate for branch taken and branch not taken
Test_branch_taken	Assert instruction fetch enable and branch, test for + and – immediate displacement, zero displacement, +/- max displacement (assertions)
Test_branch_not_taken	Assert instruction fetch enable, de assert branch
Test_pc_rollover	Assert instruction fetch enable, assert branch, keep incrementing PC by giving + immediate displacement till it roll over from last address

Instruction Decode:

Strategy: - Send a valid instruction, valid data for the register read access request to simulate the behavior of IF (instruction), Hazard (decode_enable) and register file (reg_read_addr1/2). Check for the expected decoded output (write_back_enable, source/destination address, immediate field value, ALU command).

Test	Description
Test_reset	Assert reset, check instruction_reg = 0 (assertions)
Test_decode_en	De assert decode enable, send a valid instruction and check if it is executed or not
Test_register_instruction_decode	Assert decode enable, send valid R-type instructions and check a valid/corresponding ALU command and register address is

	decoded at the output (assertions)
Test_immediate_instruction_decode	Assert decode enable, send valid I-type instructions and check a valid/corresponding ALU command and register address and immediate field is decoded at the output
Test_branch_instruction_decode	Assert decode enable, send a branch instruction, test for branch taken/not taken
Test_invalid_instructions	Assert decode enable, send an invalid instruction and check if its not decoded.

Execute:

Strategy: This unit will be tested in conjunction with the ID (after ID unit level is clean), we will use the ID test stimulus to get a valid input to the EXE stage. Check for expected ALU operation result, and rest of the signals like write_back, memory write are forwarded to the next stage. We will be using assertions to check the ALU results.

Test	Description
Test_reset	Assert reset, check pipeline_reg_out = 0 (assertions)
Test_alu_operation	Send valid ALU command and operands, check for expected results (Assertions)
Test_invalid_alu_operation	Send invalid ALU command and check results

Memory Stage:

Strategy: - We are going to test the data memory in this unit testing. Load the data memory with random values/memory map file (same memory image will be loaded to checker memory), drive valid address, toggle write enable and perform read/write. Check the read data with the checker memory copy. Assertion for checking data_in [4:0] == data_out[4:0]

Test	Description
Test_reset	Assert reset, check pipeline_reg_out = 0 (assertions)
Test_write_operation	Assert write enable, send valid address and data, check the data_out is same as data provided for write
Test_read_operation	De-assert write enable, send valid address and check whether data out is same as checker memory copy.

Write Back Stage:

Strategy: - In this unit we will be testing the ALU result/Data write back mux. Toggle write back mux control to switch between data to be written to register file, drive valid alu result/memory read data, drive destination register address and check corresponding data/address/write_en appears at the output. Use assertions to check all the outputs.

Test	Description
Test_toggle_mux	Toggle write back mux control and check if reg_write_data is selected based on selection input Write to all destination registers

Register file:

Strategy: - We are testing the behavior of simultaneous single write, dual read memory in this unit. Drive the inputs to valid address/data check for read/write behavior. A checker memory array will be used to compare the results of read/write.

Test	Description
Test_reset	Assert reset, check all register values = 0 (assertions)
Test_write_operation	Assert write enable, send valid address and data, do a read to same address and check if the data was written successfully
Test_read_operation	De-assert write enable, send valid address to both the read port, check if the data received matches the checker memory De-assert write enable, send same address to both the read port, check if the data received matches the checker memory De assert write enable, read to destination register 000 and check the result of read is zero
Test_read_write_same_address	Assert write enable, make read and write address same. Check data written appears on read port.

Hazard:

Strategy: - Drive valid source register address (ID stage) and destination register address (EXE/MEM/WB). Check for pipeline stall condition. Assertions will be used to monitor active low pipeline stall signal.

Test	Description
Test_source_zero	Source1/source2 register address = 0
Test_source_non_zero_stall	Source1/source2 register address!=0, but source1 = EXE/MEM/WB destination address Source1/source2 register address!=0, but source2 = EXE/MEM/WB destination address
Test_source_non_zero_no_stall	Source1/source2 register address!=0, but source1 != EXE/MEM/WB destination address Source1/source2 register address!=0, but source2 != EXE/MEM/WB destination address

CPU level verification and Test plan:

For top level testing we will be writing code snippets in MIPS16 assembly, use the software model to get the instruction code. Load the Instruction memory and monitor the Registers.

The expected result (Register values) for each test will be logged in a file, and the results after each simulation will be compared against it

Test	Description
Test_reset	Assert reset, check registers
Test_ISA	<p>Test each instruction individually, checks expected result from the register file.</p> <p>R-type instruction: Test involving access to all 8 registers, for each instruction Test the access to R0 special register.</p> <p>I-type Instruction: Test LD/ST instruction Test for branch instructions, +/- displacement, rollover from last accessible address. Test for +/-0 immediate values</p>
Test_snippets	Write a MIPS16 assembly code snippets and test for expected behavior
Test_Hazard	Write a assemble code with RAW pipeline hazards and check for pipeline stalls.

Unit level assertions will be blinded for top CPU level verification.

Emulation:

We have to setup the design for emulation. For this setup, we should create a design directory on our compile host or file server. After adding HDL, HVL and source files, we should setup and map analysis libraries and create veloce.config file. Now, when the veloce.config file with all the compilation options is in the current design directory, we should run the velcomp from the top level of our design directory.

Packaging of RTL and Verify:

Replace the interconnections between the pipeline stages with interface/mod ports. Individually test each unit using the test bench created for the RTL released. The RTL from git will be used as a reference model for the testing. Stitch all the units together and run CPU level testing on the top module.

Find any chance of parameterizing/encapsulation in the RTL

Makefile, Environment Setup, project package:

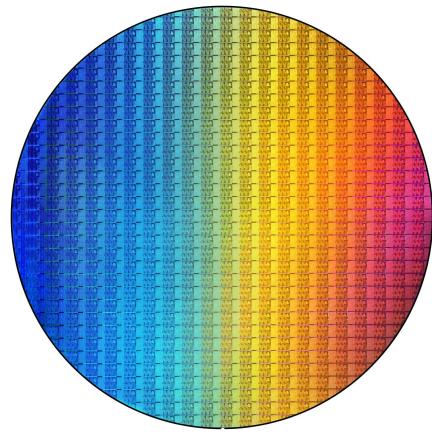
All the unit level test/CPU level testing will have its own Makefile, the make file is capable of reading the Assembly code, load the instruction memory and chose test to be run. Individual who owns the unit level test will create their own Makefile and environment setup.

Setup for Emulator is done by person who owns emulation part of the project.

Portland State University

ECE571 - SystemVerilog

Winter 2019



MIPS16 ISA Verification Plan

Aishwayra Doosa

Chenyang Li

R. Ignacio Genovese

Shouvik Rakshit

Introduction

As digital designs get bigger, the effort in verification can scale up to 70% or even 80% of the total design effort, thereby making it the most expensive step in the entire IC design flow. Besides, any behavioral or functional bugs escaping this phase will surface only after the silicon is integrated into the target system, resulting in even more costly design and silicon iterations. For this reason, nowadays it is essential for engineers to learn different approaches to overcome these bugs, getting to successfully implement verification environments aimed to reduce these costs.

Therefore, as final project for the course *ECE571 - Introduction to SystemVerilog for Design and Verification* at Portland State University, the team will develop and implement a Verification Plan for a MIPS16 ISA.

Verification Approach

Instead of using a set of functional specifications as base for our Verification Plan, we will learn the architecture and requirements from a completely functional MIPS 16 ISA design. This will be used to define a set of **unit tests** (by assembly code) that will serve as stimulus for the processor core. After running these tests, the QuestaSim **coverage tool** will be used to determine if this set of testcases is enough to get a good (branch, statement, fsm, toggle) coverage. Based on the results from the coverage tool, we will develop new testcases to get a better coverage in case it's necessary.

To assure the functional correctness of these testcases (besides taking it for granted as the design is supposed to be correct), the final state of the register file and memory will be checked, as each unit test should be simple enough to know this result.

Once we get a good set of tests, we will proceed to save into files the inputs and outputs of each of the 5 pipeline stages (Instruction Fetch, Instruction Decode, Execute, Memory and Writeback), in order to carry on with the verification of each of these independently. These files will then be used to stimulate each stage and compare its outputs.

Along with the unit tests, a set of **assertions** for each pipeline stage will be defined in a separate file and bound to the design.

Finally, this coverage and assertion based verification environment will be used to verify a “broken” design, in order to prove its effectiveness and implementation to find bugs.

Unit Tests

After reviewing and understanding the implemented MIPS16 ISA, we proceeded to consider each stage inputs to be stimulated and defined the following basic unit tests to implement:

- IF:
 - Implement branch taken and branch not taken.
 - Try the instruction memory boundaries using different offsets.
 - Produce stalls in the hazard detection unit.
- ID:
 - Implement all type of instructions using every register as destination, source operand 1 and source operand 2. There are 13 types of instructions:
 - OP_NOP
 - OP_ADD
 - OP_SUB
 - OP_AND
 - OP_OR
 - OP_XOR
 - OP_SL
 - OP_SR
 - OP_SRU
 - OP_ADD
 - OP_LD
 - OP_ST
 - OP_BZ
- EX:
 - Test all ALU operations using every register as destination, source operand 1 and source operand 2. There are 8 ALU operations:
 - ADD
 - SUB
 - AND
 - OR
 - XOR
 - SL (shift left)
 - SR (shift right, preserving sign bit)
 - SRU (shift right unsigned)
- MEM:

- Implement load and store operations to every memory position, using every register as source operand 1 (base), source operand 2 (offset) and destination register.
 - Consider that writes to register 0 (R0) will always produce a zero.
- WB:
 - Write back every register with results from the ALU and values read from the memory.
- Hazard detection unit:
 - Produce every possible stall using every register as destination, source operand 1 and source operand 2. There are 3 possible stalls:
 - When a source operand for the current instruction is the destination operand for the instruction in the EX stage.
 - When a source operand for the current instruction is the destination operand for the instruction in the MEM stage.
 - When a source operand for the current instruction is the destination operand for the instruction in the WB stage.

Verification Environment

For the first phase of the verification flow (top level verification) there will be a testcase folder containing each stage testcases. Some of these testcases will be shared between stages, as they are the same (for example, every instruction decoding will include ALU instructions, or stalls for the IF stage will be produced in the hazard detection unit testcases). A top level testbench will be used, which will have a string array containing every testcase file path and the expected result (register file and memory contents) for every unit test (used at the end of each test for proving functional correctness).

For the second phase (testing each pipeline stage separately), there will be a folder for each pipeline stage containing the input/output files for stimulating and comparing results. Each independent stage will have its own separate testbench.

For both of these stages there will be assertions that run along each testbench.

Also, a Makefile will be used for compiling and running each testbench.

Along with this, there will be a results folder containing the coverage, simulation and assertions results for each testbench.

Finally, for the third stage, this whole structure will be reproduced to test the “broken” design.

Team members responsibilities

For each stage, the designated member will create the assembly code, produce the prog file using the java assembler, create the expected register file and memory results and create and run each independent testbench.

- ID: Chenyang Li
- IF: Chenyang Li
- EX: Aishwarya Doosa
- MEM: Shouvik Rakshit
- WB: Shouvik Rakshit
- Hazard Detection Unit: R. Ignacio Genovese
- Top level integration, makefiles, coverage and assertions results: R. Ignacio Genovese
- Veloce (set environment, run examples, run MIPS16 ISA): Aishwarya Doosa/R. Ignacio Genovese



PicoBlaze™

KCPSTM3

8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-II PRO

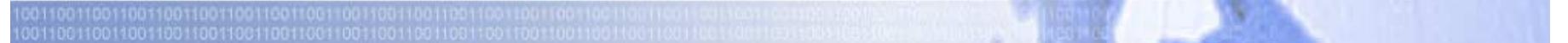
For Spartan-II(E) and Virtex(E) please use KCPSM
Virtex-II and Virtex-IIPro are also supported by KCPSM2

Ken Chapman

Xilinx Ltd

October 2003

Rev.7



Contents

Understanding KCPSM3

- 1 Title
- 2 Contents page
- 3 Limitations
- 4 What is KCPSM3?
- 5-6 KCPSM3 is small
- 7 Size and Performance
- 8 KCPSM3 Architecture
- 9-11 KCPSM3 Feature Set
- 12 Constant (k) Coded
- 13 Using KCPSM3 (VHDL)
- 14 Connecting the Program ROM
- 15 Verilog and System Generator

Instruction Set

- 16 KCPSM3 Instruction Set
- 17 JUMP
- 18 CALL
- 19 RETURN
- 20 RETURNI
- 21 ENABLE/DISABLE INTERRUPT
- 22 LOAD
- 23 AND
- 24 OR
- 25 XOR
- 26 TEST
- 27 ADD

- 28 ADDCY
- 29 SUB
- 30 SUBCY
- 31 COMPARE
- 32 SR0, SR1, SRX, SRA, RR
- 33 SL0, SL1, SLX, SLA, RL
- 34 OUTPUT
- 35 INPUT
- 36 STORE
- 37 FETCH

Interface Signals

- 38 READ and WRITE STOBES
- 39 RESET

KCPSM3 Assembler

- 40 KCPSM3 Assembler - Basic usage.
- 41 Assembler Errors
- 42 Assembler Files
- 43 ROM_form.vhd File
- 44 ROM_form.v File
- 45 ROM_form.coe File
- 46 <filename>.fmt File
- 47 <filename>.log file
- 48 constant.txt & labels.txt Files
- 49 pass.dat files
- 50-51 Program Syntax

- 52 CONSTANT Directive
- 53 NAMEREG Directive
- 54 ADDRESS Directive
- 55 KCPSM and KCPSM2 Compatibility
- 56 PicoBlaze Comparison

Interrupts and worked example

- 57 Interrupt Handling
- 58 Basics of Interrupt Handling
- 59 Example Design (VHDL)
- 60 Interrupt Service Routine
- 61 Interrupt Operation
- 62 Timing of Interrupt Pluses

Hints and Tips

- 63 CALL/RETURN Stack
- 64 Sharing program space
- 65-66 Design of Output Ports
- 67-68 Design of Input Ports
- 69 Connecting Memory
- 70 Simulation of KCPSM3
- 71-75 VHDL Simulation

Limitations

Limited Warranty and Disclaimer. These designs are provided to you “as is”. Xilinx and its licensors make and you receive no warranties or conditions, express, implied, statutory or otherwise, and Xilinx specifically disclaims any implied warranties of merchantability, non-infringement, or fitness for a particular purpose. Xilinx does not warrant that the functions contained in these designs will meet your requirements, or that the operation of these designs will be uninterrupted or error free, or that defects in the Designs will be corrected. Furthermore, Xilinx does not warrant or make any representations regarding use or the results of the use of the designs in terms of correctness, accuracy, reliability, or otherwise.

Limitation of Liability. In no event will Xilinx or its licensors be liable for any loss of data, lost profits, cost or procurement of substitute goods or services, or for any special, incidental, consequential, or indirect damages arising from the use or operation of the designs or accompanying documentation, however caused and on any theory of liability. This limitation will apply even if Xilinx has been advised of the possibility of such damage. This limitation shall apply notwithstanding the failure of the essential purpose of any limited remedies herein.

This module is not supported by general Xilinx Technical support as an official Xilinx Product.
Please refer any issues initially to the provider of the module.

Any problems or items felt of value in the continued improvement of KCPSM3 would be gratefully received by the author.

Ken Chapman
Senior Staff Engineer - Applications Specialist
email: chapman@xilinx.com

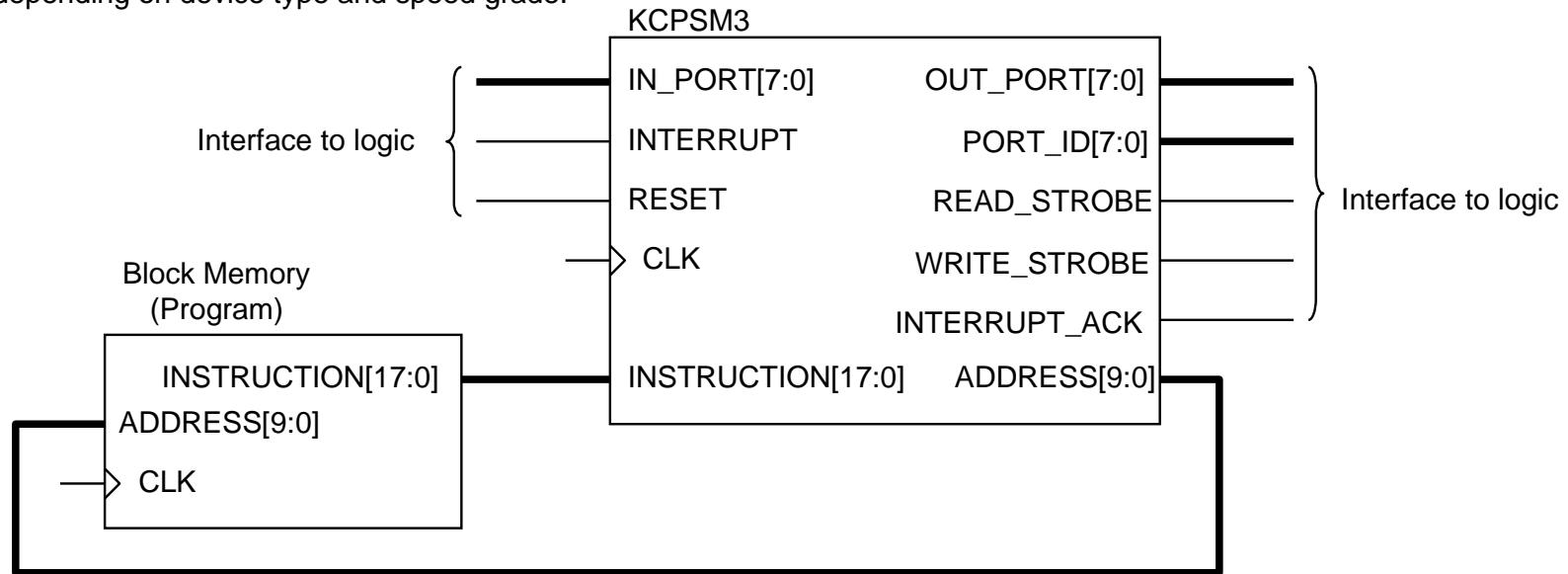
The author would also be pleased to hear from anyone using KCPSM or KCPSM2 with information about your application and how these macros have been useful.



What is KCPSM3 ?

KCPSM3 is a very simple 8-bit microcontroller primarily for the Spartan-3 devices but also suitable for use in Virtex-II and Virtex-IIPRO devices. Although it could be used for processing of data, it is most likely to be employed in applications requiring a complex, but non-time critical state machine. Hence it has the name of '(K)constant Coded Programmable State Machine'.

This revised version of popular KCPSM macro has still been developed with one dominant factor being held above all others - Size! The result is a microcontroller which occupies just **96 Spartan-3 Slices** which is just 5% of the XC3S200 device and less than 0.3% of the XC3S5000 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 1024 instructions. Even with such size constraints, the performance is respectable at approximately **43 to 66 MIPS** depending on device type and speed grade.

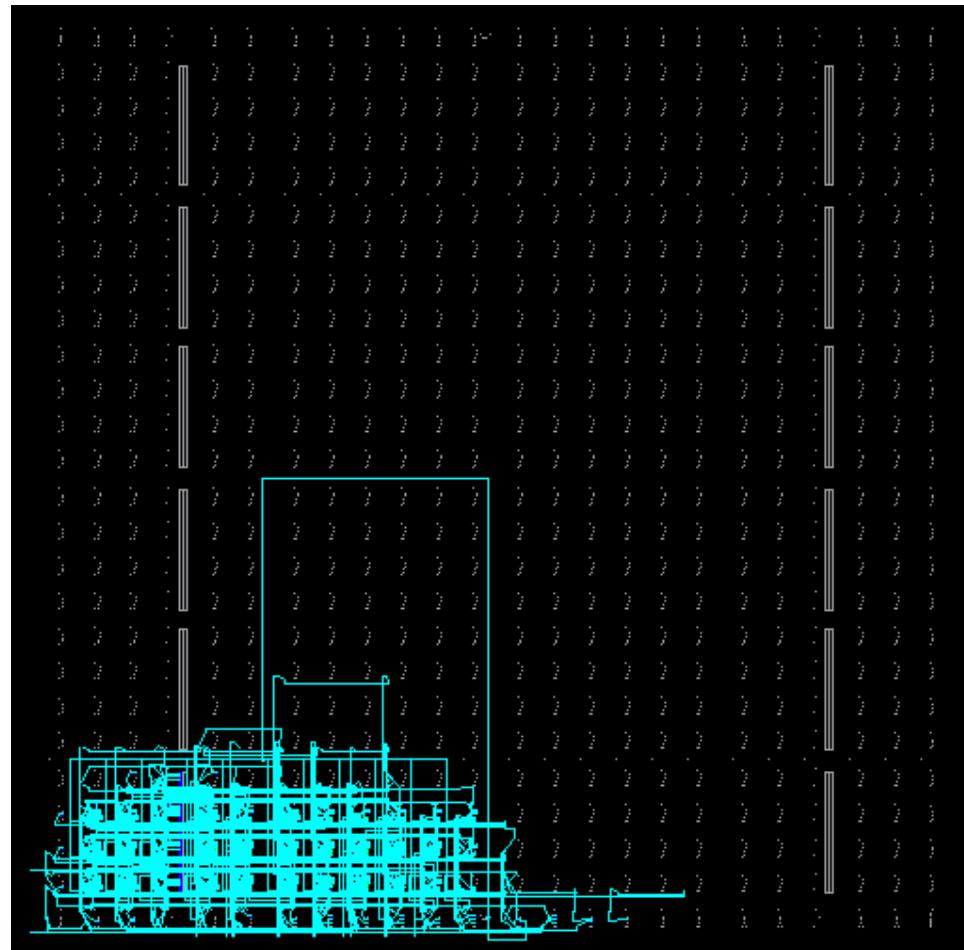


One of the most exciting features of the KCPSM3 is that it is totally embedded into the device and requires no external support. The very fact that ANY logic can be connected to the module inside the Spartan-3 or Virtex-II device means that any additional features can be added to provide ultimate flexibility. It is not so much what is inside the KCPSM3 module that makes it useful, but the environment in which it lives.



KCPSM3 is small!

KCPSM3 is supplied as VHDL and as a pre-compiled soft macro which is handled by the place and route tools to merge with the logic of a design. In large devices, the KCPSM3 is virtually free! The potential to place multiple KCPSM3 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 design.



This plot from the FPGA Editor viewer shows the macro in isolation within the XC3S200 Spartan-3 device.

96 Slices

**5% of XC3S200
Spartan-3 device**

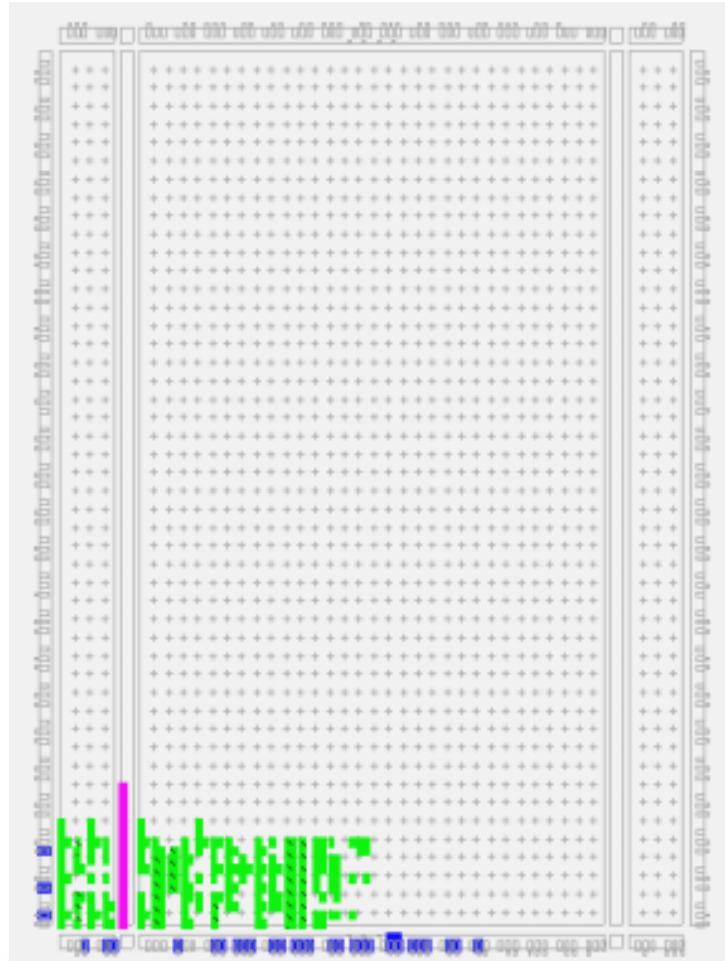
**~87MHz in -4
Speed Grade**

~43.5 MIPS



KCPSM3 is small!

This plot from the Xilinx Floorplanner shows the same implementation of KCPSM3 in an XC3S200 Spartan-3 device. This makes it easier to appreciate the actual logic resources required by the macro without the interconnect obscuring the detail.

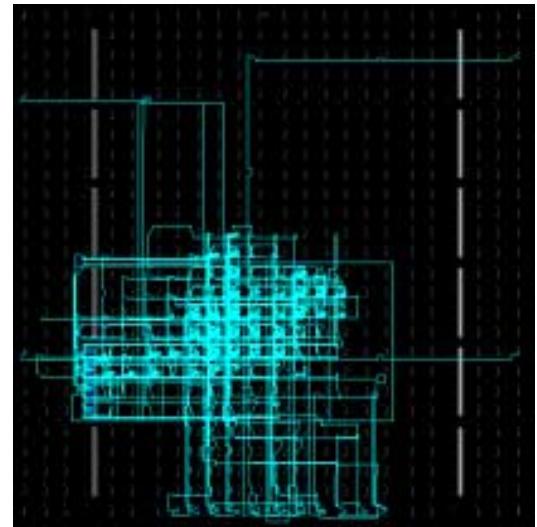


The placement in this Floorplanner view was achieved using a simple area constraint in the project UCF file.

```
INST processor_* LOC=SLICE_X0Y0:SLICE_X19Y4;
```

Such constraints are not required in normal designs and it has only been used in this case because so little of the device is occupied. Experiments have shown that placement constraints have very little effect on performance.

The FPGA Editor view shown to the right was the result when no constraints were used. The size is still 96 slices but this is now a little less obvious! The performance was actually a little higher than when using the area constraint indicating that a 'tidy' design is not always the fastest!



Size and Performance

The following device resource information is taken from the ISE reports for the KCPSM3 macro in an XC3S200 device. The reports reveal the features that are utilised and the efficiency of the macro. The 96 'slices' reported by the MAP process in this case may reduce to the minimum of 89 'slices' when greater packing is used to fit a complete design into a device.

XST Report

LUT1	:	2	}	109 LUTs (55 slices)
LUT2	:	6		
LUT3	:	68		
LUT4	:	33		
MUXCY	:	39	}	Carry and MUX logic (Free with LUTs)
MUXF5	:	9		
XORCY	:	35		
FD	:	24	}	76 Flip_flops (Free with LUTs)
FDE	:	2		
FDR	:	30		
FDRE	:	8		
FDRSE	:	10		
FDS	:	2		
RAM16X1D	:	8	— Register bank (8 slices)	
RAM32X1S	:	10	— Call/Return Stack (10 slices)	
RAM64X1S	:	8	— Scratch Pad Memory (16 slices)	
Total = 89 Slices				

MAP Report

Number of occupied Slices : 96 out of 1920 5%
Number of Block RAMs : 1 out of 12 8%
Total equivalent gate count for design: 74,814
12 × KCPSM3 can fit into the XC3S200 device (40% of the logic slices remaining). An equivalent gate count of 897,768 gates in a 200,000 gate device!

TRACE Report

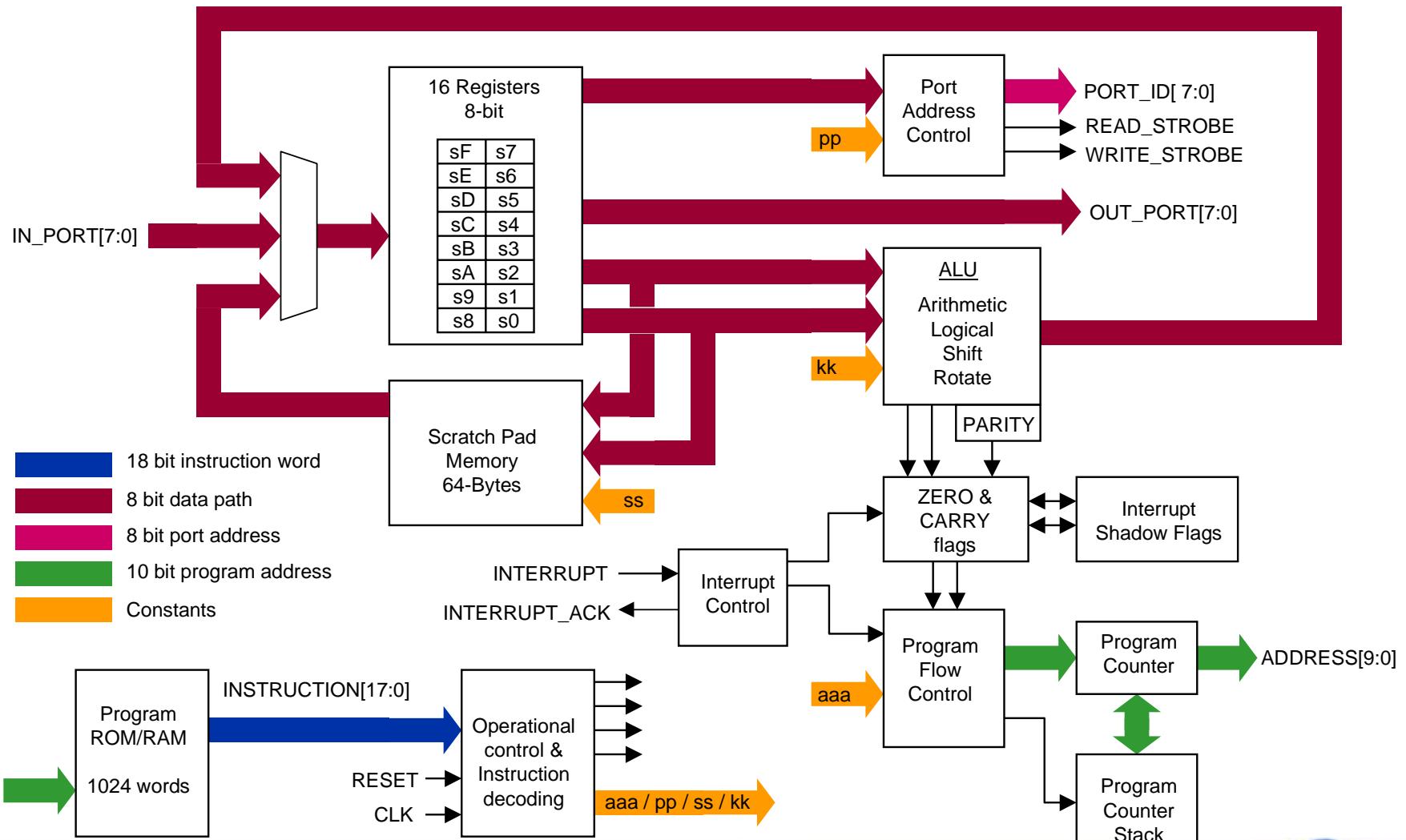
Device,speed: xc3s200,-4 (PREVIEW 1.22 2003-03-16)
Minimum period: 11.403ns
(Maximum frequency: 87.696MHz)

43.8 MIPS

TRACE Report for Virtex-II PRO

Device,speed: xcvp2,-7 (ADVANCED 1.76 2003-03-16)
Minimum period: 7.505ns
(Maximum frequency: 133.245MHz) 66.6 MIPS

KCPSM3 Architecture



KCPSM3 Feature Set

Features new to KCSPM3

KCPSM3 is a very simple processor architecture and anyone familiar with PSM, KCSPM or KCSPM2 will recognise that this is just the latest in a close family of 8-bit programmable state machines (see ‘PicoBlaze Comparison’). The motivation to develop this variant was the release of Spartan-3 devices and the highly constructive feedback from so many users of its predecessors.

Spartan-3 has adopted the 18Kbit Block RAM elements previously seen in the Virtex-II devices. This enables KCPSM3 to support programs up to 1024 locations which overcomes the most commonly encountered limit of KCSPM with Spartan-II(E).

At the risk of making KCPSM3 appear more complex than previous versions, some additional features have been included to address the most popular requests. COMPARE and TEST instructions enable register contents to be interrogated without changing their contents. The TEST instruction also calculates PARITY, useful for many communication applications. A 64-byte internal scratch pad memory allows many more variables to be held internally, more intuitive programs to be written and will typically eliminate requirement for memory attached to the I/O ports. Finally, an interrupt acknowledgement signal is provided.

The additional features make KCPSM3 26% larger than KCSPM and 14% larger than KCPSM2. However, It is expected that the additional features will enable more efficient programs to be written and for designs to require less peripheral logic.

Program Size

KCPSM3 supports a program up to a length of 1024 instructions utilising one block memory. Requirements for larger program space are typically addressed by using multiple KCPSM3 processors each with an associated block memory to distribute the various system tasks. Programs requiring significantly more memory are normally the domain of a full data processor such as MicroBlaze with its C-language programming support.

16 General Purpose Registers.

There are 16 general purpose registers of 8-bits specified as ‘s0’ through to ‘sF’ which may be renamed in the assembler code. All operations are completely flexible about the use of registers with no registers reserved for special tasks or having any priority over any other register. There is no accumulator as any register can be adopted for this task.



KCPSM3 Feature Set

ALU

The Arithmetic Logic Unit (ALU) provides many simple operations expected in an 8-bit processing unit.

All operations are performed using an operand provided from any register (sX). The result is returned to the same register.

For operations requiring a second operand, a second register can be specified (sY) or a constant 8-bit value (kk) can be supplied. The ability to specify any constant value with no additional penalty to program size or performance enhances the simple instruction set i.e. the ability to 'ADD 1' is the same as a dedicated INCREMENT operation.

Addition (ADD) and Subtraction (SUB) have the option to include the carry flag as an input (ADDCY and SUBCY) for the support of arithmetic operations requiring more than 8-bits.

LOAD, AND, OR and XOR bit-wise operators provide ability to manipulate and test values.

Comprehensive SHIFT and ROTATE group.

COMPARE and TEST instructions enable register contents to be tested without altering their contents and determine PARITY.

Flags and Program Flow Control

The results of ALU operations determine the status of the ZERO and CARRY flags. The ZERO flag is set whenever the ALU result has all bits reset (00_{16}). The CARRY flag is set when there is an overflow from an arithmetic operation. It is also used to capture the bit moved out of a register during shift and rotate instructions. During a TEST instruction, the carry flag is used to indicate if the 8-bit temporary result has ODD PARITY.

This status of the flags can be used to determine the execution sequence of the program using conditional and non-conditional program flow control instructions. JUMP commands are used to specify absolute addresses (aaa) within the program space. CALL and RETURN commands provide sub-routine facilities for commonly used sections of code. A CALL is made to an absolute address (aaa) and an internal program counter stack preserves the associated address required by the RETURN instruction. The stack supports up to 31 nested subroutine levels.

Reset

The RESET input forces the processor back into the initial state. The program will execute from address '000' and interrupts will be disabled. The status flags and CALL/RETURN stack will also be reset. Note that register contents are not affected.



KCPSM3 Feature Set

Input/Output

KCPSM3 effectively has 256 input ports and 256 output ports. The port being accessed is indicated by an 8-bit address value provided on the 'PORT_ID'. The port address can be specified in the program as an absolute value (pp), or may be indirectly specified as the contents of any of the 16 registers ((sY)).

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 pulse being output on the READ_STROBE. It is not always necessary to use this signal in the input interface logic, but it can be useful to indicate that data has been acquired by the processor. During an 'OUTPUT', the contents of any of the 16 registers are transferred to the output port. An output operation is indicated by a pulse being output on the WRITE_STROBE. This strobe signal will be used by the interface logic to ensure that only valid data is passed to external systems. Typically, WRITE_STROBE will be used as a clock enable or write enable (see 'READ and WRITE STROBES').

Scratch Pad Memory

This is an internal 64 byte general purpose memory. The contents of any of the 16 registers can be written to any of the 64 locations using a STORE instruction. The complementary FETCH instruction allows the contents of any of the 64 memory locations to be written to any of the 16 registers. This allows a much greater number of variables to be held within the boundary of the processor and tends to reserve all of the I/O space for real inputs and output signals.

The 6-bit address to specify a scratch pad memory location can be specified in the program as an absolute value (ss), or may be indirectly specified as the contents of any of the 16 registers (sY). Only the lower 6-bits of the register are used, so care must be taken not to exceed the 00 - 3F₁₆ range of the available memory.

Interrupt

The processor provides a single INTERRUPT input signal. Simple logic can be used to combine multiple signals if required. Interrupts are disabled (masked) by default, and are then enabled and disabled under program control. An active interrupt forces KCPSM3 to initiate a 'CALL 3FF' (a subroutine call to the last program memory location) from where the user can define a suitable jump vector to an Interrupt Service Routine (ISR). At this time, a pulse is generated on the INTERRUPT_ACK output, the ZERO and CARRY flags are automatically preserved and any further interrupts are disabled. The 'RETURNI' instruction ensures that the end of an ISR restores the status of the flags and specifies if future interrupts will be enabled or disabled.



Constant(k) Coded

The KCPSM3 is in many ways a machine based on Constants.....

Constant Values

Constant values may be specified for use in most 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 KCPSM3.
- Constant address values for controlling the execution sequence of the program.
- Constant address values for accessing internal scratch pad memory.

The KCPSM3 instruction set coding has been 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 will execute over 2 clock cycles.

Such constant execution rate is of great value when determining the execution time of a program particularly when embedded into a real time situation.

Constant Program Length

The program length is 1024 instructions and therefore conforms to the 1024x18 format of a single Spartan-3, Virtex-II or Virtex-II PRO Block RAM. This means that all address values are specified as 10-bits contained within the instruction coding (the assembler supports line labels to simplify the writing of programs). The fixed memory size promotes a consistent level of performance from the module. See also ‘Sharing Program Space’.



Using KCPSM3 (VHDL)

The principle method by which KCPSM3 will be used is in a VHDL design flow. The KCPSM3 macro is provided as source VHDL (kcpsm3.vhd) which has been written to provide an optimum and predictable implementation in a Spartan-3 or Virtex-II(PRO) device. The code is suitable for implementation and simulation of the macro. It has been developed and tested using XST for implementation and ModelSim for simulation. The code should not be modified in any way.

VHDL Component declaration of KCPSM3

```
component kcpsm3
    Port (
        address : out std_logic_vector(9 downto 0);
        instruction : in std_logic_vector(17 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;
        interrupt_ack : out std_logic;
        reset : in std_logic;
        clk : in std_logic);
    end component;
```

VHDL Component instantiation of KCPSM3

```
processor: kcpsm3
    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,
        interrupt_ack => interrupt_ack_signal,
        reset => reset_signal,
        clk => clk_signal);
```

Connecting the Program ROM

The principle method by which KCPSM3 program ROM will be used is in a VHDL design flow. The KCPSM3 assembler will generate a VHDL file in which a block RAM and its initial contents are defined (see assembler notes for more detail). This VHDL can be used for implementation and simulation of the processor. It has been developed and tested using XST for implementation and ModelSim for simulation.

VHDL Component declaration of program ROM

```
component prog_rom
    Port (      address : in std_logic_vector(9 downto 0);
                instruction : out std_logic_vector(17 downto 0);
                           clk : in std_logic);
end component;
```

VHDL Component instantiation of program ROM

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

Note - The name of the program ROM (shown as 'prog_rom' in the above examples) will depend on the name of your program. For example, if your program file was called 'phone.psm', then the assembler will generate a program ROM definition file called 'phone.vhd'.

To aid with development, a VHDL file called 'embedded_kcpsm3.vhd' is also supplied in which the KCPSM3 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 your own code.

Note: It is recommended that 'embedded_kcpsm3.vhd' is used for the generation of an ECS schematic symbol.

Verilog and System Generator

Although the primary design flow is VHDL, KCPSM3 can be used in any design flow supported by Xilinx. The assembler also generates program memory definition files suitable for Verilog and the Simulink based System Generator design flows.

<filename>.v - The assembler generates a Verilog file in which a block RAM and its initial contents are defined (see assembler notes for more detail). This Verilog can be used for implementation and simulation of the processor. The kcspm3.ngc file will be used to define the processor.

kcspm3.ngc - The NGC file provided was generated by synthesising the kcspm3.vhd file with XST (without inserting I/O buffers). This file can be used as a 'black box' in a Spartan-3, Virtex-II or Virtex-IIIPRO design, and it will be merged with the rest of your design during the translate phase (ngdbuild). Note that busses are defined in the style 'IN_PORT<7:0>' with individual signals 'in_port_0' through to 'in_port_7'.

<filename>.m - The assembler generates a m-function used to define the contents of a System Generator memory block within the MATLAB Simulink environment. (see System Generator documentation for more information on this design flow).

<filename>.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 your chosen design flow and connected to the kcspm3 'black box' in your design (see assembler notes for more details).

Simulation Models

If the NGC file is used in the design flow, then some form of back annotated description will be required for simulation of your design in order to fill in the 'black box' details. The following command can be used to generate a Verilog simulation model (see the Xilinx online manuals for more details - Synthesis and Simulation Design Guide - section 6).

```
ngd2ver kcspm3.ngd sim_model_kcspm3.v
```



KCPSM3 Instruction Set

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

'kk' represents a constant value in the range 00 to FF.

'aaa' represents an address in the range 000 to 3FF.

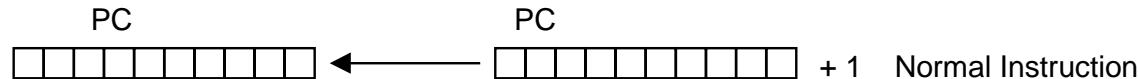
'pp' represents a port address in the range 00 to FF.

'ss' represents an internal storage address in the range 00 to 3F.

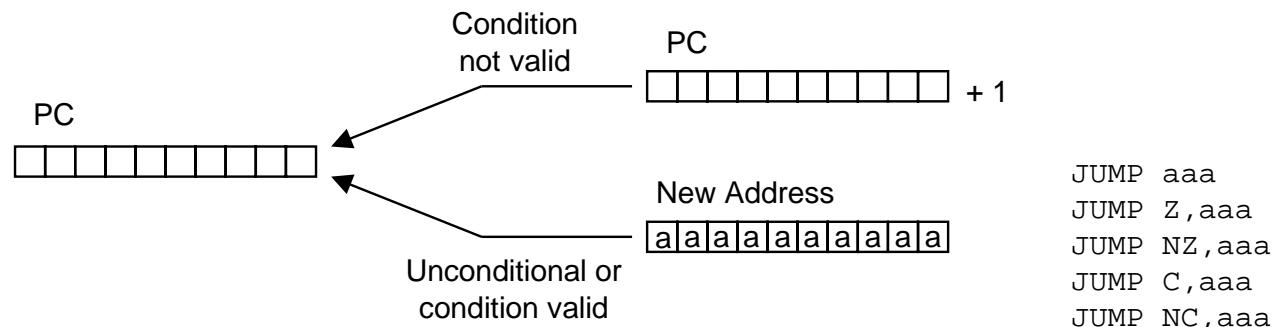


JUMP

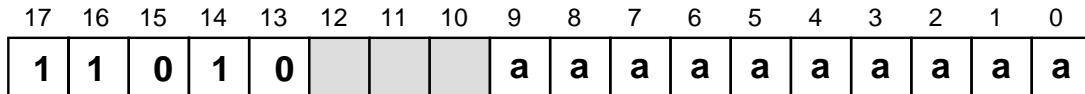
Under normal conditions, the program counter (PC) increments to point to the next instruction. The address space is fixed to 1024 locations (000 to 3FF hex) and therefore the program counter is 10 bits wide. It is worth noting that the top of memory is 3FF hex and will increment to 000.



The JUMP instruction may be used to modify this sequence by specifying a new address. However, the JUMP instruction may be conditional. A conditional JUMP will only be 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 10-bit address as a 3 digit hexadecimal value. The assembler supports labels to simplify this process.

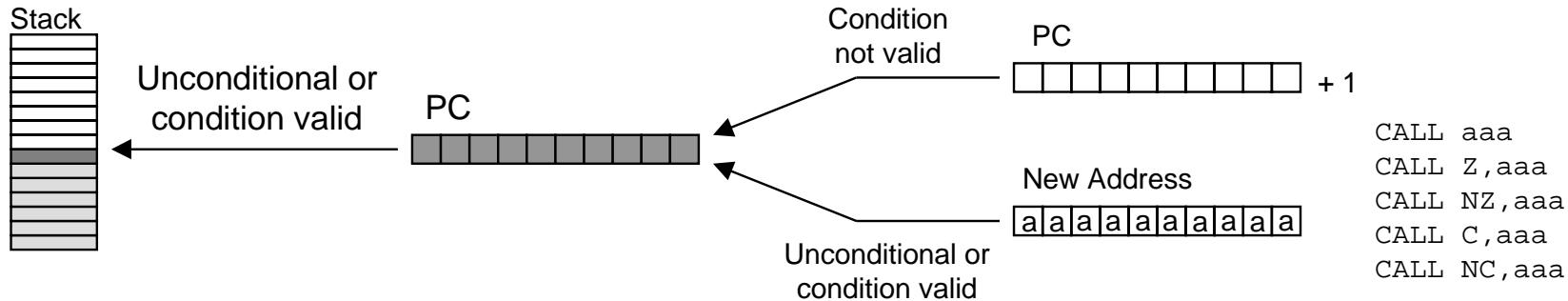


Bit 12 0 - UNCONDITIONAL
 1 - CONDITIONAL

<u>Bit 11</u>	<u>Bit 10</u>	<u>Condition</u>
0	0	if Zero
0	1	if NOT Zero
1	0	if Carry
1	1	if NOT Carry

CALL

The CALL instruction is similar in operation to the JUMP instruction in that it will modify the normal program execution sequence by specifying a new address. The CALL instruction may 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.



The program counter stack supports a depth of 31 address values. This enables nested 'CALL' sequences to a depth of 31 levels to be performed. However, the stack will also be used during an interrupt operation and hence 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 becomes full, it simply 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 other memory needs to be reserved or provided for the stack.

Each CALL instruction must specify the 10-bit address as a 3 digit hexadecimal value. The assembler supports labels to simplify this process.

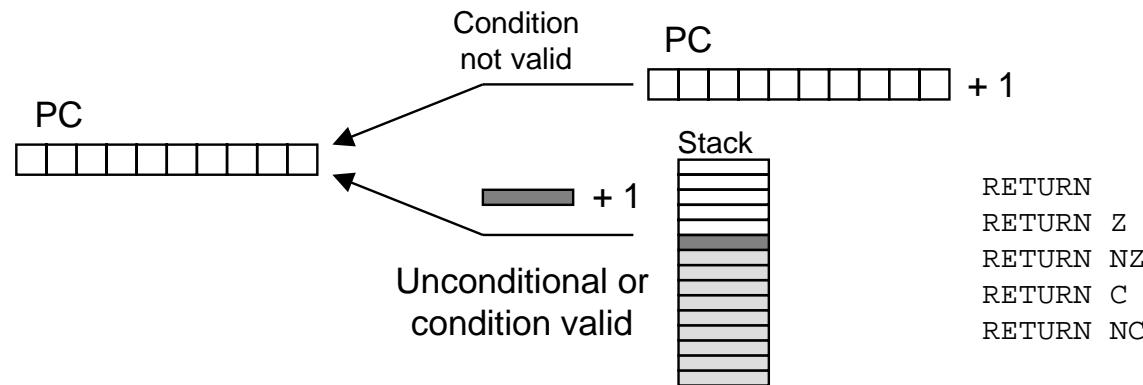
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0				a	a	a	a	a	a	a	a	a	a

Bit 12 0 - UNCONDITIONAL
 1 - CONDITIONAL

Bit 11	Bit 10	Condition
0	0	if Zero
0	1	if NOT Zero
1	0	if Carry
1	1	if NOT Carry

RETURN

The RETURN instruction is the complement to the CALL instruction. The RETURN instruction may also be conditional. In this case the new program counter (PC) value will be formed internally by incrementing the last value on the program address stack. This ensures that the program will execute the instruction following the CALL instruction which resulted in the subroutine. The RETURN instruction has no effect on the status of the flags.

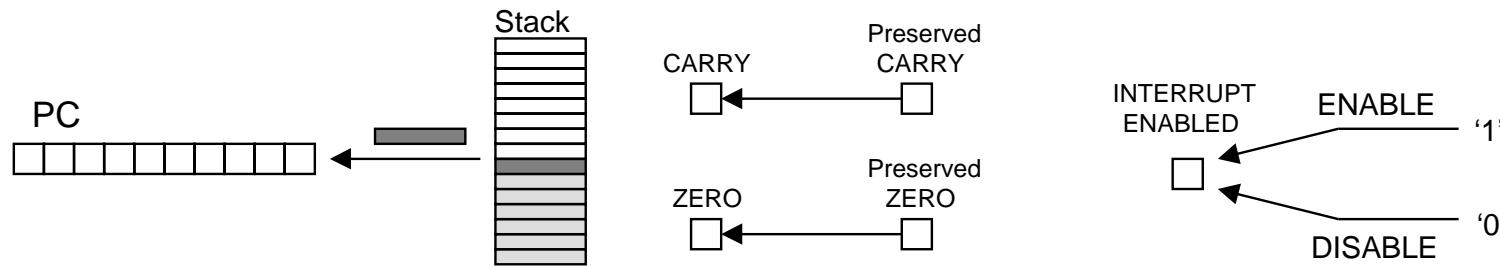


It is the responsibility of the programmer to ensure that a RETURN is only performed in response to a previous CALL instruction such that the program counter stack contains a valid address. The cyclic implementation of the stack will continue to provide values for RETURN instructions which can not be defined.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit 11	Bit 10	Condition
1	0	1	0	1				0	0	0	0	0	0	0	0	0	0	0	0	if Zero if NOT Zero if Carry if NOT Carry

RETURNI

The RETURNI instruction is a special variation of the RETURN instruction which should be used to conclude an interrupt service routine. The RETURNI is unconditional and therefore will always load the program counter (PC) with the last address on the program counter stack (the address is not incremented in this case since the instruction at the address stored will need to be executed). The RETURNI instruction restores the flags to the condition they were in at the point of interrupt. The RETURNI also determines the future ability of interrupts using ENABLE and DISABLE as an operand.

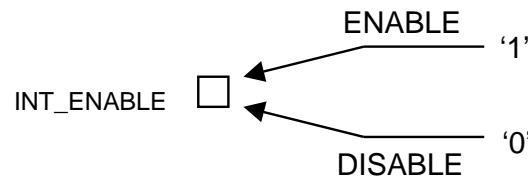


It is the responsibility of the programmer to ensure that a RETURNI is only performed in response to an interrupt. Each RETURNI must specify if further interrupt is to be enabled or disabled.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RETURNI ENABLE	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
RETURNI DISABLE	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ENABLE/DISABLE INTERRUPT

These instructions are used to set and reset the INT_ENABLE flag. Before using ENABLE INTERRUPT a suitable interrupt routine must be associated with the interrupt address vector (located at address 3FF). Interrupts should never be enabled whilst performing an interrupt service routine.

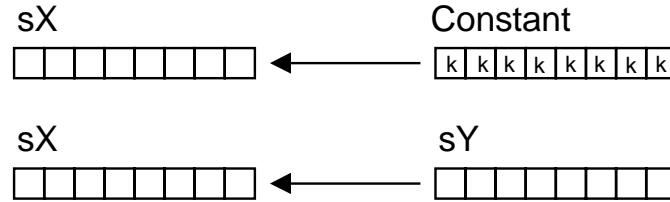


Interrupts are masked when the INT_ENABLE flag is low. This is the default state of the flag following device configuration or a KCPSM3 reset. The INT_ENABLE is also reset during an active interrupt.

ENABLE INTERRUPT	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	
DISABLE INTERRUPT	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	

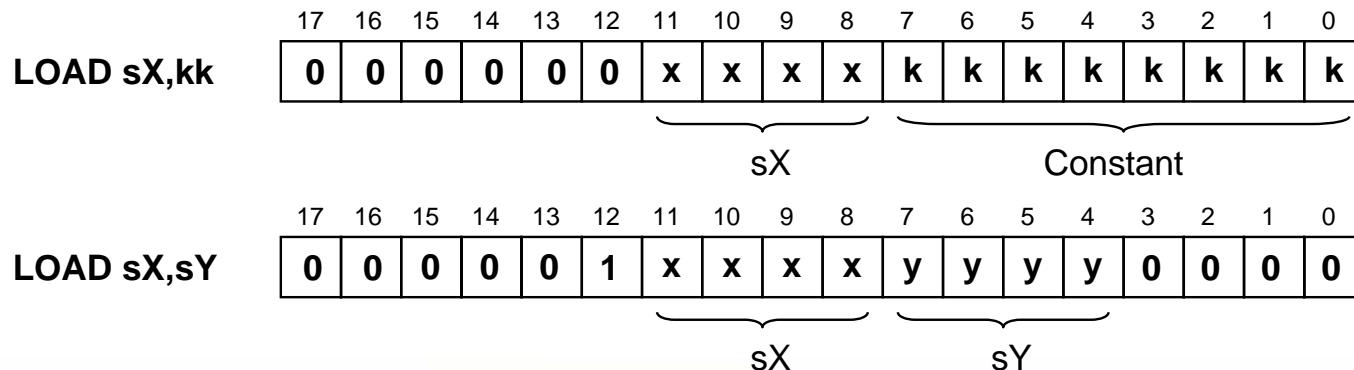
LOAD

The LOAD instruction provides a method for specifying the contents of any register. The new value can be a constant, or the contents of any other register. The LOAD instruction has no effect on the status of the flags.



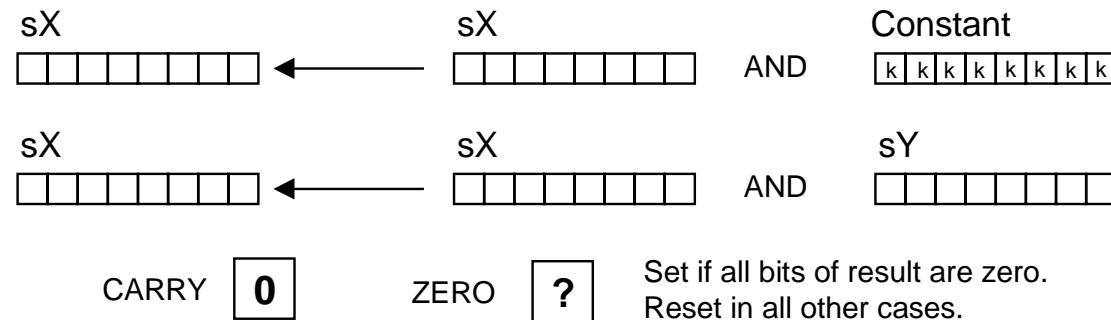
Since the LOAD instruction does not effect the flags it may be used to reorder and assign register contents at any stage of the program execution. The ability to assign a constant with no impact to the program size or performance means that the load instruction is the most obvious way to assign a value or clear a register.

The first operand of a LOAD instruction must specify the register to be loaded as register 's' followed by a hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

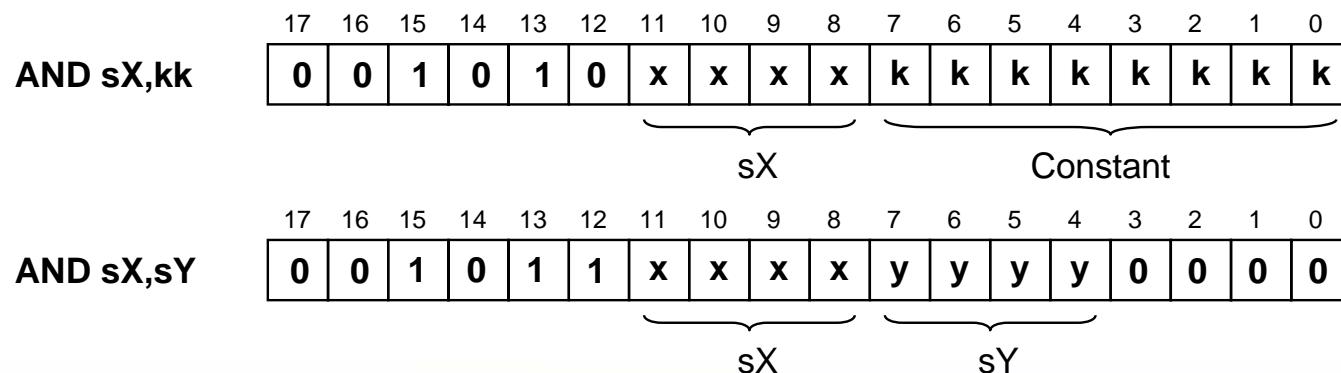


AND

The AND instruction performs a bit-wise logical ‘AND’ operation between two operands. For example 00001111 AND 00110011 will produce the result 00000011. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The AND operation is useful for resetting bits of a register and performing tests on the contents (see also TEST instruction). The status of the ZERO flag will then control the flow of the program.

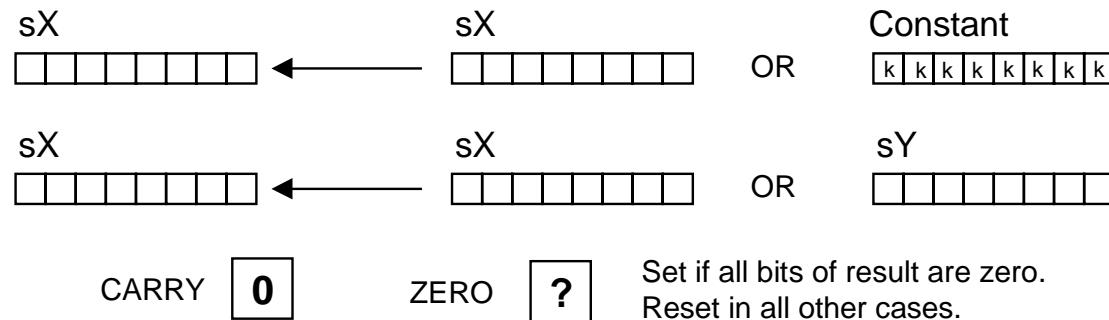


Each AND instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

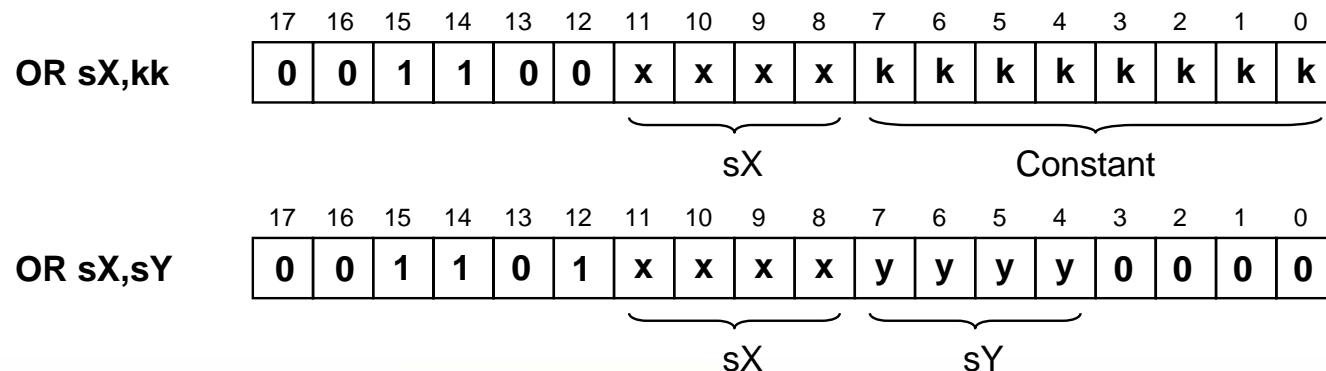


OR

The OR instruction performs a bit-wise logical ‘OR’ operation between two operands. For example 00001111 OR 00110011 will produce the result 00111111. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. OR provides a way to force any bits of the specified register to be set which can be useful in forming control signals.

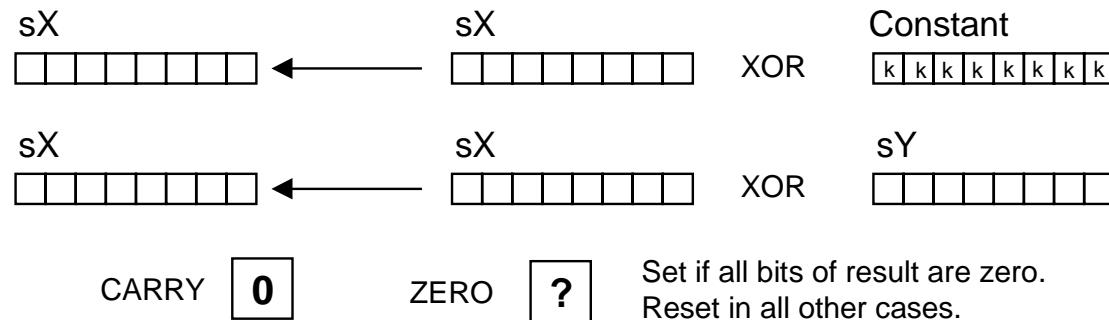


Each OR instruction must specify the first operand register as ‘s’ followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

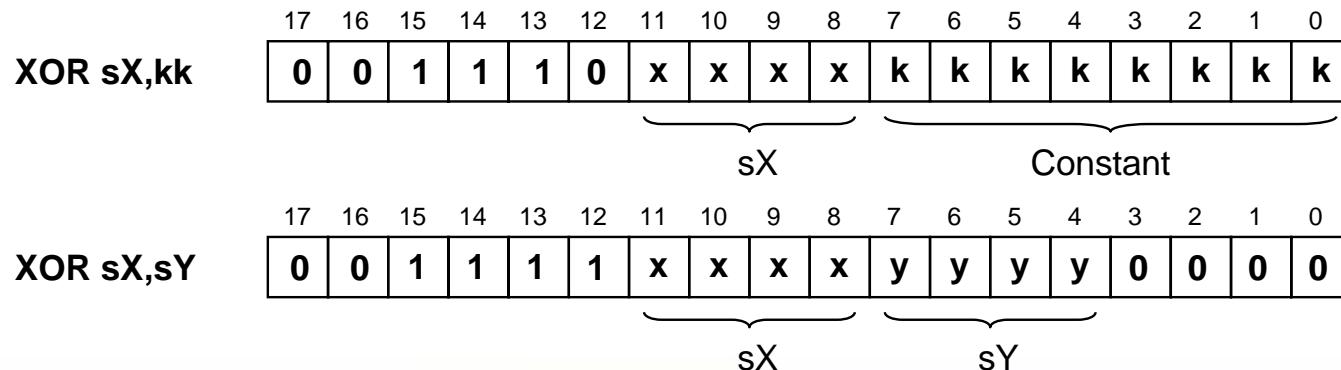


XOR

The XOR instruction performs a bit-wise logical ‘XOR’ operation between two operands. For example 00001111 XOR 00110011 will produce the result 00111100. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The XOR operation is useful for inverting bits contained in a register which is useful in forming control signals.

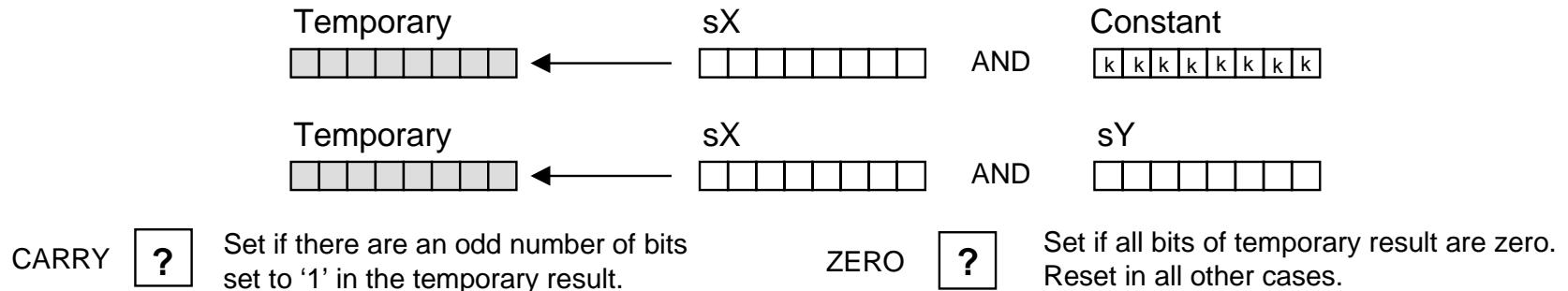


Each XOR instruction must specify the first operand register as ‘s’ followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

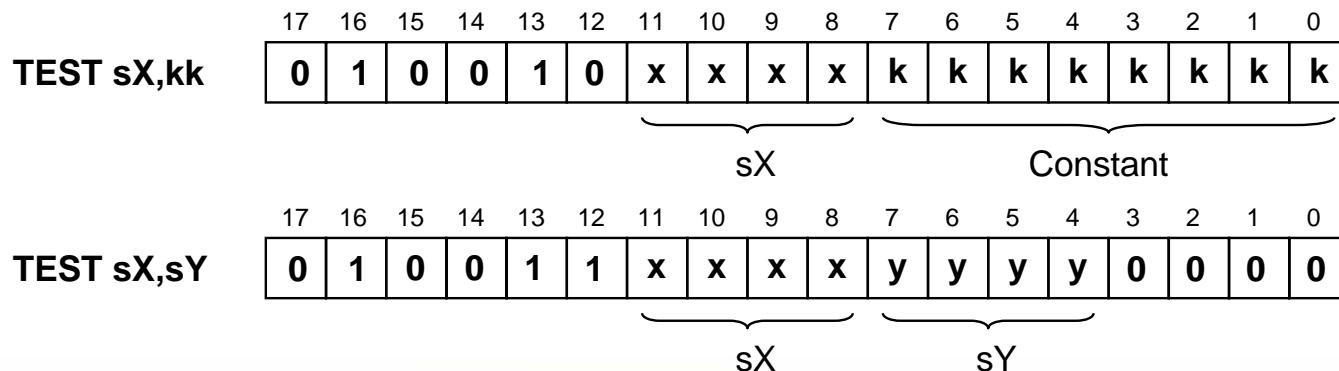


TEST

The TEST instruction performs a bit-wise logical ‘AND’ operation between two operands. Unlike the ‘AND’ instruction, the result of the operation is discarded and only the flags are affected. The ZERO flag is set if all bits of the temporary result are low. The CARRY flag is used to indicate the **ODD PARITY** of the temporary result. Parity checks typically involve a test of all bits, i.e. if the contents of ‘s5’ = 3D (00111101), the execution of TEST s5,FF will set the CARRY flag indicating ODD parity. Bit testing is typically used to isolate a single bit. For example TEST s5,04 will test bit2 of the ‘s5’ register which would set the CARRY flag if the bit is high (reset if the bit is low) and set the ZERO flag if the bit is low (reset if the bit is high).

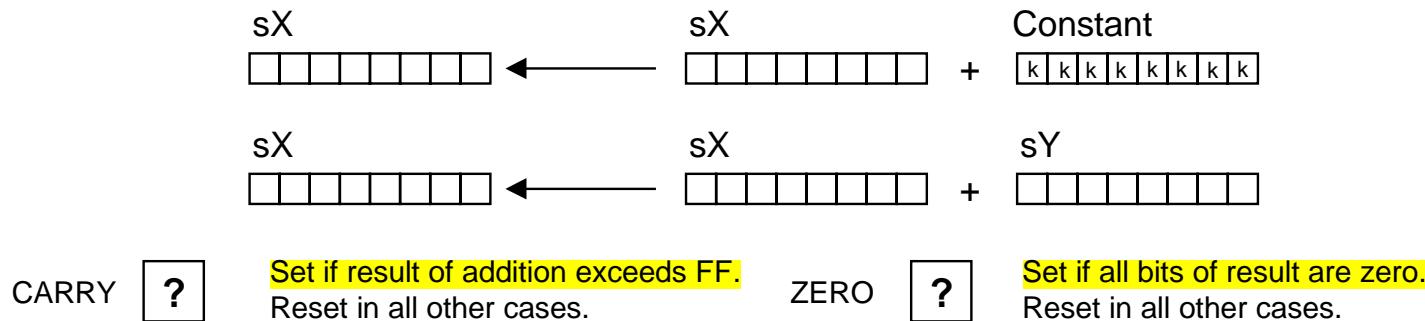


Each TEST instruction must specify the first operand register as ‘s’ followed by a hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

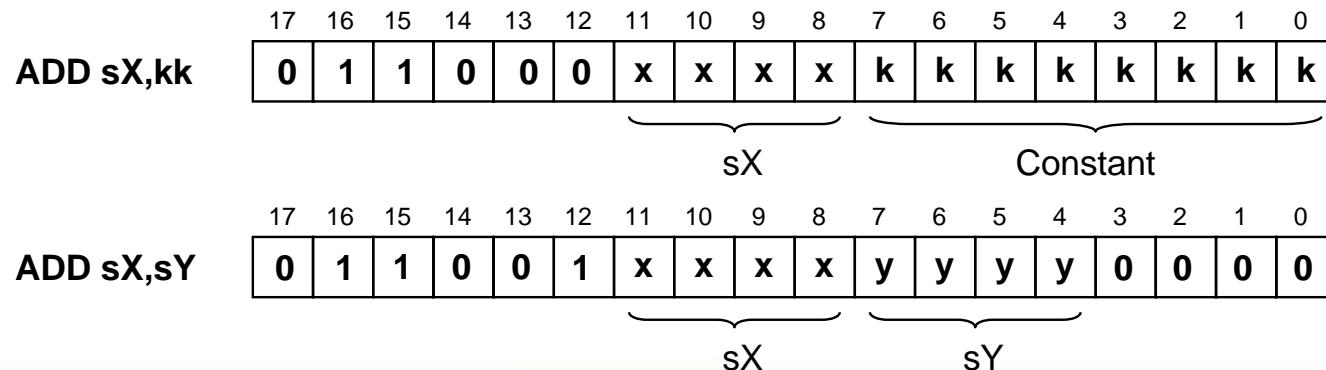


ADD

The ADD instruction performs an 8-bit addition of two operands. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected 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 and counters.

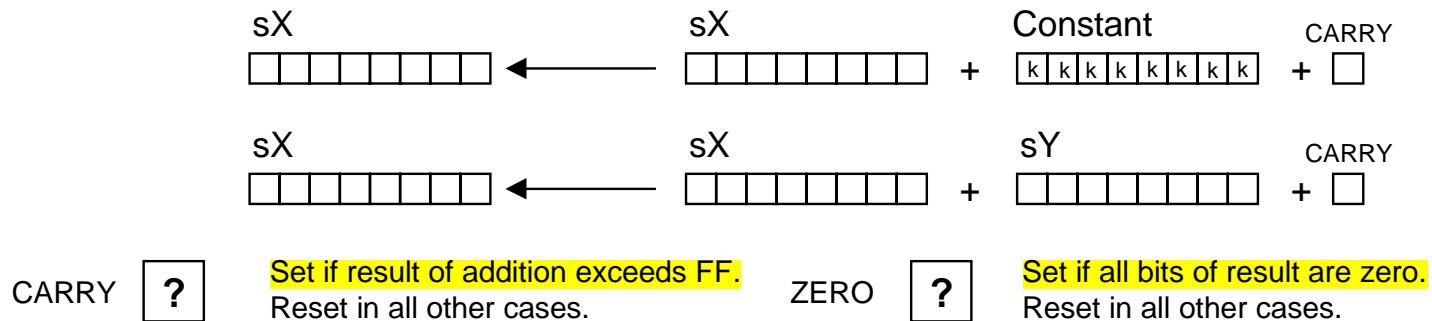


Each ADD instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

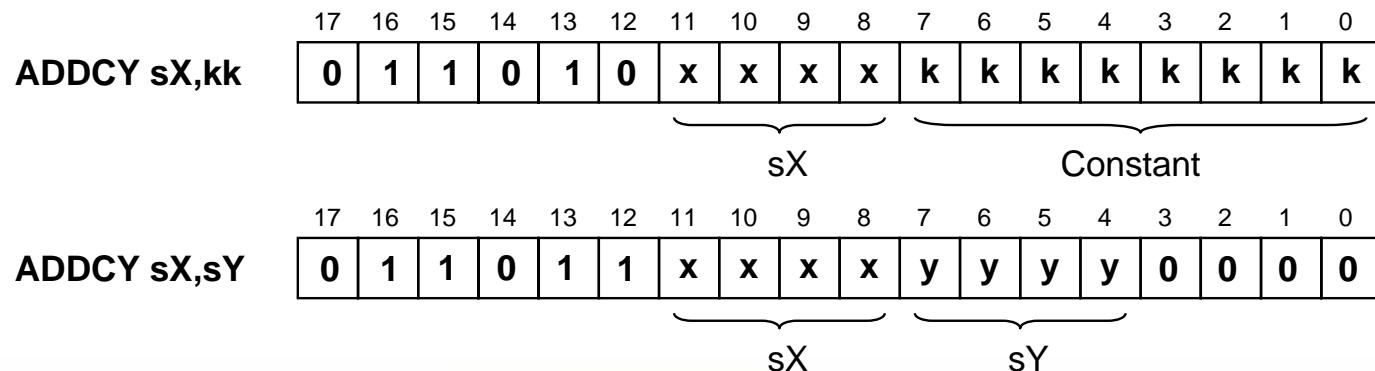


ADDCY

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 it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The ADDCY operation can be used in the formation of adder and counter processes exceeding 8 bits.

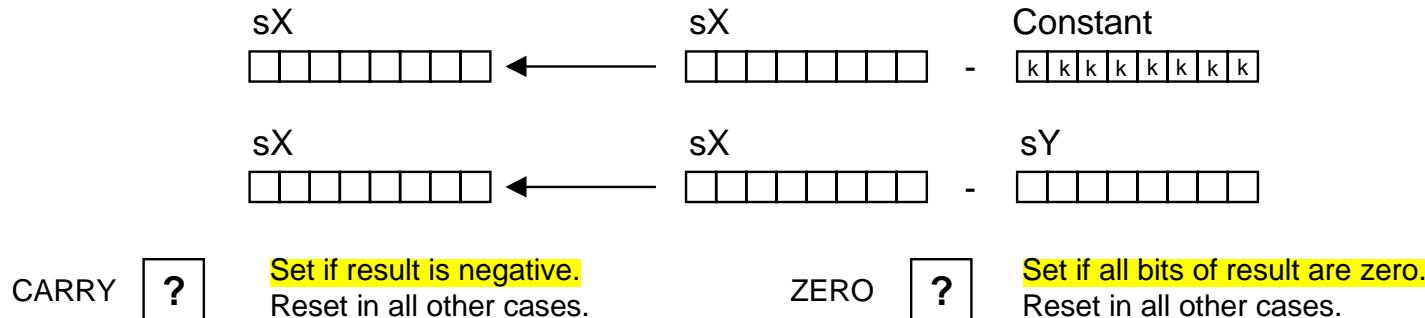


Each ADDCY instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

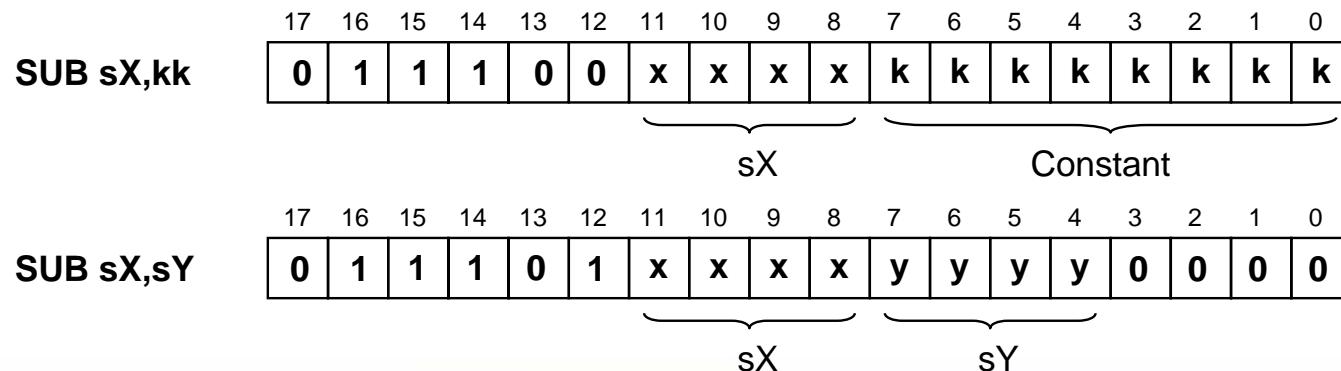


SUB

The SUB instruction performs an 8-bit subtraction of two operands. The first operand is any register, and it is this register which will be assigned the result of the operation. The second operand may also be any register or an 8-bit constant value. Flags will be effected 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 's05' contains 27 hex and the instruction SUB s05,35 is performed, then the stored result will be F2 hex and the CARRY flag will be set.

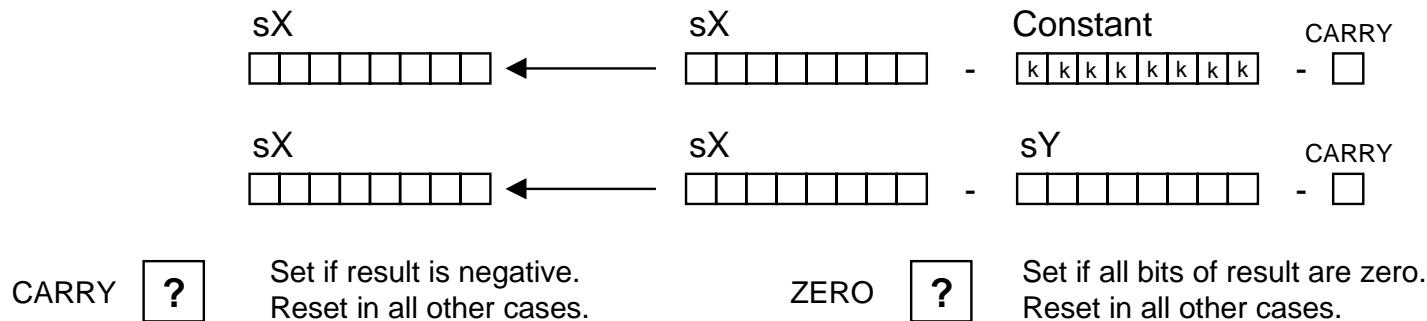


Each SUB instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

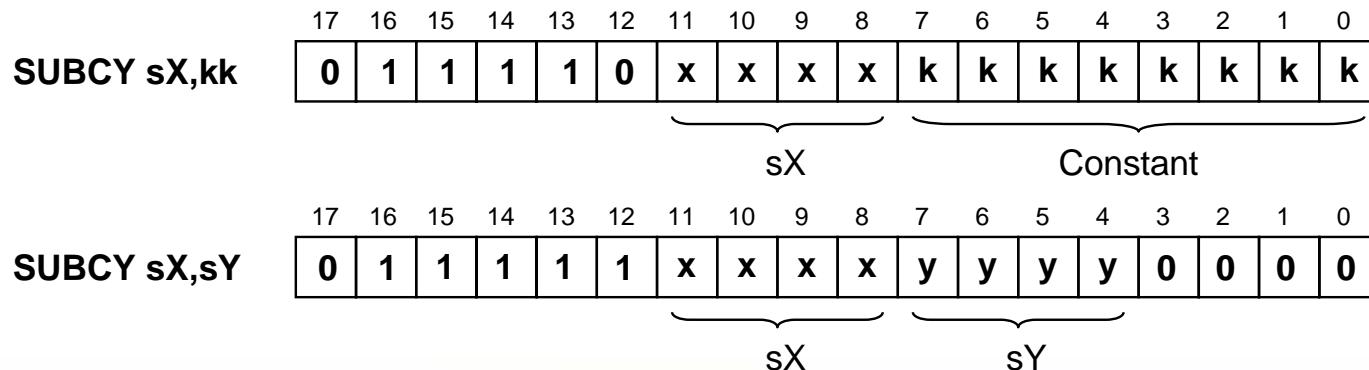


SUBCY

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 it is this register which will be assigned the result of the operation. The second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The SUBCY operation can be used in the formation of subtract and down counter processes exceeding 8 bits.

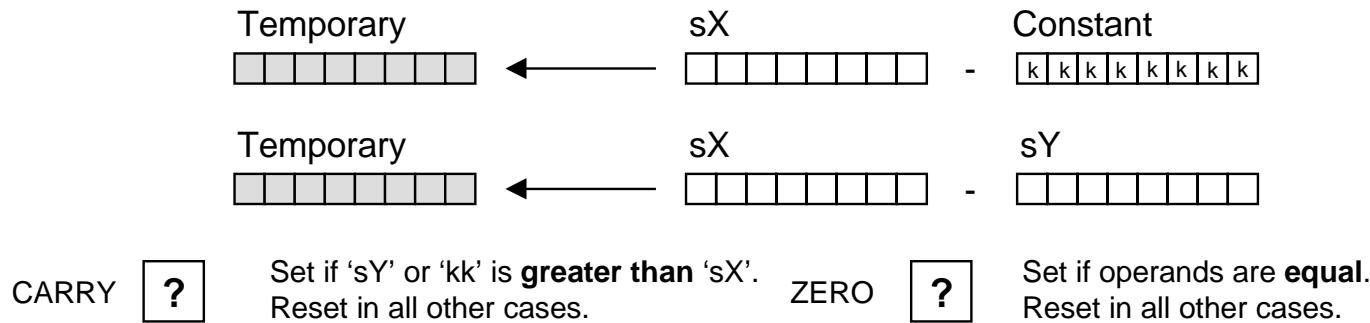


Each SUBCY instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

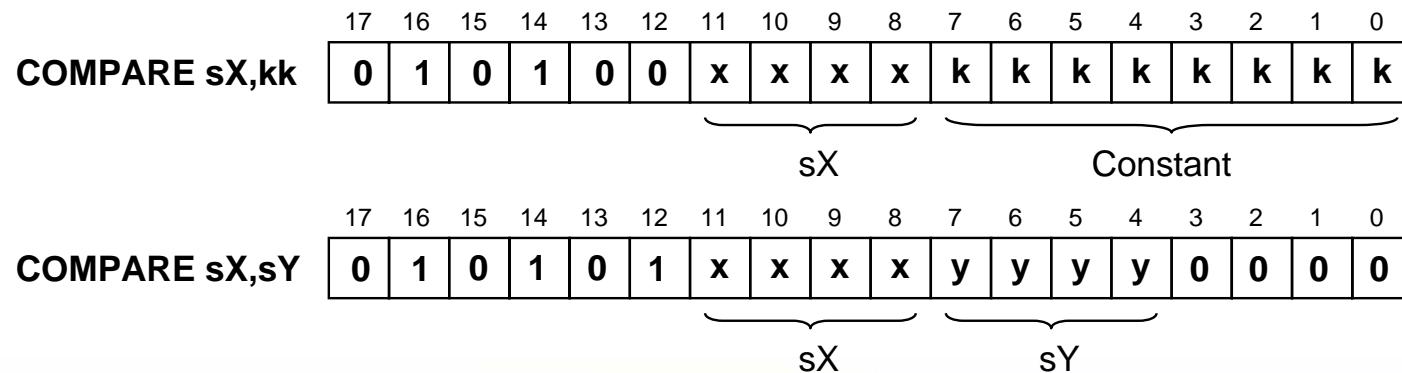


COMPARE

The COMPARE instruction performs an 8-bit subtraction of two operands. Unlike the 'SUB' instruction, the result of the operation is discarded and only the flags are affected. The ZERO flag is set when all the bits of the temporary result are low and indicates that both input operands were identical. The CARRY flag indicates when an underflow has occurred and indicates that the second operand was larger than the first. For example, if 's05' contains 27 hex and the instruction COMPARE s05,35 is performed, then the CARRY flag will be set ($35 > 27$) and the ZERO flag will be reset ($35 \neq 27$).

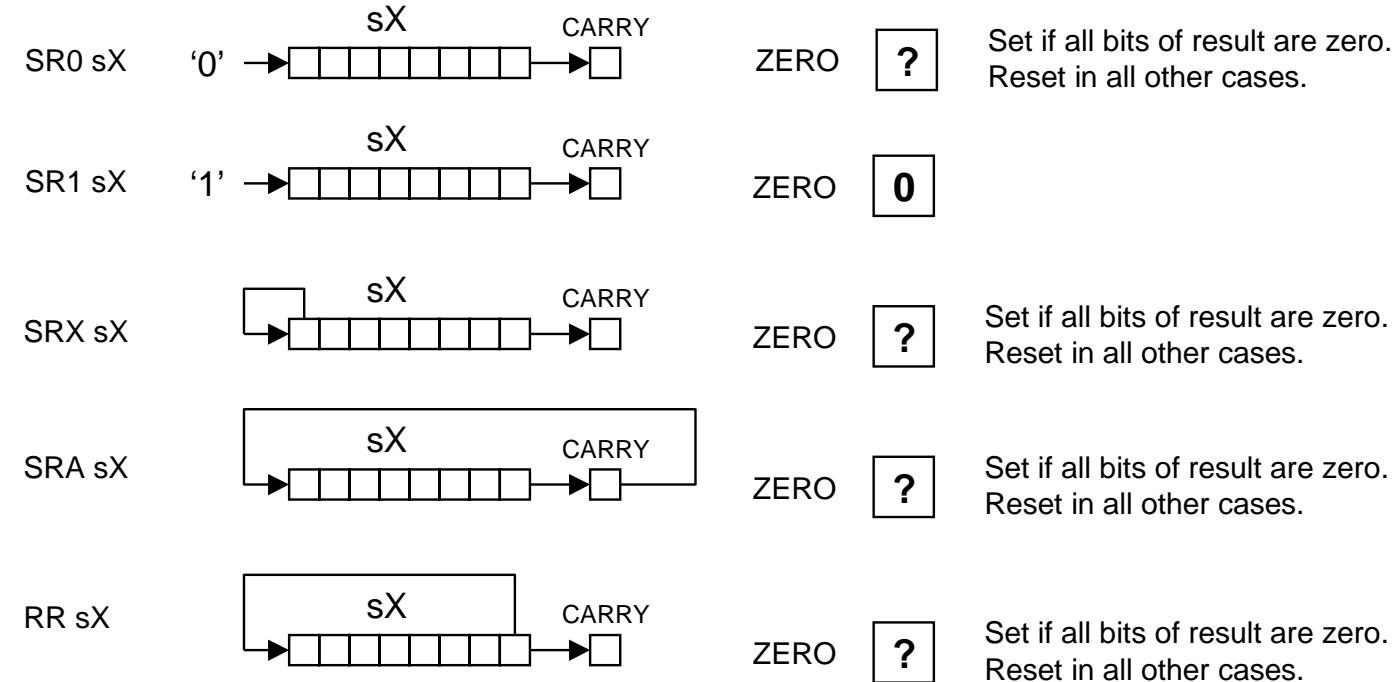


Each COMPARE instruction must specify the first operand register as 's' followed by a hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.



SR0, SR1, SRX, SRA, RR

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



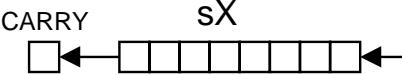
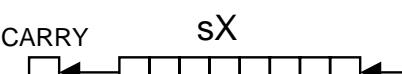
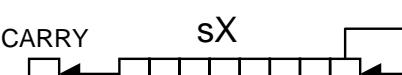
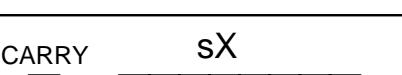
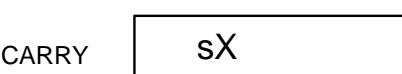
Each instruction must specify the register as 's' followed by a hexadecimal digit. The assembler supports register naming to simplify the process.

Simplify the process.																		<u>Bit 2</u>	<u>Bit 1</u>	<u>Bit 0</u>	Instruction
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	1	1	0	SR0 sX
1	0	0	0	0	0	x	x	x	x	0	0	0	0	1				1	1	1	SR1 sX
																		0	1	0	SRX sX
																		0	0	0	SRA sX
																		1	0	0	RR sX
<u>sX</u>																					



SL0, SL1, SLX, SLA, RL

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

SL0 sX		ZERO	<input type="checkbox"/> ?	Set if all bits of result are zero. Reset in all other cases.
SL1 sX		ZERO	<input checked="" type="checkbox"/> 0	
SLX sX		ZERO	<input type="checkbox"/> ?	Set if all bits of result are zero. Reset in all other cases.
SLA sX		ZERO	<input type="checkbox"/> ?	Set if all bits of result are zero. Reset in all other cases.
RL sX		ZERO	<input type="checkbox"/> ?	Set if all bits of result are zero. Reset in all other cases.

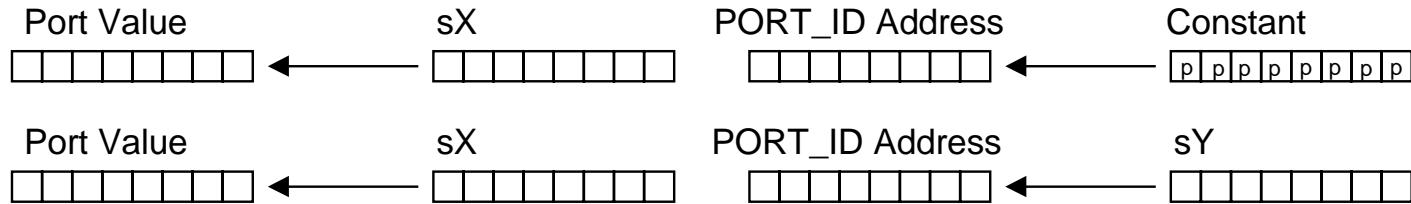
Each instruction must specify the register as 's' followed by a hexadecimal digit. The assembler supports register naming to simplify the process.

	Bit 2	Bit 1	Bit 0	Instruction
17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0				
1 0 0 0 0 0 x x x x 0 0 0 0 0 0	1	1	0	SL0 sX
	1	1	1	SL1 sX
	1	0	0	SLX sX
	0	0	0	SLA sX
	0	1	0	RL sX



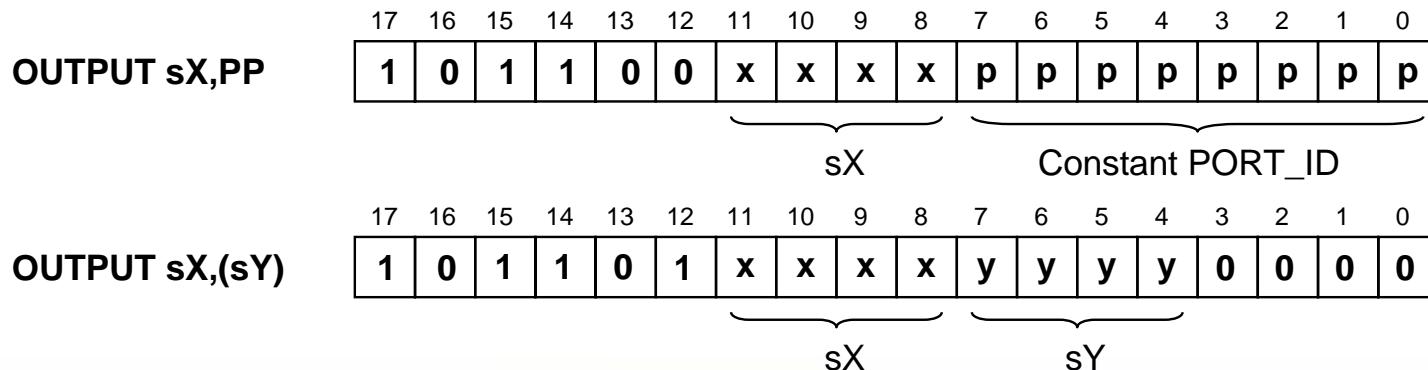
OUTPUT

The OUTPUT instruction enables the contents of any register to be transferred to logic external to KCPSM3. The port address (in the range 00 to FF) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.



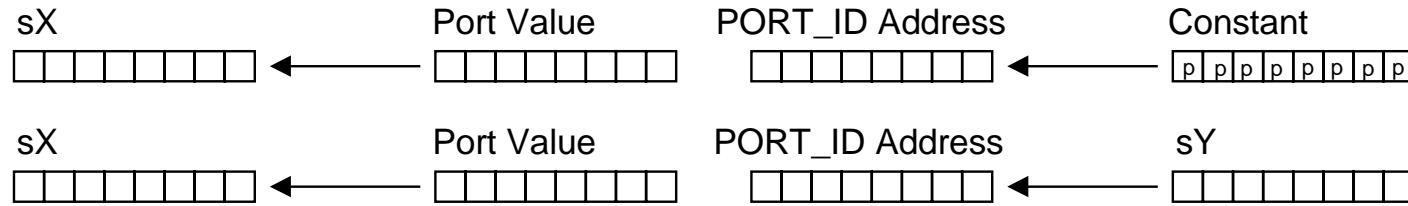
The user interface logic is required to decode the PORT_ID port address value and capture the data provided on the OUT_PORT. The WRITE_STROBE is set during an output operation (see ‘READ and WRITE STROBES’), and should be used to clock enable the capture register or write enable a RAM (see ‘Design of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

Each OUTPUT instruction must specify the source register as ‘s’ followed by a hexadecimal digit. It must then specify the output port address using a register value in a similar way or specify an 8-bit constant port identifier using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.



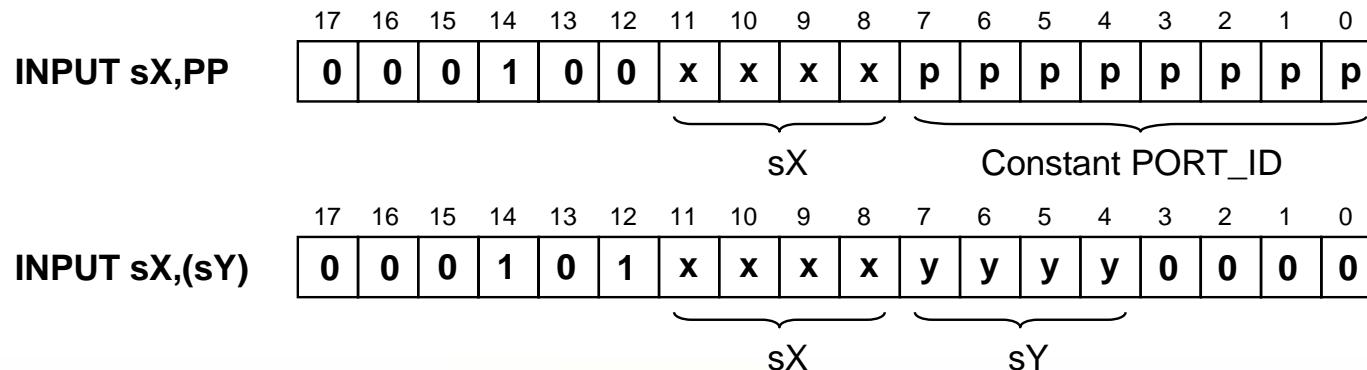
INPUT

The INPUT instruction enables data values external to KCPSM3 to be transferred into any one of the internal registers. The port address (in the range 00 to FF) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.



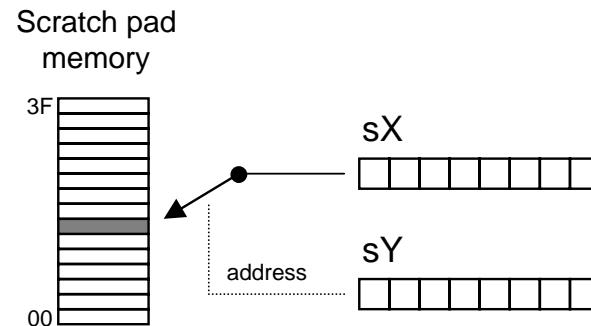
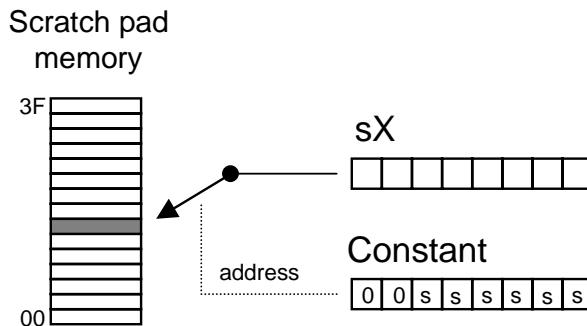
The user interface logic is required to decode the PORT_ID port address value and supply the correct data to the IN_PORT. The READ_STROBE is set during an input operation (see 'READ and WRITE STROBES'), but it is not always necessary for the interface logic to decode this strobe. However, it can be useful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

Each INPUT instruction must specify the destination register as 's' followed by a hexadecimal digit. It must then specify the input port address using a register value in a similar way or specify an 8-bit constant port identifier using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.



STORE

The STORE instruction enables the contents of any register to be transferred to the 64-byte internal scratch pad memory. The storage address (in the range 00 to 3F) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.

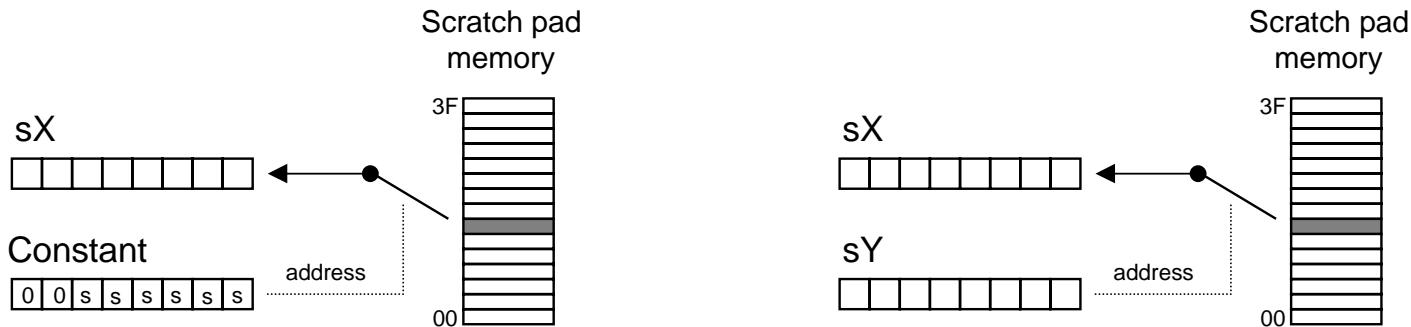


Each STORE instruction must specify the source register as 's' followed by a hexadecimal digit. It must then specify the storage address using a register value in a similar way or specify a 6-bit constant storage address using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process. Although the assembler will reject constants greater than 3F, it is the responsibility of the programmer to ensure that the value of 'sY' is within the address range.

STORE sX,PP	17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
	1 0 1 1 1 0 x x x x 0 0 s s s s s s s	
	sX	Constant address
STORE sX,(sY)	17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
	1 0 1 1 1 1 x x x x y y y y 0 0 0 0 0	
	sX	sY

FETCH

The FETCH instruction enables data held in the 64-byte internal scratch pad memory to be transferred any of the internal registers. The storage address (in the range 00 to 3F) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.

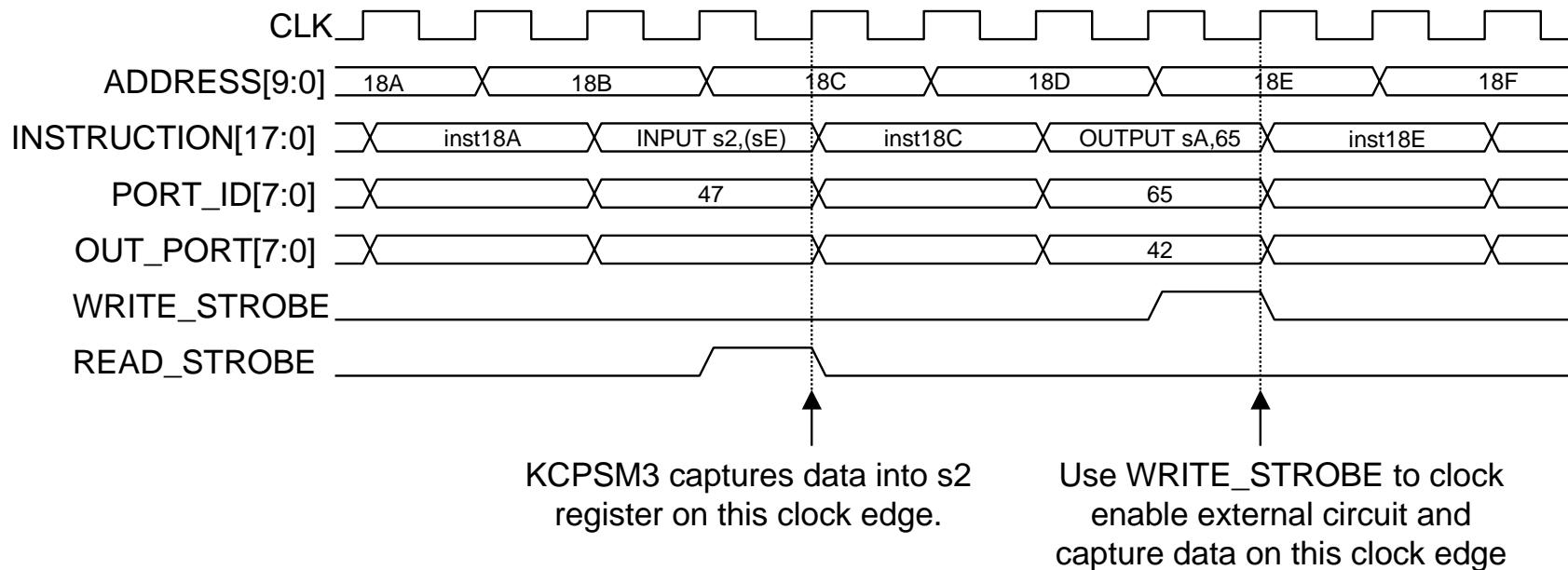


Each FETCH instruction must specify the destination register as 's' followed by a hexadecimal digit. It must then specify the storage address using a register value in a similar way or specify a 6-bit constant storage address using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process. Although the assembler will reject constants greater than 3F, it is the responsibility of the programmer to ensure that the value of 'sY' is within the address range.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FETCH sX,PP	0	0	0	1	1	0	x	x	x	x	0	0	s	s	s	s	s	s
sX																Constant address		
FETCH sX,(sY)	0	0	0	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
sX								sY										

READ and WRITE STROBES

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



PORT_ID[7:0] is valid for 2 clock cycles providing additional time for external decoding logic and enabling the connection of synchronous RAM. The WRITE_STROBE is provided on the second clock cycle to confirm an active write by KCPSM3. In most cases, the READ_STROBE will not be utilised 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.

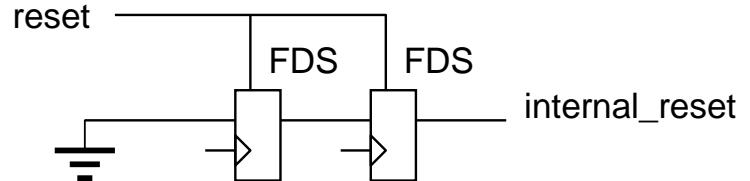
Note for timing critical designs, your timing specifications can allow 2 clock cycles for PORT_ID and data paths, and only the strobes need to be constrained to a single clock cycle. Ideally, a pipeline register can be inserted where possible (see ‘Design of Input Ports’, ‘Design of Output Ports’ and ‘Connecting Memory’).



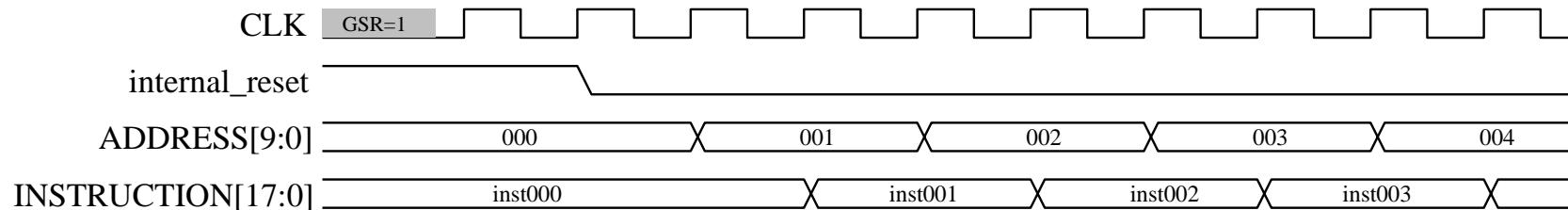
RESET

KCPSM3 contains an internal reset control circuit to ensure the correct start up of KCPSM3 following device configuration or global reset (GSR). This reset can also be activated within your design.

The KCPSM3 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 right) ensures that the release of the internal reset is clean and controlled.

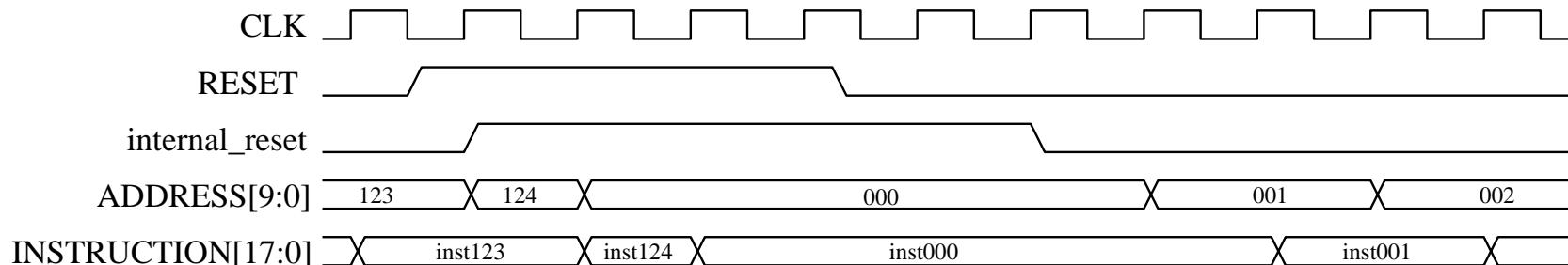


Release of Reset after configuration.



Application of user reset input

The reset input can be tied to logic ‘0’ if not required and the ‘filter’ will still be used to ensure correct power-up sequence.

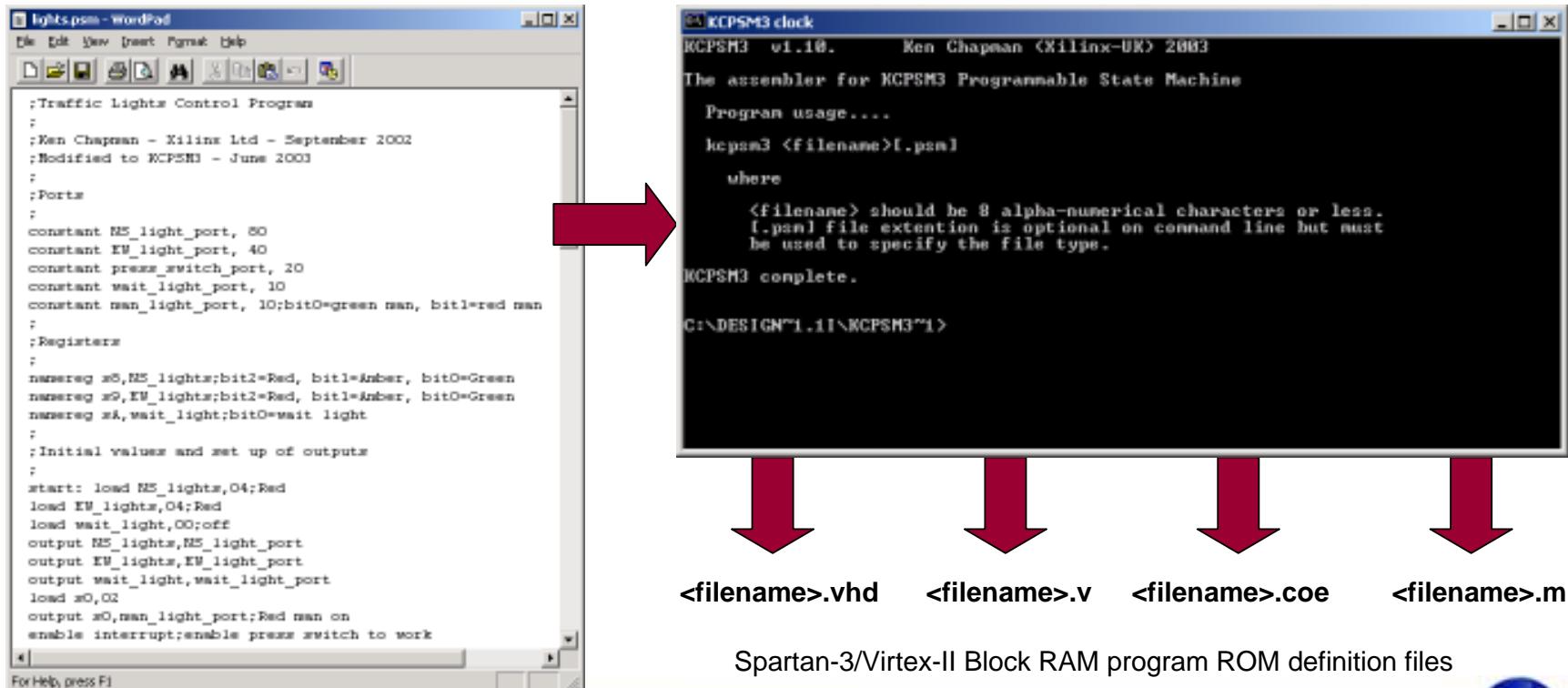


KCPSM3 Assembler

The KCPSM3 Assembler is provided as a simple DOS executable file together with three template files. Copy all the files KCPSM3.EXE, ROM_form.vhd, ROM_form.v and ROM_form.coe into your working directory.

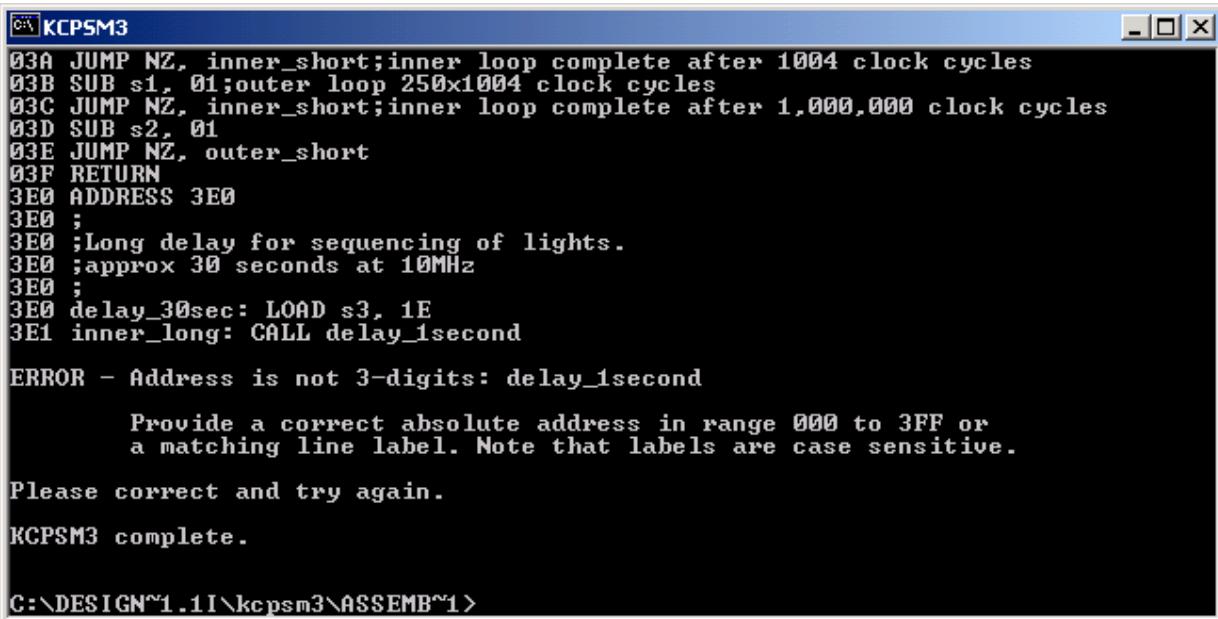
Programs are best written with either the standard Notepad or Wordpad tools. The file is saved with a '.psm' file extension (8 character name limit).

Open a DOS box and navigate to the working directory. Then run the assembler 'kcpsm3 <filename>[.psm]' to assemble your program. It all happens very fast!!



Assembler Errors

The assembler will stop as soon as an error is detected. A short message will be displayed to help determine the reason for the error. The assembler will also display the line it was analyzing when it detected the problem. The user should fix each reported problem in turn and re-execute the assembler.



The screenshot shows a window titled "KCPSM3" displaying assembly code. The code includes various instructions like JUMP, SUB, and CALL, along with comments and labels. An error message is present at the bottom of the code area:

```
03A JUMP NZ, inner_short;inner loop complete after 1004 clock cycles
03B SUB s1, 01;outer loop 250x1004 clock cycles
03C JUMP NZ, inner_short;inner loop complete after 1,000,000 clock cycles
03D SUB s2, 01
03E JUMP NZ, outer_short
03F RETURN
3E0 ADDRESS 3E0
3E0 ;
3E0 ;Long delay for sequencing of lights.
3E0 ;approx 30 seconds at 10MHz
3E0 ;
3E0 delay_30sec: LOAD s3, 1E
3E1 inner_long: CALL delay_isecond

ERROR - Address is not 3-digits: delay_isecond
Provide a correct absolute address in range 000 to 3FF or
a matching line label. Note that labels are case sensitive.

Please correct and try again.

KCPSM3 complete.

C:\DESIGN\1.1I\kcpsm3\ASSEMB\1>
```

Annotations with arrows point to specific parts of the window:

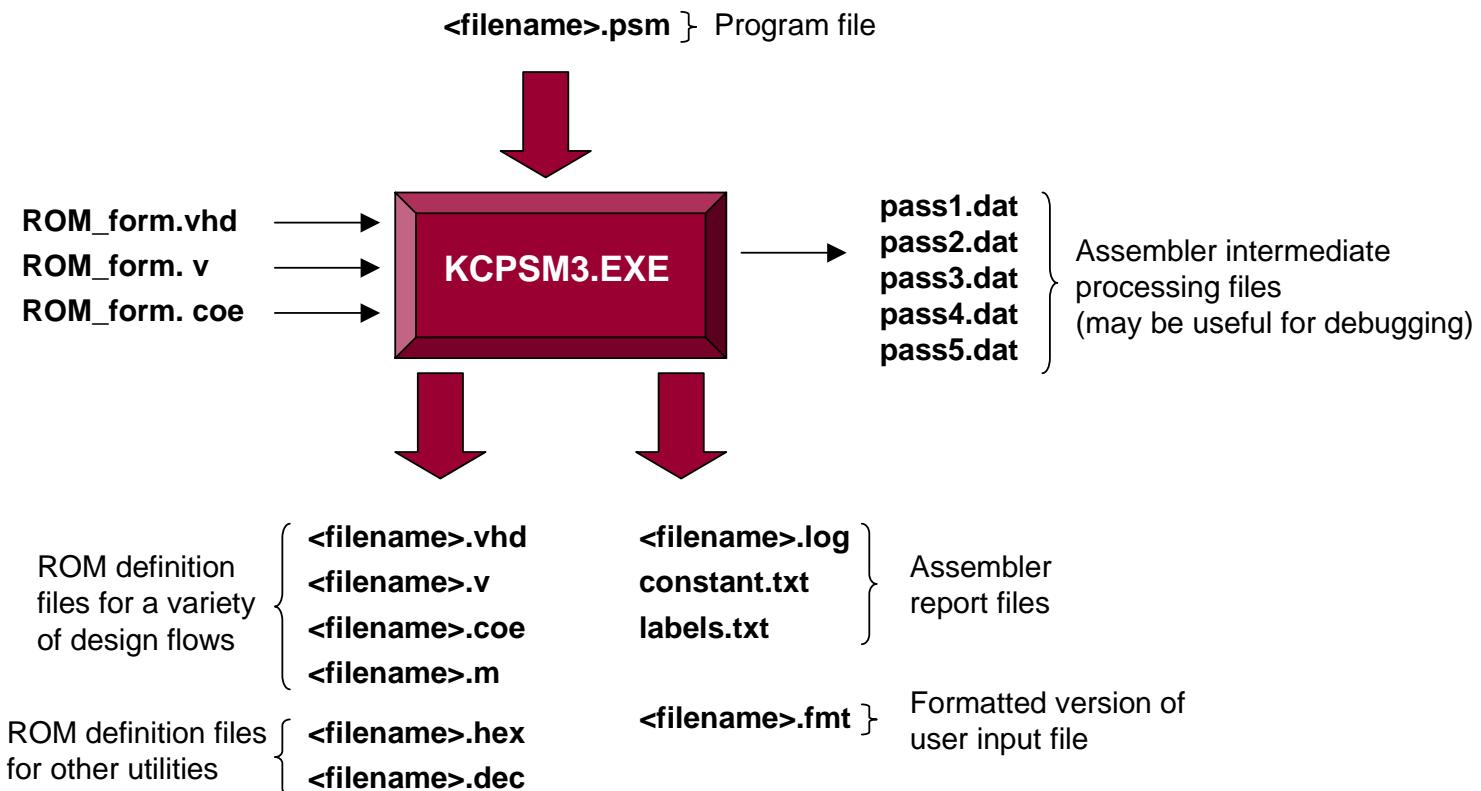
- A brace on the left side of the code area groups the first few lines of code as "Previous Progress".
- An arrow points to the line "3E0 delay_30sec: LOAD s3, 1E" with the label "Line being processed".
- An arrow points to the error message text with the label "Error message".

Since the execution of the assembler is very fast, it is unlikely that you will be able to 'see' it making progress and the display will appear to be immediate. If you would like to review everything that the assembler has written to the screen, the DOS output can be redirected to a text file using..... **kcpsm3 <filename>[.psm] > screen_dump.txt**



Assembler Files

The KCPSM3 assembler actually reads four input files and generates 15 output files. These are described in more detail on the following pages.



Note - All output files are overwritten each time the assembler is executed.

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

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 Spartan-3, Virtex-II or Virtex-IIIPRO configured as a ROM. You can adjust this template to define the type of memory you want. The template supplied includes some additional notes on how the template works

ROM_form.vhd

```
entity {name} is
    Port (      address : in std_logic_vector(9 downto 0);
                instruction : out std_logic_vector(17 downto 0);
                           clk : in std_logic);
end {name};
--
architecture low_level_definition of {name} is
.
.
attribute INIT_00 of ram_1024_x_18 : label is  "{INIT_00}";
attribute INIT_01 of ram_1024_x_18 : label is  "{INIT_01}";
attribute INIT_02 of ram_1024_x_18 : label is  "{INIT_02}";
```

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.

The template contains some special text strings contained in {} brackets. These are `{begin template}`, `{name}`, and a whole family of initialisation 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.v File

This file provides the template for the Verilog 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.v template file defines a Single Port Block RAM for Spartan-3, Virtex-II or Virtex-II PRO configured as a ROM. You can adjust this template to define the type of memory you want. The template supplied includes some additional notes on how the template works

ROM_form.v

```
module {name} (address, instruction, clk);

input [9:0] address;
input clk;

output [17:0] instruction;
.

.

defparam ram_1024_x_18.INIT_00 = 256'h{INIT_00};
defparam ram_1024_x_18.INIT_01 = 256'h{INIT_01};
defparam ram_1024_x_18.INIT_02 = 256'h{INIT_02};
```

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

The template contains some special text strings contained in {} brackets. These are {begin template}, {name}, and a whole family of initialisation identifiers such as {INIT_01}. The assembler uses {begin template} to identify where the Verilog 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 Spartan-3, Virtex-II or Virtex-II PRO in which the A-port is read only and the B-port is read/write. You can adjust this template to define the type of memory you want Core Generator to implement.

ROM_form.coe

```
component_name={name};  
width_a=18;  
depth_a=1024;  
. . .  
memory_initialization_radix=16;  
global_init_value=00000;  
memory_initialization_vector=
```

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 may contain the special text string **{name}** which the assembler will intercept and replace with the name of the program file. In this example you can see that {name} has been replaced with 'simple'.

It is vital that the last line of the template contains the key words...
memory_initialization_vector=

These are used by the Core Generator to identify that the data values follow, and the assembler will append the 1024 values required. Indeed, the template could simply contain this one line provided the Core Generator GUI is used to define all other parameters.

KCPSM3 Assembler

<filename>.coe

```
component_name=simple;  
width_a=18;  
depth_a=1024;  
. . .  
memory_initialization_radix=16;  
global_init_value=00000;  
memory_initialization_vector=  
01400, 23412, 09401, 100A0, 0C018, 35401, 34000, 00000, ...
```



<filename>.fmt File

When a program passes through the assembler additional files to the '.vhd' and '.coe' files are produced to be of assistance to the programmer. One of these is called '<filename>.fmt'. This file is the original program but formatted to look nice. Looking at this file is also an easy way to see that everything has been interpreted the way you had expected.

<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
```



KCPSM3 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
```

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



Write your PSM program quickly and then use KCPSM3 to make a nice formatted version for you to adopt as your own.

<filename>.log File

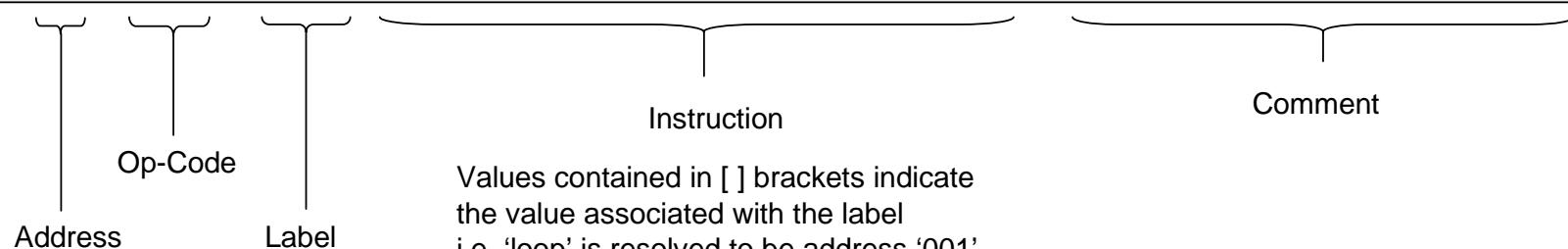
The '.log' file provides you with the most detail about the assembly process which has been performed. This is where you can observe how each instruction and directive has been used. Address and op-code values are associated with each line of the program and the actual values of addresses, registers, and constants defined by labels are specified.

<filename>.log

```
KCPSM3 Assembler log file for program 'simple.psm'.
Generated by KCPSM3 version 1.01
Ken Chapman (Xilinx Ltd) 2003.

Addr Code

000      CONSTANT max_count, 18           ;count to 24 hours
000      NAMEREG s4, counter_reg          ;define register for counter
000      CONSTANT count_port, 12          ;
000 00400 start: LOAD counter_reg[s4], 00   ;initialise counter
001 2C412  loop: OUTPUT counter_reg[s4], count_port[12]
002 18401      ADD counter_reg[s4], 01       ;increment
003 01040      LOAD s0, counter_reg[s4]
004 18018      ADD s0, max_count[18]        ;test for max value
005 35401      JUMP NZ, loop[001]          ;next count
006 34000      JUMP start[000]            ;reset counter
```



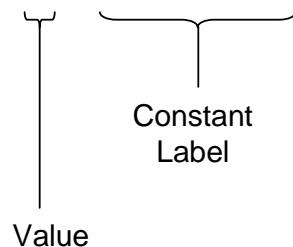
constant.txt & labels.txt Files

These two files provide a list of the line labels and their associated addresses, and a list of constants and their values as defined by 'constant' directives in the program file. These can be useful during the development and testing of larger programs.

constant.txt

Table of constant values and their specified constant labels.

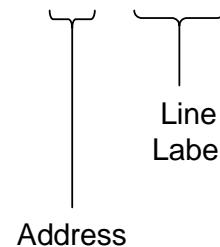
```
18 max_count  
12 count_port
```



labels.txt

Table of addresses and their specified labels.

```
000 start  
001 loop
```



pass.dat Files

These are really internal files to the assembler and represent intermediate stages of the assembly process. These files will typically be ignored, but may just help in identifying how the assembler has interpreted the program file syntax. The files are automatically deleted at the start of the assembly process. If there is an error detected by the assembler, the '.dat' files will only be complete until the point of the last successful processing.

Part of pass1.dat

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

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
```

Part of pass5.dat

```
ADDRESS-002
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
```

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

Program Syntax

Probably the best way to understand what is and is not valid syntax is to look at the examples and try the assembler. However there are some simple rules which are of assistance from the beginning.

No blank lines - A blank line will be ignored by the assembler and removed from any formatted files. If you would like to keep a line use a blank comment (a semicolon).

Comments - Any item on a line following a semi-colon (;) will be ignored by the assembler. Whilst comments are useful, it is helpful if they are kept concise otherwise you will have very long lines and find it difficult to print out programs and log files.

Registers - All registers should be defined as the letter 's' immediately followed by one hexadecimal digit the range 0 to F. The assembler will accept any mixture of upper and lower case characters and automatically convert them to the 'sX' format where 'X' is one of 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. The NAMEREG directive can be used to assign new register names.

Constants - A constant must be specified using hexadecimal. Data values and Port Addresses in range 00 to FF. Memory store values in the range 00 to 3F and program addresses in the range 000 to 3FF. The assembler will accept any mixture of upper and lower case characters and automatically convert them to upper case.

Labels - Labels are any text string which the user defines. Labels are case sensitive for additional flexibility. Labels must not contain any spaces although the under-score character is supported. Valid characters are '0' to '9', 'a' to 'z', and 'A' to 'Z'. Again it is helpful for labels to be reasonably concise if only for the formatting of a program to be reasonable. Labels which could be confused with hexadecimal values or register specifications are rejected by the assembler.

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

```
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
```

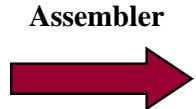


Program Syntax

Instructions - The instructions should be of the format described in the “KCPSM3 instruction set” page of this document. 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 will accept any mixture of upper and lower case characters for the instruction and automatically convert them to upper case. The following examples all show acceptable instruction specifications, but the formatted output shows how it was expected.

load s5,7E	LOAD s5, 7E
AddCY s8,SE	ADDCY s8, se
ENABLE interrupt	ENABLE INTERRUPT
Output S2, (S8)	OUTPUT s2, (s8)
jump Nz, 2a7	JUMP NZ, 2A7
ADD sF, step_value	ADD sF, step_value
INPUT S9,28	INPUT s9, 28
s11 se	SL1 se
store S8,(sf)	STORE s8, (sf)



Most other syntax issues will be solved by reading the error messages provided by the assembler.



CONSTANT Directive

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 KCPSM3.

The CONSTANT directive provides a way to assign an 2-digit hexadecimal value to a label. In this way the program can declare constants such as port and storage 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 using absolute values in the program lines. The following example illustrates the directive syntax and its uses.

```
CONSTANT light_port, 03      ;light sensor port
CONSTANT light_sensor, 01    ;bit0 is light sensor
CONSTANT temp_sensor, 40    ;temperature sensor port
NAMEREG sF, light_count_msb ;16-bit light pulse counter
NAMEREG sE, light_count_lsb
NAMEREG sD, new_temp        ;current temperature
CONSTANT peak_temp, 2E      ;peak temperature memory
light_test: INPUT s1, light_port
TEST s1, light_sensor
JUMP Z, temp_test           ;jump if no light
ADD light_count_lsb, 01     ;increment counter
ADDCY light_count_msb, 00
temp_test: INPUT new_temp, temp_sensor ;read temperature
FETCH s2, peak_temp
COMPARE s2, new_temp         ;compare with peak value
JUMP NC, light_test          ;new value is smaller
STORE new_temp, peak_temp    ;write new peak value
JUMP light_test
```

Note - A constant is global.
Even if a constant is defined at the end of the program file, it can be used in instructions anywhere in the program.

Constant names must not contain any spaces although the under-score character is supported. Valid characters are '0' to '9', 'a' to 'z', and 'A' to 'Z'.

'`light_port`' and '`temp_sensor`' are used to specify port addresses. This is particularly useful when defining the hardware interface, and allows the program to be developed before the I/O addresses are fully defined. '`light_sensor`' is being used to specify a data constant which in this case identifies which bit is to be tested. '`peak_temp`' defines a scratch pad memory location which is then used to hold a variable.



NAMEREG Directive

The NAMEREG directive provides a way to assign a new name to any of the 16 registers. In this way the program can refer to ‘variables’ by name rather than as absolute register specifications. By naming registers in this way it is often easier to understand the meaning in the program without the need for so many comments. It can also help to prevent inadvertent reuse of a register with associated data corruption.

Important - The NAMEREG directive is applied in-line with the code by the assembler. Before the NAMEREG directive, the register will be named in the ‘sX’ style. Following the directive, only the new name will apply. It is also possible to rename a register again (i.e. NAMEREG counter_reg, hours) and only the new name will apply in the subsequent program lines.

```
CONSTANT light_port, 03      ;light sensor port
CONSTANT light_sensor, 01    ;bit0 is light sensor
CONSTANT temp_sensor, 40    ;temperature sensor port
NAMEREG sF, light_count_msb ;16-bit light pulse counter
NAMEREG sE, light_count_lsb
NAMEREG sD, new_temp        ;current temperature
CONSTANT peak_temp, 2E      ;peak temperature memory
light_test: INPUT s1, light_port      ;test for light
      TEST s1, light_sensor
      JUMP Z, temp_test       ;jump if no light
      ADD light_count_lsb, 01 ;increment counter
      ADDCY light_count_msb, 00
temp_test: INPUT new_temp, temp_sensor ;read temperature
      FETCH s2, peak_temp
      COMPARE s2, new_temp    ;compare with peak value
      JUMP NC, light_test     ;new value is smaller
      STORE new_temp, peak_temp ;write new peak value
      JUMP light_test
```

Register names must not contain any spaces although the under-score character is supported. Valid characters are ‘0’ to ‘9’, ‘a’ to ‘z’, and ‘A’ to ‘Z’.

The register ‘sD’ has been renamed to be ‘**new_temp**’ and is then used in multiple instructions making it clear what the meaning of the register contents actually are.



ADDRESS Directive

The ADDRESS directive provides a way 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 3-digit hexadecimal value in the range '00' to '3FF'.

In the following code segment, the ADDRESS directive defines the address for the interrupt vector.

```
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 3FF                     ;Interrupt vector
      JUMP ISR
;end of program
```

The log file clearly shows that the ADDRESS directive has forced 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.

```
3E3 357E1      JUMP NZ, inner_long[3E1]
3E4 2A000      RETURN
3E5          ;Interrupt Service Routine
3E5 00A01    ISR: LOAD wait_light[sA], 01           ;register press of switch
3E6 2CA10      OUTPUT wait_light[sA], wait_light_port[10] ;turn on light
3E7 38000      RETURNI DISABLE                  ;continue light sequence but...
3FF          ADDRESS 3FF                     ;Interrupt vector
3FF 343E5      JUMP ISR[3E5]
3FF          ;end of program
```

KCPSM and KCPSM2 Compatibility

KCPSM and KCPSM2 are very much ‘brothers’ with many similarities (see ‘PicoBlaze Comparison’). However, each has been tuned to the specific device architecture so there are differences.

Common points

The KCPSM3 assembler has slightly different rules concerning which labels for lines, constants, and registers are acceptable. Therefore, it may be necessary to adjust some of the user names in your program code. Typically, labels are nicely ‘descriptive’ and this issue will not be encountered.

The KCPSM3 macro has an INTERRUPT_ACK output signal which the previous versions did not have. It is not vital to use this signal in your design, but should be included in the component port definitions.

The internal scratch pad memory will often mean that external memory connected to I/O ports can be removed. This will simplify the logic design and require the code to reflect the use of STORE and FETCH instructions in place of INPUT and OUTPUT.

KCPSM to KCPSM3

KCPSM3 is in every way a superset of KCPSM so there will be very few issues migrating a KCPSM based design and code. The address range of KCPSM3 supports a program which is four times larger than KCPSM and therefore all programs will be able to fit. Code will need to reflect that absolute address values need to be specified with 3 hexadecimal digits (not 2). The use of line labels will mean that most cases will be handled automatically by the assembler, but special care should be taken with ADDRESS directives. Most critical is that the interrupt vector will need to be located at ‘3FF’ (not FF).

KCPSM2 to KCPSM3

KCPSM3 has 16 registers compared with the 32 registers of KCPSM2. The default register names used in KCPSM2 are ‘s00’ to ‘s1F’ and will need to be modified to conform to the default names ‘s0’ to ‘sF’ available in KCPSM. Although the use of NAMEREG directives will be helpful, some fundamental changes will almost certainly be required to compensate for the lower number of available registers. The internal scratch pad memory provides 64 locations which should more than compensate for the lower number of registers but obviously requires a change to the coding style. The program address range and interrupt vector are identical.



PicoBlaze Comparison

This chart shows a comparison of the features offered by the FPGA variants of PicoBlaze. XAPP387 describes the CoolRunner implementation of an 8-bit micro controller which was also based on the original KCPSM processor.

	KCPSM	KCPSM2	KCPSM3
Target Devices	Spartan-II, Spartan-IIIE, Virtex, Virtex-E	Virtex-II, Virtex-IIIPRO	Spartan-3, Virtex-II, Virtex-IIIPRO
Program Size	256 instructions (256x16 Block RAM)	1024 instructions (1024x18 Block RAM)	1024 instructions (1024x18 Block RAM)
Registers	16	32	16
Scratch-Pad Memory	-	-	64 Bytes
Size	76 Slices	84 Slices	96 Slices
CALL/RETURN stack	15 levels	31 levels	31 levels
Features and Comments	Smallest and oldest! Very well used and proven. Relatively small program space.	Register rich. Virtex-II devices only. Can <u>not</u> migrate design directly to Spartan-3.	COMPARE and TEST instructions, PARITY test, Scratch-pad memory, INTERRUPT_ACK signal

As with most things, there is a clear trend for PicoBlaze to become larger as more features are added. The author welcomes all feedback regarding this trend to determine the size acceptable for a programmable state machine (PSM).



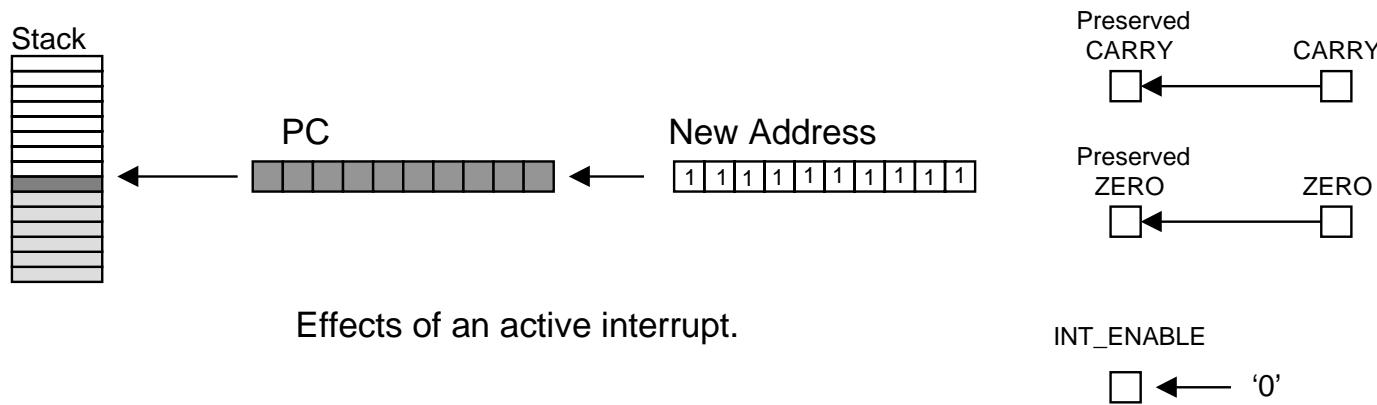
Interrupt Handling

Effective interrupt handling is a skillful task and this document does not attempt to explain how and when an interrupt should be used. The information supplied should be adequate for the capability of KCPSM3 to be assessed and for interrupt based systems to be created.

Default State - By default the interrupt input is disabled. This means that the entire 1024 words of program space can be 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 a program execution where an interrupt would be unacceptable, a DISABLE INTERRUPT can be used. Since an active interrupt will automatically disable the interrupt input, the interrupt service routine will end with a RETURNI instruction which also includes the option to ENABLE or DISABLE the interrupt input as it returns to the main program.

What happens during an interrupt? The program counter is pushed onto the stack and the values of the CARRY and ZERO flags are preserved (to be restored by the RETURNI instruction). The interrupt input is automatically disabled. Finally the program counter is forced to address 3FF (last program memory location) from which the next instruction is executed.



Basics of Interrupt Handling

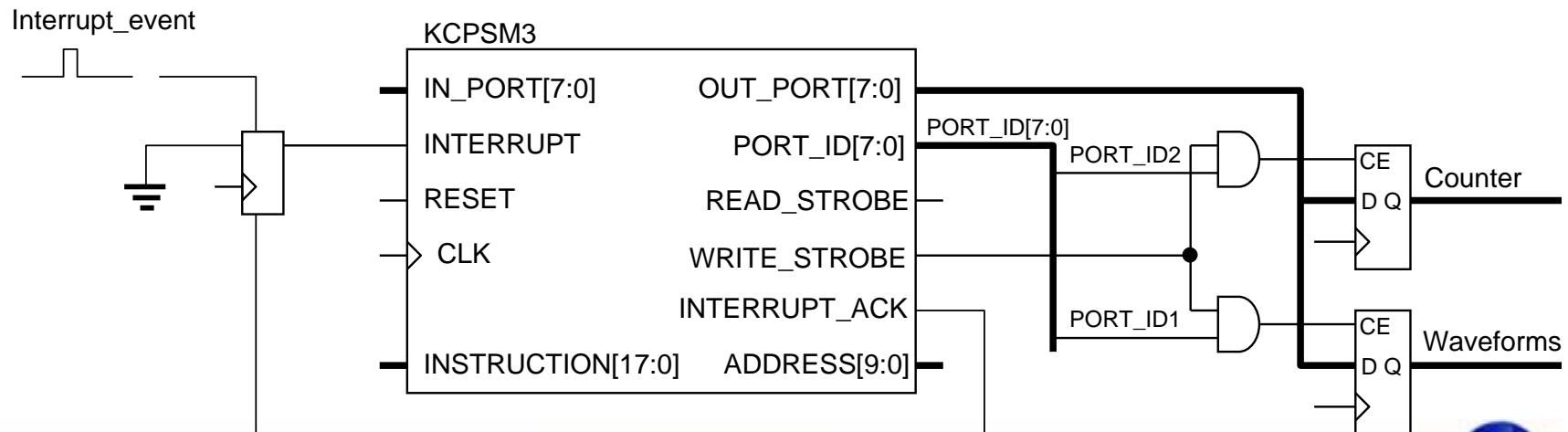
Since the interrupt will force the program counter to address '3FF' it will generally be necessary to ensure that a jump vector to a suitable interrupt service routine (ISR) is located at this address otherwise the program will 'roll over' to address zero.

In most cases an ISR will be provided. The routine can be located at any position in the program and jumped to by the interrupt vector located at the '3FF' address. The ISR will perform the required tasks and then end in RETURNI with ENABLE or DISABLE.

Simple Example - The following example illustrates a very simple interrupt handling routine.....

The KCPSM3 is generally involved with generating 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 7 times in a loop.

When an interrupt is asserted, the KCPSM3 breaks off from the waveform generation and simply increments a separate counter register (sA) and writes the counter value to the 'counter_port' (port address 04).



Example Design (VHDL)

The following VHDL shows the addition of the data capture registers and interrupt control to the processor. Note the simplified port decoding logic through careful selection of port addresses. The complete VHDL file is supplied as 'kcpsm3_int_test.vhd'.

```
IO_registers: process(clk)
begin

    if clk'event and clk='1' then

        -- waveform register at address 02
        if port_id(1)='1' and write_strobe='1' then
            waveforms <= out_port;
        end if;

        -- Interrupt Counter register at address 04
        if port_id(2)='1' and write_strobe='1' then
            counter <= out_port;
        end if;

    end if;
end process IO_registers;
```

```
interrupt_control: process(clk)
begin

    if clk'event and clk='1' then

        if interrupt_ack='1' then
            interrupt <= '0';
        elsif interrupt_event='1' then
            interrupt <= '1';
        else
            interrupt <= interrupt;
        end if;

    end if;
end process interrupt_control;
```

Interrupt Service Routine

In the assembler log file for the example, it can be seen that the interrupt service routine has been forced to compile at address '2B0', and that the waveform generation is located in the base addresses. This makes it easier to observe the interrupt in action in the operation waveforms. This program is supplied as 'int_test.psm' for you to assemble yourself.

```
000 ;Interrupt example
000 ;
000 CONSTANT waveform_port, 02 ;bit0 will be data
000 CONSTANT counter_port, 04
000 CONSTANT pattern_10101010, AA
000 NAMEREG sA, interrupt_counter
000 ;
000 00A00 start: LOAD interrupt_counter[sA], 00 ;reset interrupt counter
001 002AA LOAD s2, pattern_10101010[AA] ;initial output condition
002 3C001 ENABLE INTERRUPT
003 ;
003 2C202 drive_wave: OUTPUT s2, waveform_port[02] ;Main program delay loop where
004 00007 LOAD s0, 07 most time is spent
005 1C001 loop: SUB s0, 01 } ;delay size
006 35405 JUMP NZ, loop[005] } ;delay loop
007 0E2FF XOR s2, FF
008 34003 JUMP drive_wave[003] ;toggle waveform
009 ;
0B0 ADDRESS 2B0 ←
0B0 18A01 int_routine: ADD interrupt_counter[sA], 01 ;increment counter
0B1 2CA04 OUTPUT interrupt_counter[sA], counter_port[04]
0B2 38001 RETURNI ENABLE
0B3 ;
0FF ADDRESS 3FF
0FF 342B0 JUMP int_routine[2B0] }
```

Diagram annotations:

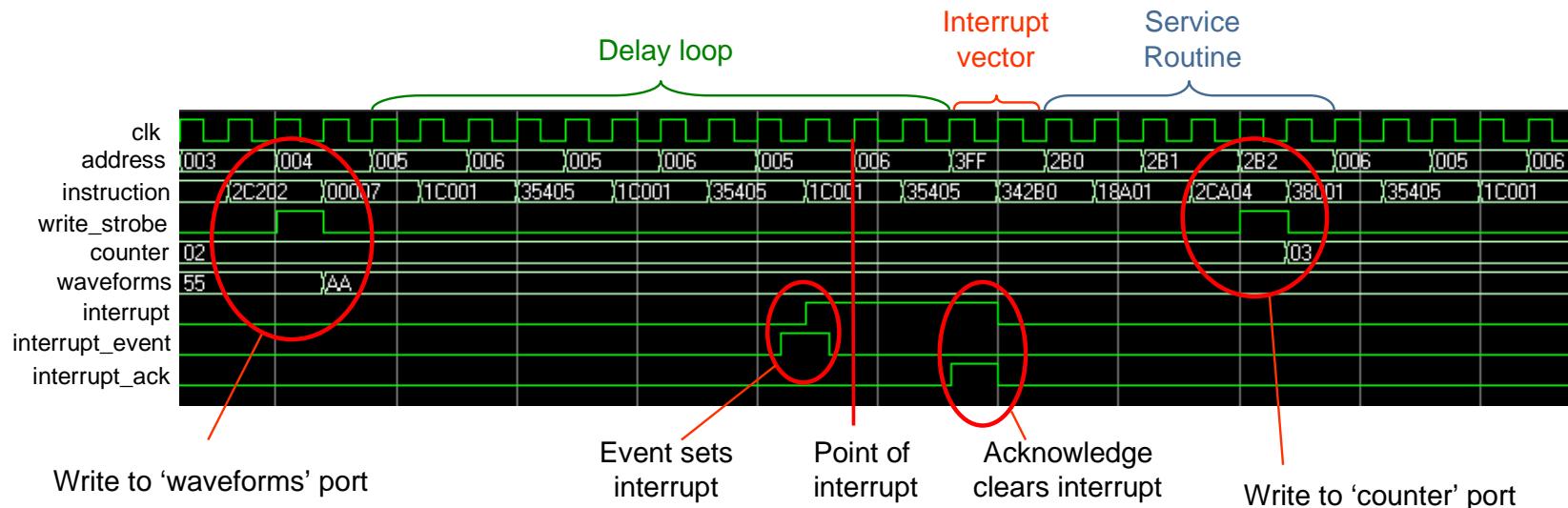
- A callout box points to the line "ADDRESS 2B0 ←" with the text "Main program delay loop where most time is spent".
- A callout box points to the line "JUMP int_routine[2B0]" with the text "Interrupt Service Routine (located at address 2B0 onwards)".
- A callout box points to the line "ADDRESS 3FF" with the text "Interrupt vector set at address 3FF and causing JUMP to service routine".

Interrupt Operation

The waveforms below taken from an actual ModelSim-XE simulation show the operation of KCPSM3 when executing the example program at the time of an interrupt. The VHDL test bench used to generate these waveforms is supplied as ‘testbench.vhd’.

By observing the address bus, it is possible to see that the program is busy generating the waveforms and even shows the ‘waveforms’ port being written the ‘AA’ pattern value. Then whilst in the delay loop which repeats addresses ‘005’ and ‘006’ it receives an interrupt pulse.

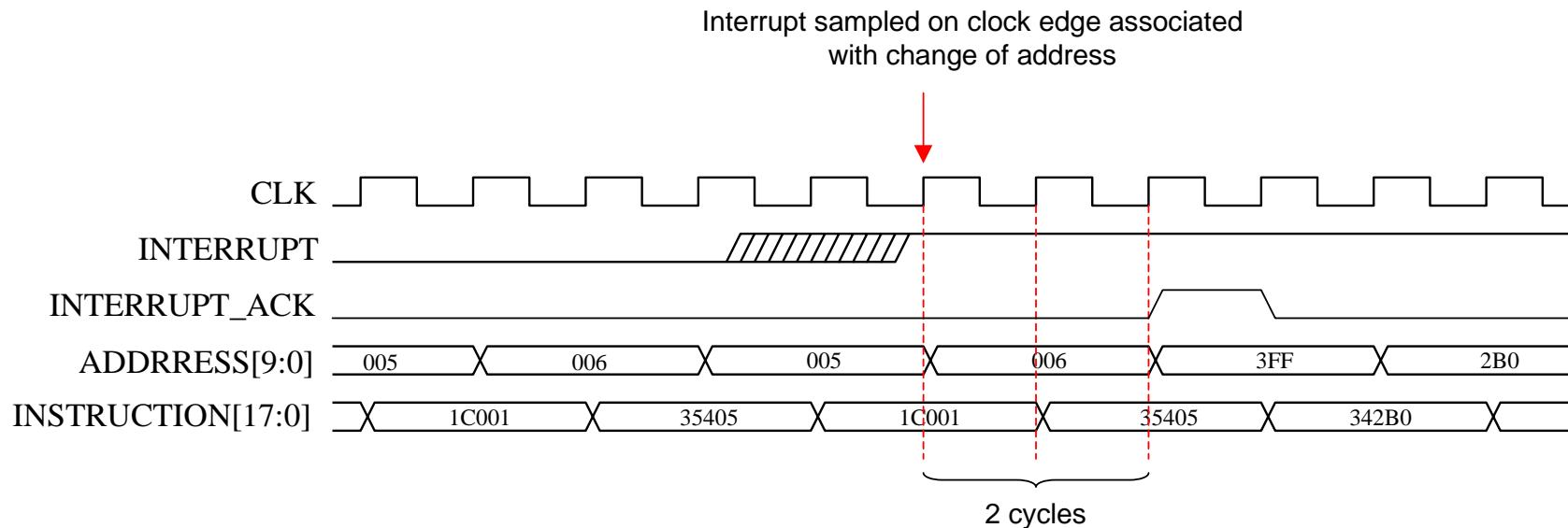
It can be seen that KCPSM3 took a few clock cycles to respond to this particular pulse (see ‘timing of interrupt pulses’) before forcing the address bus to ‘3FF’ and issuing an INTERRUPT_ACK pulse. From ‘3FF’, the obvious JUMP to the service routine located at ‘2B0’ can be seen to follow and a new counter value (in this case ‘03’) is written to the ‘counter’ port.



The operation of a KCPSM3 interrupt can also be observed. It can be seen that the last address active before the interrupt is ‘006’. The JUMP NZ instruction obtained at this address (op-code 35405) is not executed. The flags preserved are those which were set at the end of the instruction at the previous address (SUB s0,01). The RETURNI has restored the flags and returned the program to address ‘006’ in order that the JUMP NZ instruction can at last be executed.

Timing of Interrupt Pulses

It is clear from the previous simulation waveforms that the constant two cycles per instruction is maintained at all times. Since this includes an interrupt, the use of single cycle pulse for interrupt can be risky. However, the following waveform can be used to determine the exact cycle on which the interrupt is observed and the true reaction rate of KCPSM3.



It is therefore advisable that an interrupt signal should be active for a minimum of two KCPSM3 rising clock cycle edges. It is generally advisable to use the INTERRUPT_ACK signal in a similar way to that demonstrated in the example to ensure that an interrupt is not missed.

When using logic to combine multiple sources of interrupt, a typical interrupt service routine will read a specific port to determine the reason for interrupt. In this case, the READ_STROBE and PORT_ID can be decoded and used to clear the external interrupt register.

CALL/RETURN Stack

KCPSM3 contains an automatic embedded stack 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 initialised or require any control by the user. However, **the stack can only support nested subroutine calls to a depth of 31**.

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 31 (1F hex) is calculated to be 496 (01F0 hex). This is achieved by using a recursive call of a subroutine and results in the full depth of the call/return stack being utilised. Obviously, this is not a particularly efficient implementation of this algorithm, but it does fully test the stack.

```
NAMEREG s0, total_low
NAMEREG s1, total_high
NAMEREG s8, value
;

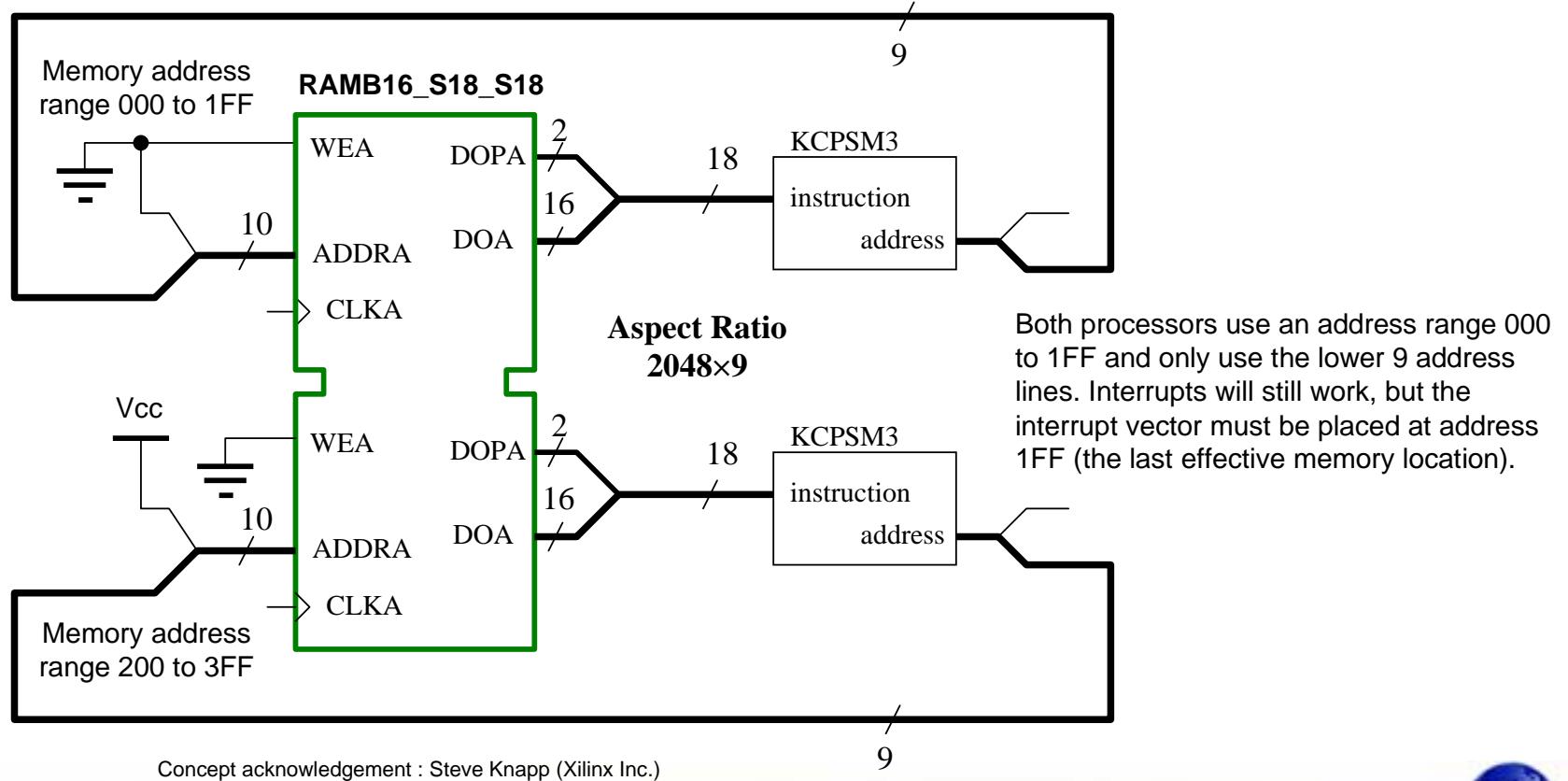
start: LOAD value, 1F          ;find sum of all values to 31
       LOAD total_low, 00      ;clear 16-bit total
       LOAD total_high, 00
       CALL sum_to_value        ;calculate sum of all numbers up to value
       OUTPUT total_high, 02    ;Result will be 496 (01F0 hex)
       OUTPUT total_low, 01
       JUMP start
;
;Subroutine called recursively
;
sum_to_value: ADD total_low, value      ;perform 16-bit addition
              ADCY total_high, 00
              SUB value, 01           ;reduce value by 1
              RETURN Z                ;finished if down to zero
              CALL sum_to_value        ;recursive call of subroutine
              RETURN                   ;definitely finished!
```

Increasing value to 20 (32 decimal) will result in incorrect operation of KCPSM3. The stack is a cyclic buffer, so the 'bottom' of the stack will be overwritten by the 'top' of the stack during the 32nd nested CALL instruction.

Sharing Program Space

For ease of design and possibly to meet system performance requirements, it is often desirable to use multiple KCPSM3 macros in the same device. Each KCPSM3 is designed to work with a single Block RAM which provides 1024 locations in the Spartan-3 and Virtex-II devices. For many control and state machine applications, this program size may be found to be excessive and lead to wasted block memory resources.

Since block RAM is dual port, it is quite possible to connect two KCPSM3 macros to the same block memory.....



Concept acknowledgement : Steve Knapp (Xilinx Inc.)

9

Design of Output Ports

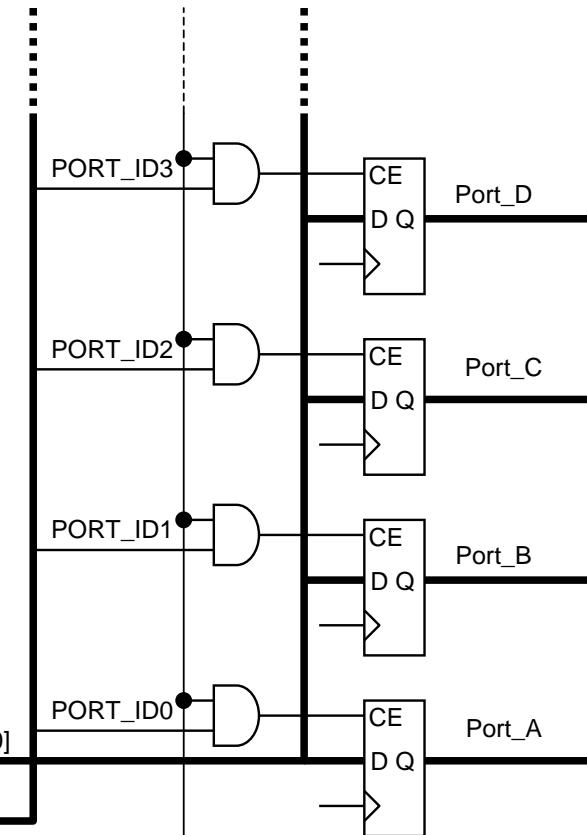
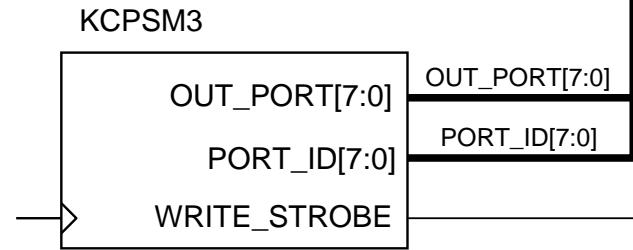
Being thoughtful about your interface circuit design will enable the logic to remain compact and performance to be maintained. The following diagrams show suitable circuits for output ports, input ports and connection of memory. If you are using a synthesis tool, it is advisable to check that your code is not describing a circuit which is more complex than is really required and that the synthesis tool is implementing the correct logic.

Simple Outputs

For 8 or less simple output ports try to assign ‘one-hot’ addresses and then make sure that your design only decodes the appropriate PORT_ID signal. This greatly reduces the logic for address decoding which is advantageous for lower cost and performance. It also reduces the loading on the PORT_ID bus which is often critical to overall system performance.

Use of CONSTANT directives in the program make the code readable and help ensure that the correct ports are used.

```
CONSTANT Port_A,01
CONSTANT Port_B,02
CONSTANT Port_C,04
CONSTANT Port_D,08
;
OUTPUT s0,Port_A
OUTPUT s1,Port_B
OUTPUT s2,Port_C
OUTPUT s4,Port_D
```



Note that all blocks share a common clock

Design of Output Ports

Fully Decoded Outputs and high performance

When there is a requirement to address blocks of memory and many simple ports, a large number of the 256 output port locations may be used requiring the PORT_ID addresses to be more fully decoded. If performance is critical, then careful design will again be advantageous.

The key observation is that during a write operation the PORT_ID and OUT_PORT are provided for 2 clock cycles with the WRITE_STROBE only active during the second of the two cycles (see read and write strobes). Although time specifications can be used to cover the 2-cycle paths, it is often easier to insert pipeline stages and split the address decoding effort as shown here.

Port Mapping

Dual Port (16 bytes) - 00 to 0F

Single Port (32 bytes) - 20 to 3F

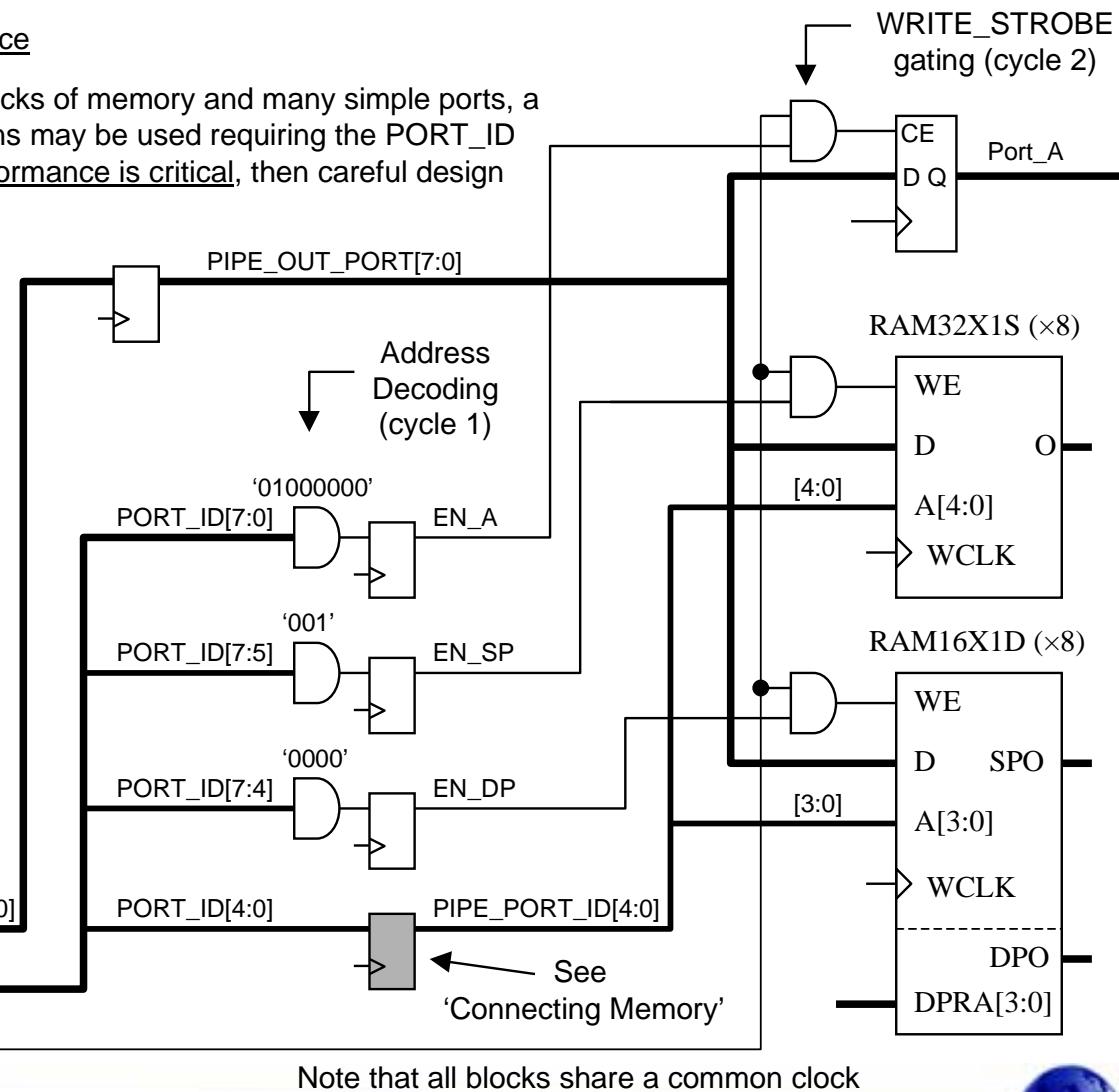
Port_A - 40

KCPSM3

OUT_PORT[7:0]

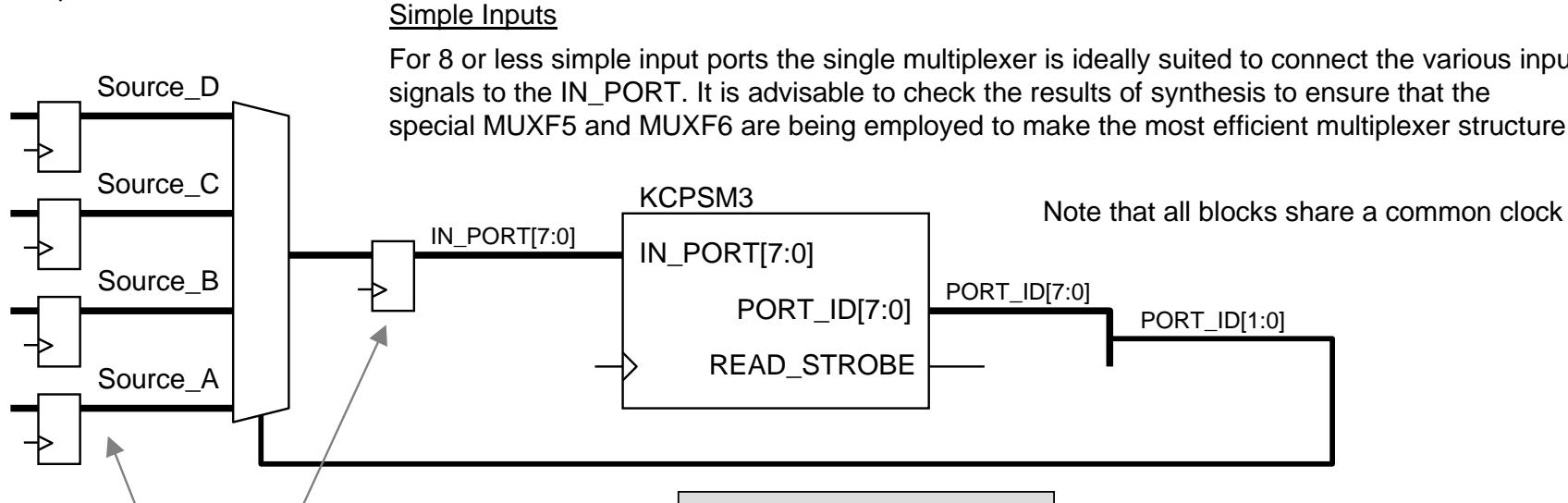
POR T_ID[7:0]

WRITE_STROBE



Design of Input Ports

The connection of input ports leads to the definition of a multiplexer. Obviously the size of this multiplexer is proportional to the number of inputs and having many inputs can lead to issues with performance unless care is taken with the description of this multiplexer structure.



Because the PORT_ID is valid for 2 clock cycles the multiplexer can be registered to maintain performance.

In the majority of cases, the actual clock cycle at which an input is read by the processor isn't critical. Therefore the paths from the sources can typically be registered such as using the I/O registers when coming from actual device pins. This will help simplify time specifications, avoid reports of 'false paths' and lead to reliable designs.

```
CONSTANT Source_A,00  
CONSTANT Source_B,01  
CONSTANT Source_C,02  
CONSTANT Source_D,03  
;  
INPUT s0,Source_A  
INPUT s1,Source_B  
INPUT s2,Source_C  
INPUT s3,Source_D
```

The multiplexer means that the best addresses to assign for input ports are normal binary encoding.

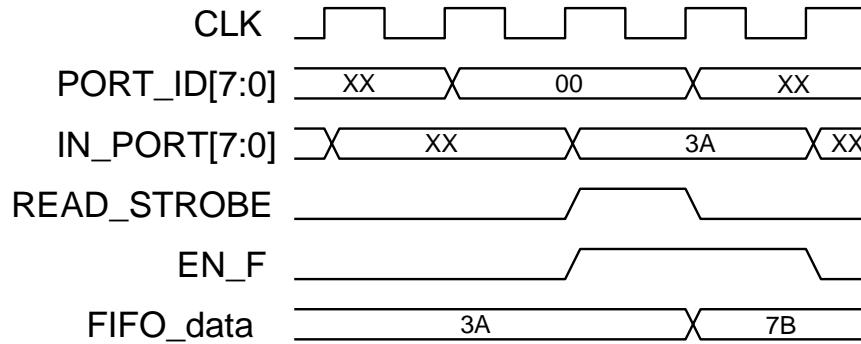
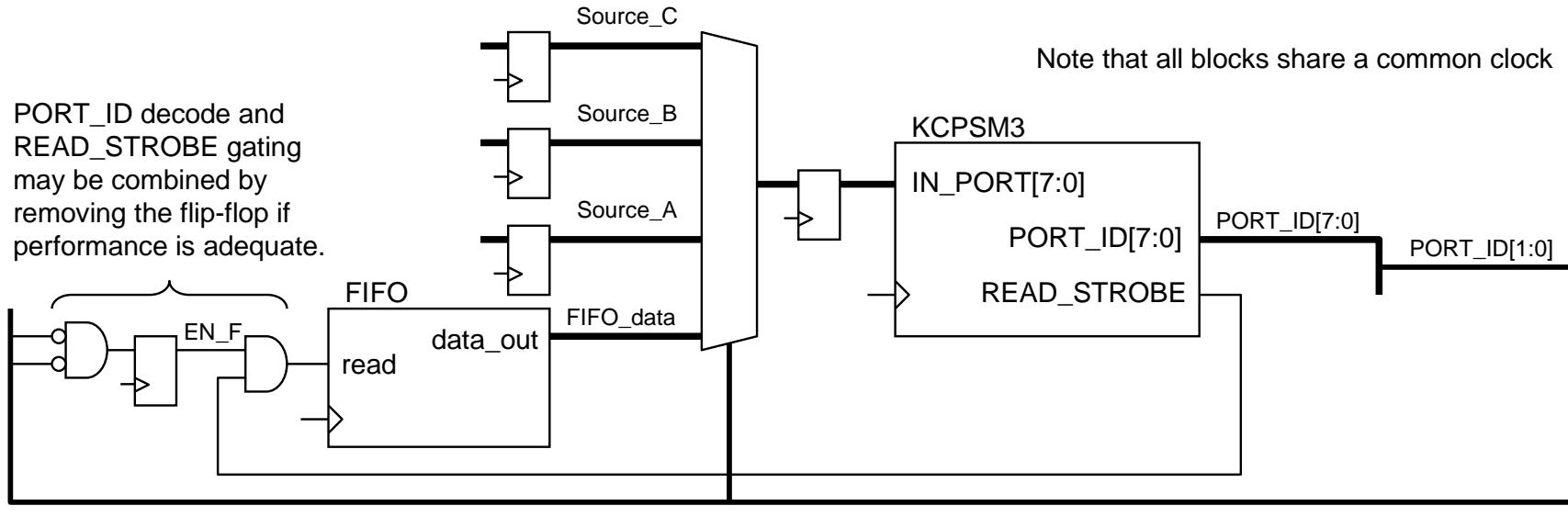
IMPORTANT

Failure to include a register anywhere in the path from PORT_ID to IN_PORT is the most common reason for observing significantly lower clock rates than indicated in the 'Size and Performance' section of this manual. So make sure you have one!



Design of Input Ports

Occasionally it will be important that a circuit providing data to KCPSM3 to know that it has been read. The obvious example is a FIFO buffer which will then prepare the next data to be read.

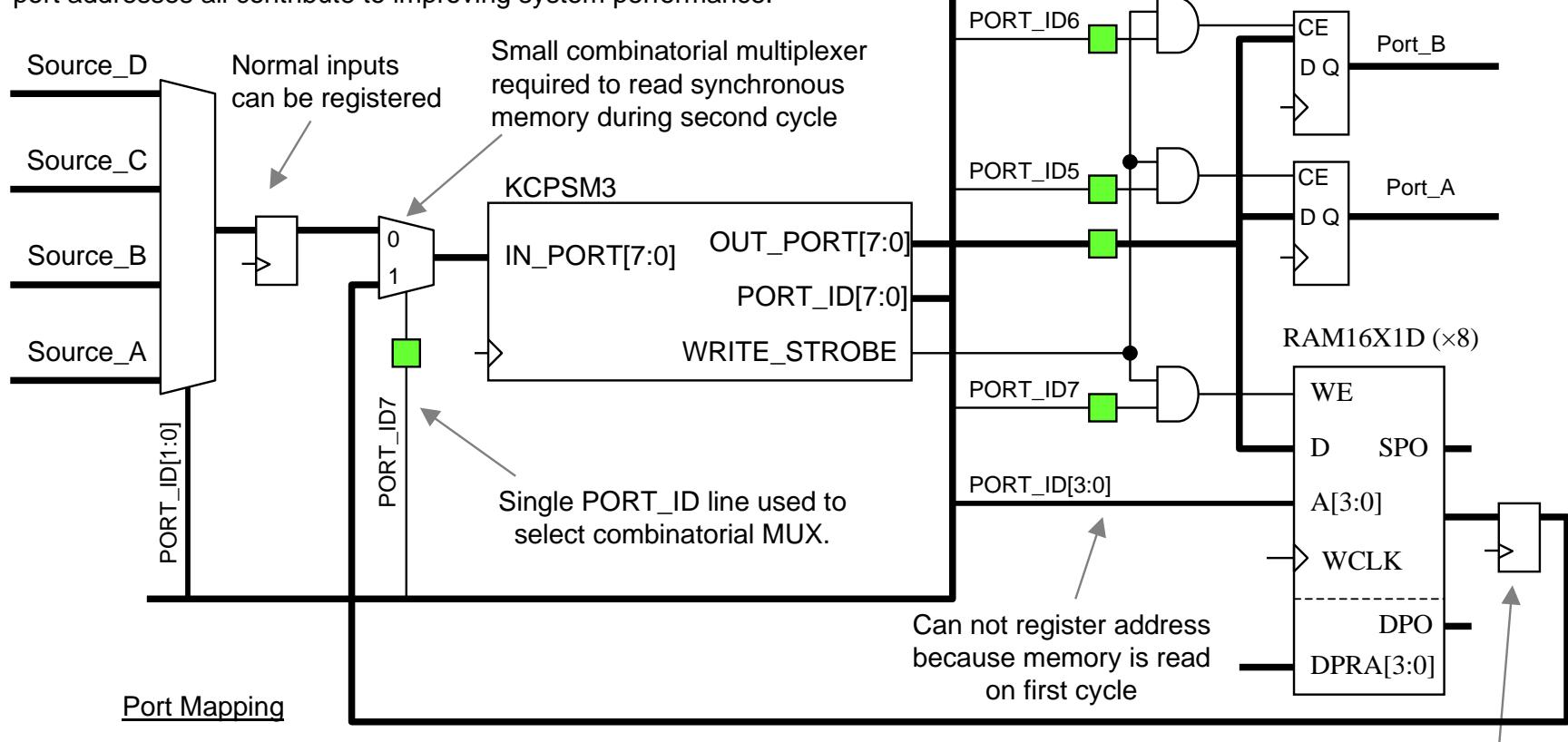


The data path from the FIFO is quite separate to the read acknowledgement circuit.

In this example the FIFO is assigned the address '00'. Initially the FIFO is providing data with the value '3A'. The act of reading the port causes the FIFO to provide the next data of value '7B'.

Connecting Memory

The connection of memory (Dual port is ideal for communication with other modules) is the most common cause for reduction in system performance. Observing where pipeline registers can be inserted, splitting the input multiplexer and careful allocation of port addresses all contribute to improving system performance.



■ = Additional places to insert flip-flops if really necessary for performance.

Simulation of KCPSM3

KCPSM3 is supplied as a VHDL macro together with an assembler. No tools are currently supplied for the direct simulation of code. However, this immediate lack of simulation tools does not appear to have deterred many thousands of Engineers from using PicoBlaze macros over the past few years. Common reasons for this acceptance of this situation are:-

Interaction with hardware

It is very common for PicoBlaze to be highly interactive with the hardware in which it is embedded. With virtually continuous interaction between the processor and the input and output ports, it would be difficult to simulate these interactions in a purely software isolated environment. In a similar way, the simulation of the hardware design requires the stimulus from the processor. So in many cases, the simulation of the processor will become part of the hardware simulation using a tool such as ModelSim. The following pages illustrate how the KCPSM3 macro can be used directly in a VHDL simulation and describes some features within the coding of the macro which enhance the simulation of the PSM software execution as well as the I/O ports.

It would all be too slow!

Hardware is very fast in that it can work every clock cycle. Hardware simulators are required to display results in pico-seconds and nano-seconds. In contrast, PicoBlaze is often employed in operations which are less time critical or deliberately slow in comparison. For example, a real time clock is impractical to simulate using a hardware simulator or a software simulator and UART based communication, even at high baud rates, is desperately slow relative to a 50MHz clock.

The solution in these cases is quite simply to use the hardware directly as the testing and debugging medium. It is quite possible to recompile a small design in less than a minute to make iterative changes to code and hardware. The key to success is to start with very simple experiments and only make small changes and additions each iteration. The dual port block RAM can be exploited to provide a development platform with a rapid way to download new programs. One method is to use the user port on the JTAG controller and a reference design for this is described by Kris Chaplin in his Tech Xclusive article which can be found at.. <http://www.xilinx.com/support/techxclusives/techX-home.htm>

Other tools are available

Some Engineers that have used PicoBlaze over the years have been busy writing their own development tools. One company, Mediatronix, has been kind enough to make a full PicoBlaze Integrated Design Environment (IDE) available to other designers at no charge simply by downloading it from the 'tools' section of their web site..... <http://www.meditronix.com>
Many thanks to Mediatronix!



VHDL Simulation

The 'kcpsm3.vhd' file is written in a style which is suitable for simulation as well synthesis. The default template used in the generation of the program ROM VHDL file also includes the necessary definition of a block RAM for simulation. Therefore no special steps need to be taken to simulate KCPSM3 as part of your design or in a smaller test case.

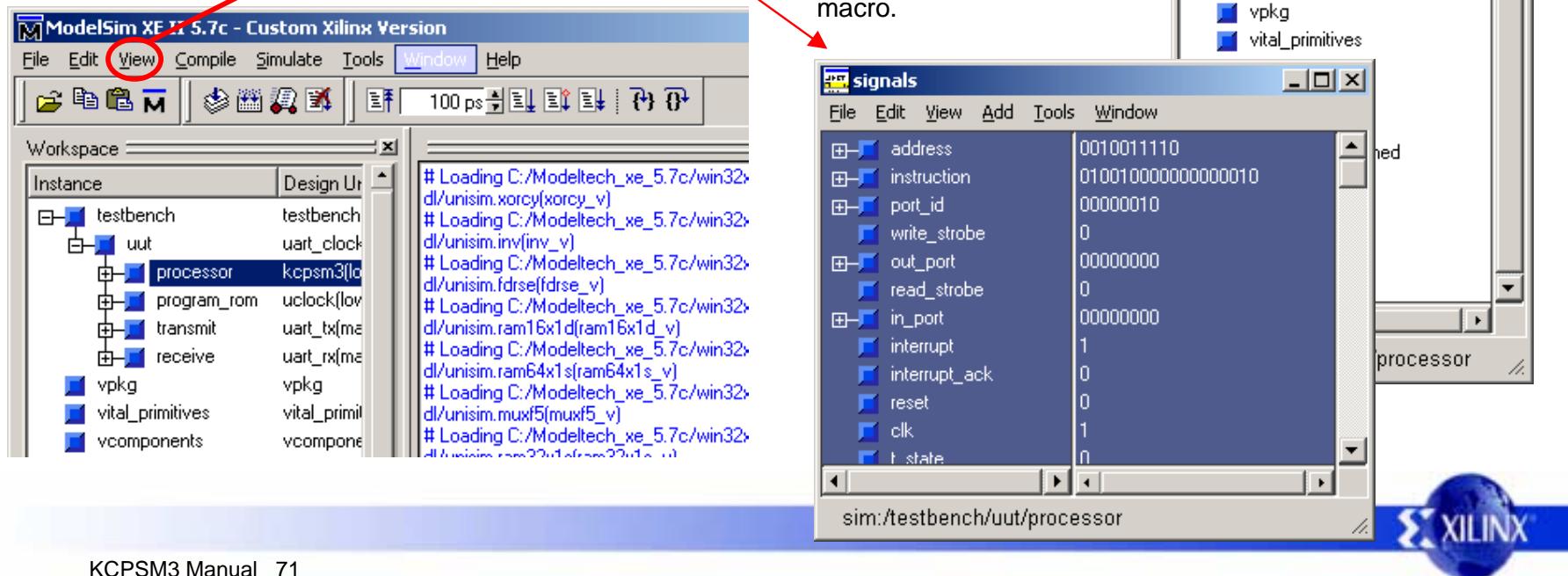
Signals

All of the signals forming connections to and from the KCPSM3 macro and the program ROM should be available to you via your simulator. ModelSim makes signals available as illustrated below.

View allows you to display further windows although some may open automatically.

- wave
- structure
- signals
- variables
- process

structure enables you to identify the processor in your design and **signals** will then display a list of all the signals within the processor macro.



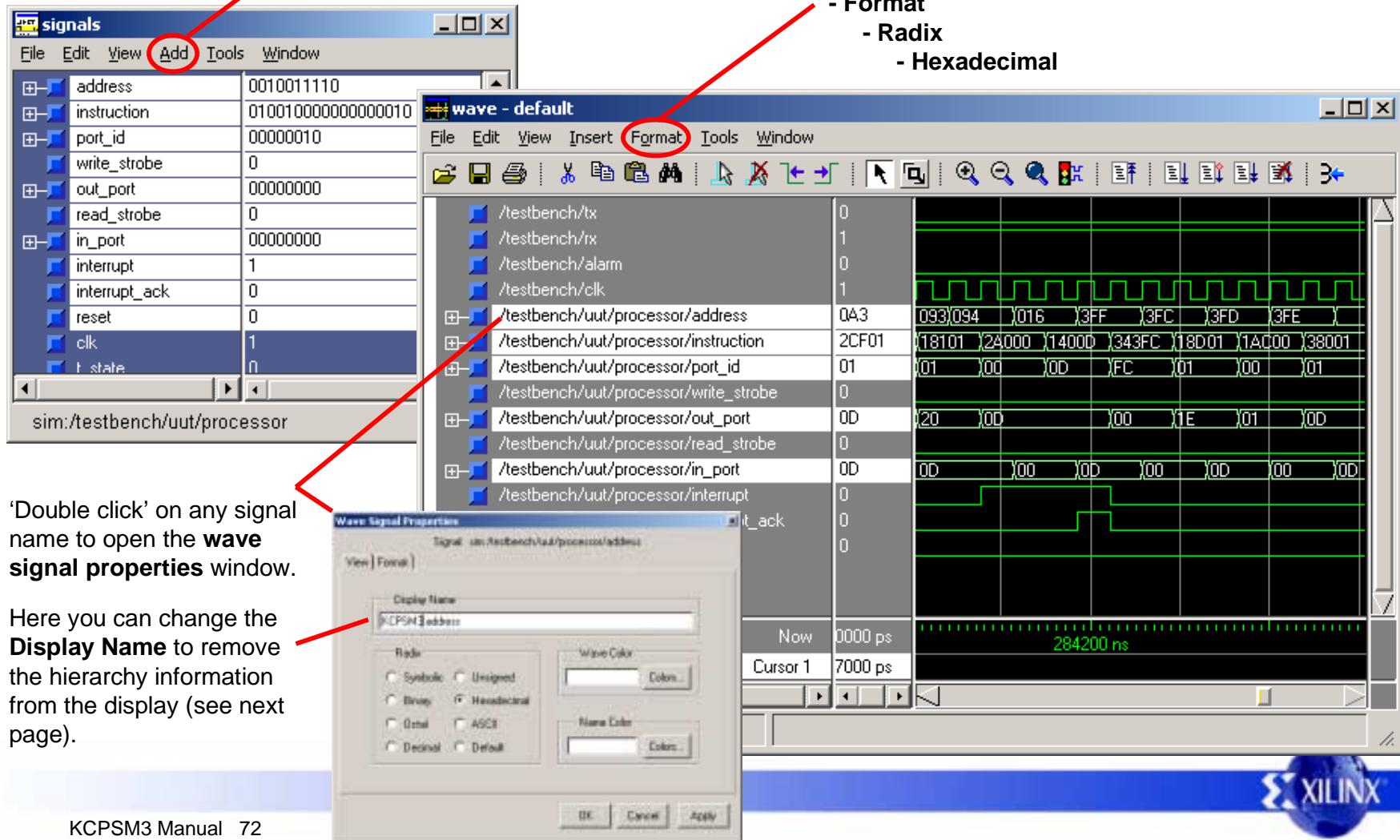
VHDL Simulation

Select the signals you require and then via the **Add** menu include them in the wave display.

- Add
- Wave
- Selected Signals

The **Format** menu allows you to display the bus signals in hexadecimal which relates directly the information in the assembler '.log file'.

- Format
- Radix
- Hexadecimal



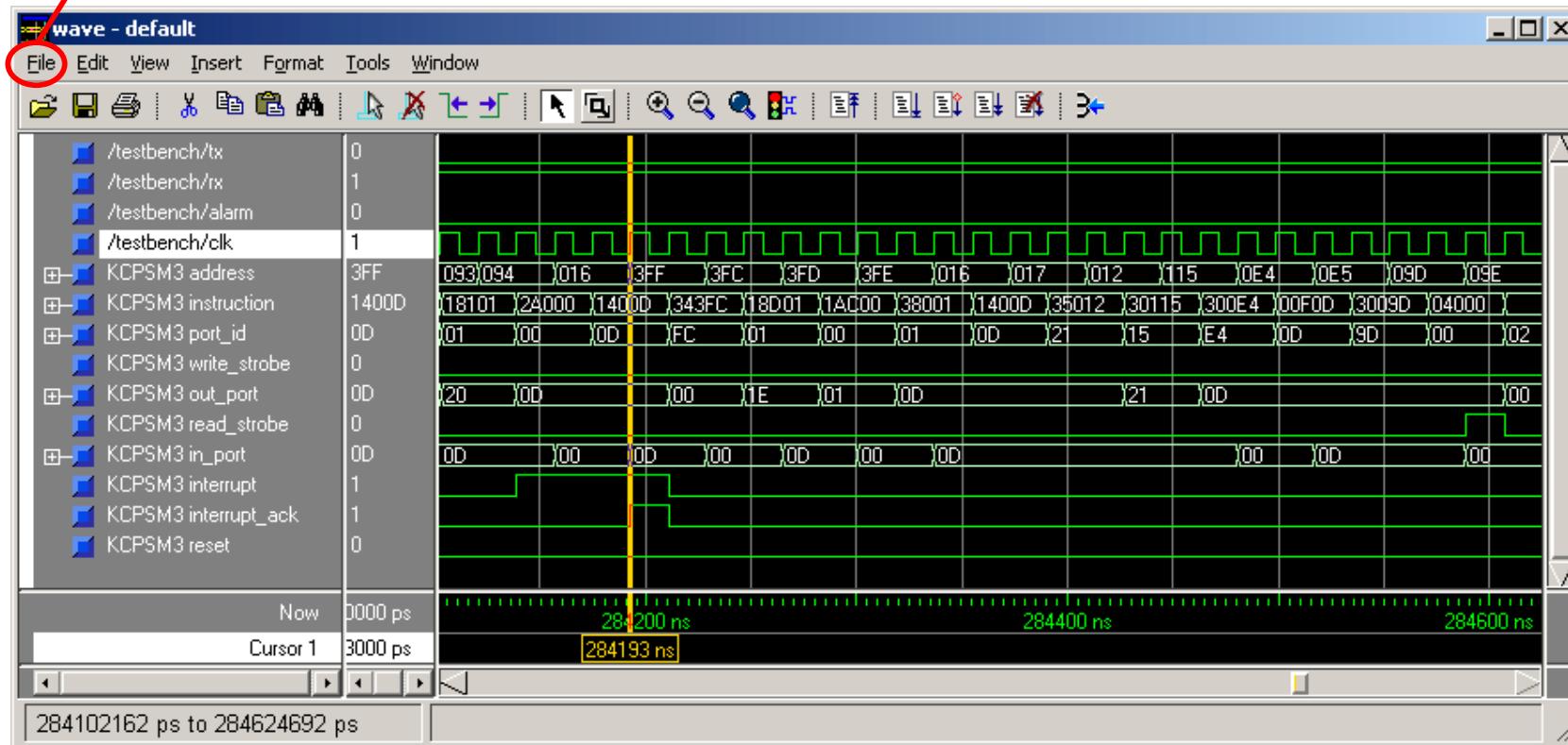
'Double click' on any signal name to open the **wave signal properties** window.

Here you can change the **Display Name** to remove the hierarchy information from the display (see next page).

VHDL Simulation

- File
- Save Format

Once you have the wave display the way you like it, don't forget to save it as a '.do' File. Use **Load Format** to read in your wave format in another session.



The '.do' file is a simple text file. Once you can see the format of the commands, it may be easier to add and format other signals by directly editing this file.

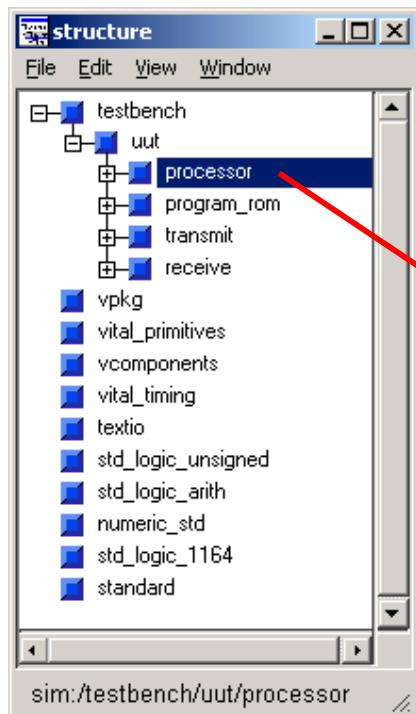
```
add wave -noupdate -format Literal -label {KCPSM3 address} -radix hexadecimal /testbench/uut/processor/address
```



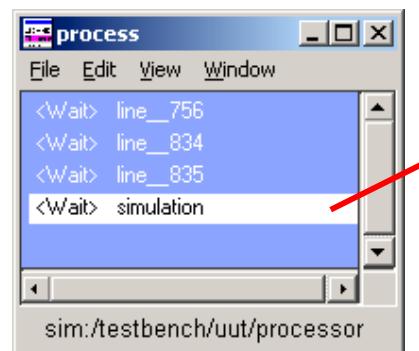
VHDL Simulation

Although it is possible to simulate a KCPSM3 design purely by reference to the signals, the actual operation of the program is difficult to follow. For this reason, the 'kcpsm3.vhd' includes a process called 'simulation' specifically to enhance simulation. The useful output of this code is in the form of variables.

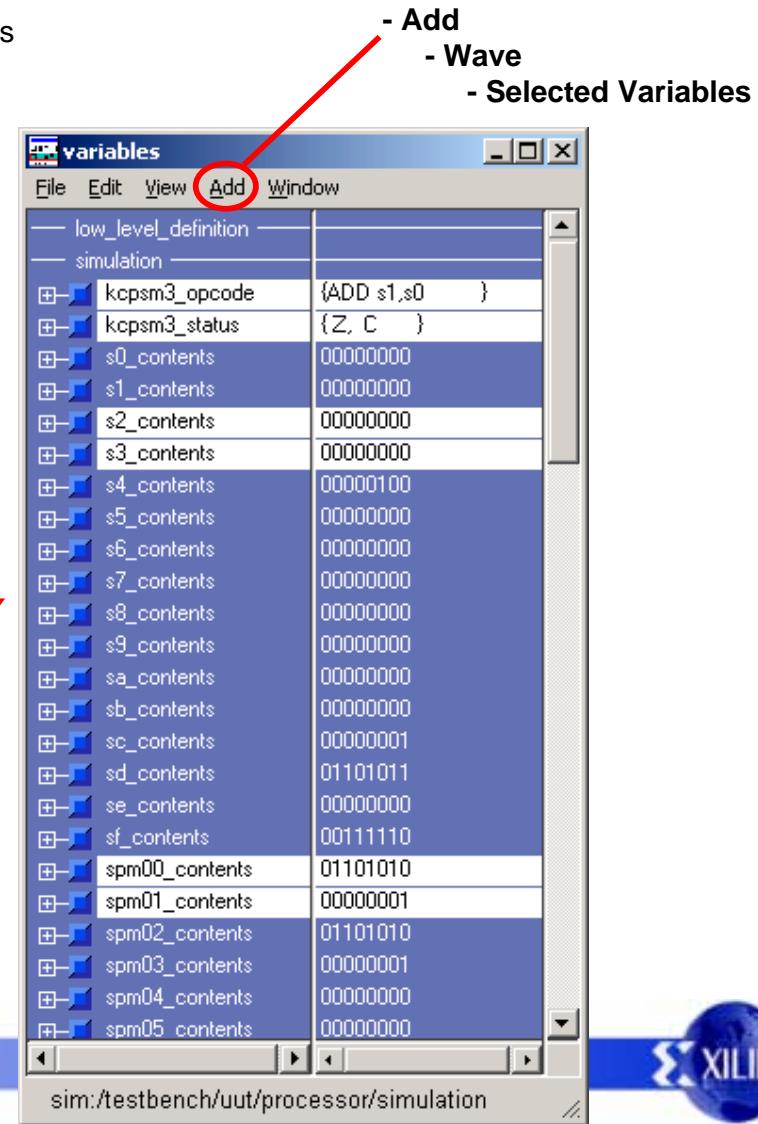
Variables



As with selecting signals, first use **structure** to identify the processor in your design. Then use **process** to select the process which has been called 'simulation'.



Select the variables you require.



Concept acknowledgement : Prof. Dr.-Ing. Bernhard Lang.
University of Applied Sciences,
Osnabrueck, Germany.

VHDL Simulation

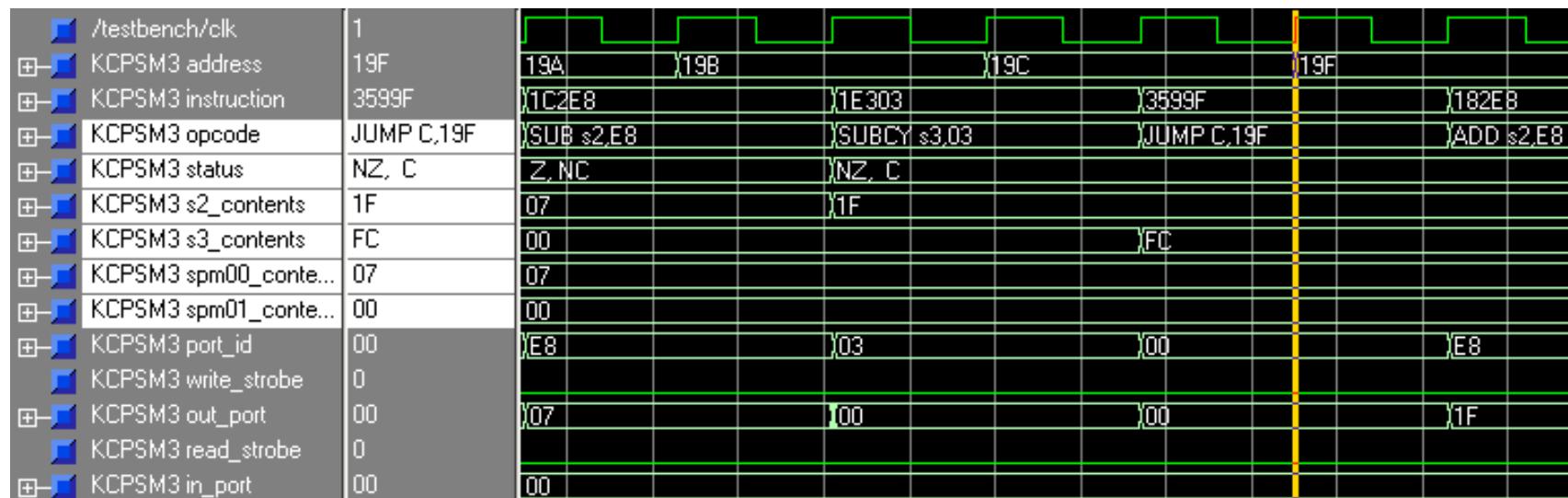
Variable names

kcpsm3_opcode - Represents the current instruction as a text string which makes the code execution very easy to follow. For example, the snap shot below has decoded '3599F' as the instruction 'JUMP C,19F'.

kcpsm3_status - The status of the ZERO and CARRY flags are represented as text. The status 'NZ, C' is displayed at the time of the 'JUMP C,19F' in the snap shot below indicates that the condition has been met. The status will also include 'Reset' when the 'internal_reset' is active (see page 39).

s0_contents through to **sf_contents** - These provide a 'std_logic_vector' representing the contents of each register. The snap shot shows the contents of 's2' and 's3' being updated by the associated 'SUB' and 'SUBCY' instructions.

spm00_contents through to **spm3f_contents** - These provide a 'std_logic_vector' representing the contents of each location of the scratch pad memory.



Once the variables are included in the wave display they can be formatted (name and radix) in just the same way as signals and the format saved in your '.do' file.





PacoBlaze

Version 2006-03-14

What is PacoBlaze and KCasm?

PacoBlaze is a from-scratch synthesizable & behavioral Verilog clone of Ken Chapman's popular PicoBlaze embedded microcontroller. KCasm is a lightweight PicoBlaze assembler written in Java.

Why does PacoBlaze exist?

While Ken's version aims toward the most efficient implementation in the Xilinx FPGA architecture, PacoBlaze tries to be as device-independent as possible maintaining source code compatibility and code cycle accuracy with the original PicoBlaze. Moreover, as there are actually 3 versions of the PicoBlaze microcontroller, PacoBlaze's final achievement is to provide all PicoBlaze versions in one configurable Verilog file set.

The nice thing™ of having a behavioral Verilog model of PicoBlaze is that it is easier to modify, trim or expand the core in order to adapt it for special purpose applications (e.g. PacoBlaze mods with a multiply or bit count instructions). It also makes pure behavioral Verilog simulations possible; something really neat that has allowed many of its users to reach Nirvana. ☺

Who wrote it and why?

Pablo Bleyer Kocik, [yours truly](#). I started working in PacoBlaze because I was in need of a small embedded controller for my FPGA-based projects and, even I was a big fan of the original PicoBlaze, its lack of configurability was restraining my creativity... No, really. ☺

PacoBlaze has been written for Verilog-2001 compliance and is tested with Stephen Williams' [Icarus Verilog](#) compiler and Pragmatic C's [GPL Cver](#) compiler. Xilinx [ISE WebPACK](#) is used for testing core synthesis, including module inference. KCasm is compiled and tested using [Sun's J2SE JDK](#) and utilizes the [JavaCC](#) Java compiler-compiler.

What is PacoBlaze's license?

PacoBlaze and KCasm are released under the BSD License for maximum flexibility and usefulness. This means you can use the project however you wish, without compromising your whole project and liberating me from any responsibilities (so both of us can have peaceful dreams). A copy of the license is included at the end of this document. Although you are not legally bound, it is considered morally and socially acceptable if you send me back bug reports.

Files

File name	Description
Makefile	Makefile with rules to build the simulations for the distribution.
pacoblaze_inc.v	Main include file for definitions and macros.
pacoblaze.v	Main PacoBlaze module file, parametrized with macros.
pacoblaze_idu.v	Instruction Decode Unit implementation.
pacoblaze_register.v	Register file implementation.
pacoblaze_stack.v	Address stack implementation.
pacoblaze_scratch.v	Scratchpad implementation.
pacoblaze_util.v	Code utilities.
pacoblaze{1,2,3}.v	PacoBlaze 1, 2 or 3 module definition. You should use this for each PacoBlaze instantiation.
pacoblaze{1,2,3}_xst.v	Module instantiation with relative paths that I use for XST synthesis (since XST doesn't currently allow specifying Verilog include directories. Arrrgh!).
blockram.v	Behavioral BlockRAM description for synthesis.

Other files

File name	Description
pacoblaze_dregister.v	Dual register file implementation (even/odd and dual register accesses).
X_tb.v	Testbenches.

Macros

Macro name	Description
PACOBLAZE{1,2,3}	Define this before including pacoblaze_inc.v to specify which PacoBlaze configuration to use.
HAS_RESET_LATCH	When defined, the reset signal drives an internal reset to avoid reset glitches.
USE_ONEHOT_ENCODING	Use one-hot encoding for opcode decoding.
HAS_INTERRUPT_ACK	Export interrupt acknowledge signal in the top module.
HAS_SCRATCH_MEMORY	Enable resources for scratchpad memory implementation.
HAS_COMPARE_OPERATION	Enable the "compare" operation.
HAS_TEST_OPERATION	Enable the "test" operation.
HAS_DEBUG	When defined will export debugging signals.

Extensions

Macro name	Description
HAS_MUL_OPERATION	Enable the 8x8 multiply operation.

Using KCAsm

The command line to assembly the KCPSM3-psm file *my_assembler_file.psm* and create the *my_module_name.v* 18-bit BlockRAM Verilog configuration file together with the *my_listing_file.rmh* Verilog-readmemh listing file, is:

```
java -Dkcpasm=3 -Dboram=18 -Dmodule=my_module_name -jar KCAsm.jar
my_assembler_file.psm my_listing_file.rmh
```

PicoBlaze resources

- [PicoBlaze @ Xilinx](#)
- [PicoBlaze Product Brief](#)
- [PicoBlaze User Resources](#)
- [TechXclusives: Creating Embedded Microcontrollers \(Programmable State Machines\) by Ken Chapman \[PDF\]](#)
- [Xilinx UG129 PicoBlaze 8-bit Embedded Microcontroller for Spartan-3, Virtex-II, and Virtex-II Pro FPGAs](#)
- [FPGA IFF Copy Protection Using Dallas Semiconductor/Maxim DS2432 Secure EEPROMs](#)
- [PicoBlaze 8-Bit Microcontroller for Virtex-II Series Devices](#)
- [Reducing the Size of SD-SDI EDH Processing Using the PicoBlaze Processor](#)
- [Multi-Rate SDI Integration Examples for the Serial Digital Video Demonstration Board](#)
- [Embedded Processing and Control Solutions for Spartan-3 FPGAs](#)
- [On the Fly Reconfiguration with CoolRunner-II CPLDs](#)
- [PicoBlaze 8-Bit Microcontroller for CPLD Devices \[ZIP\]](#)
- [CryptoBlaze: 8-Bit Security Microcontroller \[ZIP\]](#)
- [PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices](#)
- [DVI, VGA, and Component Video Demonstration](#)
- [pBlaze IDE - an Integrated Development Environment dedicated to the KCPSM soft core](#)
- [kpicosim - a development environment for the Xilinx PicoBlaze-3 soft-core processor for the KDE Desktop](#)
- [Enterpoint's Picoblaze Enhancements](#)

Contributors

Here are some recent kind users which have helped identifying bugs and petted PacoBlaze with affection.

- Allan Herriman
- Ed Blanchard

Please contact me if I have forgotten you and you wish to be added to the list.

BSD License (aka 'Modified' BSD License)

Modified BSD License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



PicoBlaze

Creating Embedded Microcontrollers (Programmable State Machines) - Part 1

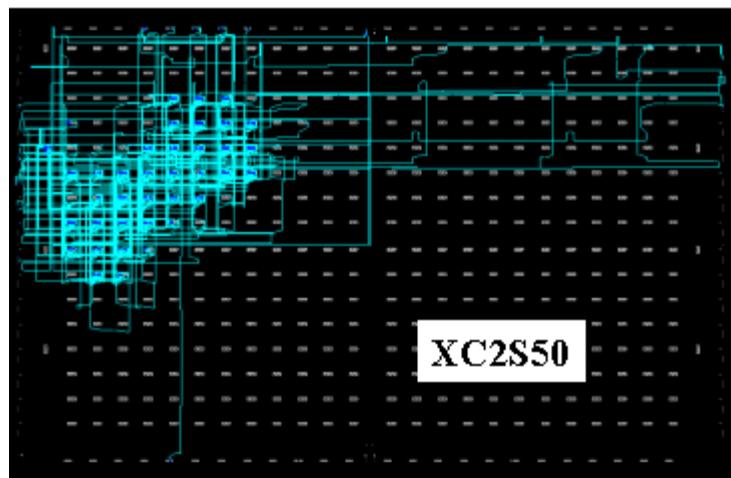
Introduction

It may seem strange that, as a Xilinx Applications Engineer, I am such an advocate of microprocessors. However, as my previous articles regarding "Performance + Time = Memory" explained, microprocessors really are a very efficient use of silicon. Although specific hardware implementations using time sharing techniques can be formulated, a microprocessor offers superior levels of flexibility via the software programming methodology, providing the "Time" factor is adequate. This is particularly useful for applications in which the processing to be implemented is "esoteric" in nature and would require very complex state machines.

With the release of the MicroBlaze™ RISC processor soft core last year, and the more recent introduction of the Virtex-II Pro™ devices with their embedded PowerPC 405 hard cores, you can imagine how much potential I can see for future applications with Xilinx devices. However, I have not merely waited for these cores to become available, but have been implementing my own processor macros inside Xilinx devices since 1993. From the publications on the subject, I also know that I have not been the only one exploiting Xilinx devices in this way.

My particular interest is in the creation of very small processor macros. These have much more in common with the world of microcontrollers than full-blown data processors, which are larger and more powerful. My focus is to bring the most significant advantages that a processor can offer to a design environment at minimum cost. For this reason, I have considered these small processors to be "Programmable State Machines" and refer to them as "PSM".

By exploiting the Xilinx device architecture, it has been possible to create processors such as the KCPSM, which occupies just 35 CLBs in a Spartan-II™. The plot below shows a single KCPSM in an XC2S50 Spartan-II device. You can actually fit 8 PSM processors in this device and still have 30% of the device remaining for hardware circuits. Even the smallest Spartan-II (2S40) can support 2 of these PSMs.



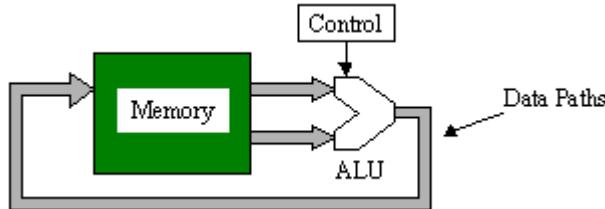
I know that many of you have enjoyed using this macro and I never cease to be amazed at the applications you find for it. For many applications, a single PSM combined with hardware circuits has been more than adequate. For applications requiring more processing power, the ability to use multiple KCPSMs in a single Xilinx device has the added advantage of distributing the processing and keeping the interfaces-to-hardware circuits simple and independent. The introduction of MicroBlaze and PowerPC cores has only increased the usage of PSM macros. These tiny processors can handle tasks independently to the main processor and with much tighter and predictable coupling to the hardware circuits.

In this *techXclusive* mini-series, I will explain how I derived the architecture of KCPSM and how it exploits the

architecture of the Xilinx FPGA devices. I hope you will be inspired to create your own application-specific PSM processors as well as find new applications for existing PSM macros.

How many bits?

How many bits should a processor have? The nice thing about an FPGA is that you can decide what is most suitable for your applications and implement the data width that you require. In general, the wider the data width, the more logic you will need to support it, and the more capable the processor will become. Conversely, the narrower the data width, the smaller the implementation, but the lower the arithmetic performance. My primary objective was a small size and low cost, so this meant a narrow data width and acceptance of the accompanying lower arithmetic performance. As the processor is surrounded by programmable hardware that offers the ultimate in high performance, the processor was intended to be a complex programmable state machine and not a DSP processor.



Even so, I found it very hard to decide on a bit width, as I did not have one specific application in mind. So when I designed my first PSM in 1993 using XBLOX™ (Anyone remember that one? It was fundamentally a schematic synthesis tool), I avoided the issue by creating a processor in which the data width was defined only at the point of synthesis. It turned out to be quite difficult to construct a processor around this decision, and this lead to many other restrictions, especially when combined with the limits of the XC4000™ devices then available. However, it proved to me that efficient PSM processors could be made, and I started to gather feedback from the most important people -- our customers. Of the people that used this first PSM, nearly all defined an 8-bit data width. It was a simple case of conformance!

Given the choice of conforming to a standard of 8, 16, or 32 bits, the future PSM processors just had to be 8 bits. Not exactly a sound engineering decision, but I learnt never to fight with nature! However, I would ask you to consider "non-standard" bus widths if it fits with your needs.

Embedded Program

A PSM, like any other processor, will execute a program. A program is formed by a set of instructions that are defined by the user and held in a memory. Each instruction is encoded into a machine code. That much is obvious, but where should that program be stored?

If you use a standard processor, it may be natural to think of the program being stored in a ROM or RAM device, or even on a floppy disk. But if you were implementing a state machine as part of your Xilinx FPGA design, you would not connect any external components. It was clear to me that a PSM must be 100% embedded in the device and totally self-sufficient. In this way, you can make the decision to use one or more PSMs anywhere it makes sense to do so without concern for the design of the PCB on which the FPGA is sitting. This meant that the program had to be implemented inside the device.

Unfortunately the XC4000 family did not contain any dedicated memory, so the CLBs had to be converted to program ROM. However efficient the processor, this ROM was expensive, so programs had to be small and very efficient. Imagine my joy at seeing block RAM appear in Virtex™! Here were 4096 bits of memory just asking to hold a PSM program.

In Peter Alfke's *techXclusive* "Using Leftover Multipliers and Block RAM in Your Design", we can see that a block RAM can be used to implement a state machine. This is similar in many ways to the way a PSM works, but it does not exploit the dimension of time. The result is a relatively simple state machine that operates very quickly. With a PSM, we can achieve very complex state machines that work relatively slowly.

Block RAMs are very flexible in aspect ratio and are initialisable. It would again be very awkward to have to go

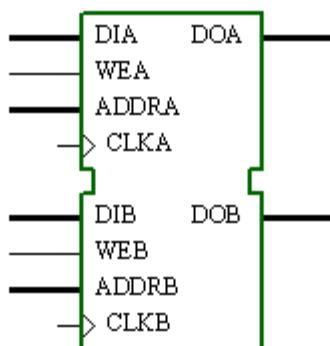
through a program memory-booting phase before the state machine could become active. However, since all 4096 bits of each block RAM are defined in the configuration bit stream of the device, the PSM is able to operate from the very first clock edge.

So, the block RAM of Virtex (which then became the basis of Spartan-II devices) was to be the program ROM. This defined the second major limit for the PSM in that the size of program would be limited to that supported by one block RAM.

Block RAM Aspect Ratios

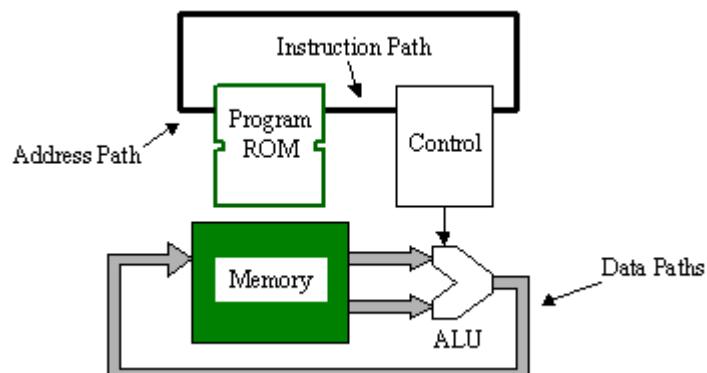
Locations	Instruction Width	Address Bits
4096	1	12
2048	2	11
1024	4	10
512	8	9
256	16	8

Block RAM



Embedded Program Advantage

Embedding the program ROM inside the FPGA has the advantage that all of the inputs and outputs of both the program ROM and the processor are "virtual pins" within the FPGA fabric. This means that there is no need to consider bus sharing to reduce the number of pins and busses. It is also obvious that a Harvard architecture is the natural selection. All the requirements for multiplexed data and instructions that would waste time, rather than exploit it, are avoided.



Given that there is no real restriction on the number of virtual pins the PSM can have, it is possible to explore all of the aspect ratios supported by the block RAM. At first, it may appear reasonable to adopt 8-bit instructions to be the same as the data path. This would make sense in a shared memory and bus system, but here we do not need to be constrained in that way.

By having a narrow instruction width, we have the advantage of more program memory locations, but a greater need to encode instructions. It will also be necessary to have fetch cycles to obtain operands when required. This will increase the amount of decoding and sequencing logic and lower performance due to the number of cycles per instruction. As multiple locations will be needed for many instructions, having more memory locations does not actually imply that the program can be longer.

Opting for a wider instruction path means that less decoding logic is required, and there is the potential for each instruction to be completely self-contained in terms of operation and operands obviating the need for fetch cycles. Given the desire to keep the PSM small, it was therefore decided that the 256×16 aspect would be most suitable. Admittedly, a program of only 256 instructions was going to be a constraint, but the intention was to implement a complex programmable state machine rather than full-blown data processor.

In Part 2

In the next article I will consider the relative merits of register, stack, and accumulator processor architectures when a PSM is implemented in a Xilinx device.

If you want to have a look at the KCPSM, it is available for download at the address below. (There is also full documentation and an assembler for you to use.)

Xilinx Application Note 213: "8-bit Microcontroller for Virtex Devices"

<http://www.xilinx.com/xapp/xapp213.pdf>

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

This is suitable for all Virtex, Virtex-E and Spartan-II devices. If you would like a PSM specially tuned to the Virtex-II architecture, drop me an e-mail at ken.chapman@xilinx.com and I will be pleased to send it to you.

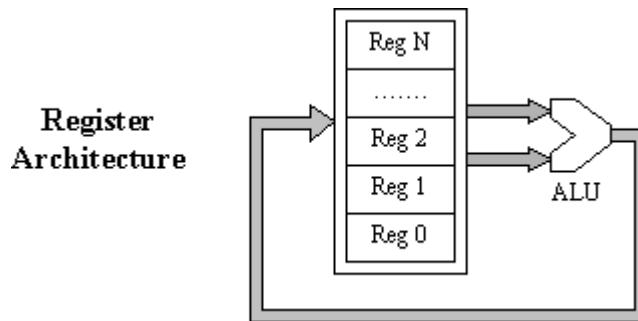
Creating Embedded Microcontrollers (Programmable State Machines) - Part 2

In the first part of this series, I examined the concept of a programmable state machine (PSM). The idea of this is to offer the ability to define a software-programmable complex state machine and exploit the "time domain" available to achieve a function in a small, cost-effective form. From this concept, and with some consideration for block RAM, we know that the PSM being studied will have an 8-bit data path, along with completely separate address and instruction paths to a program that is stored in a 256×16 aspect program ROM.

Before starting on the details of implementation, we are required to choose a processing architecture.

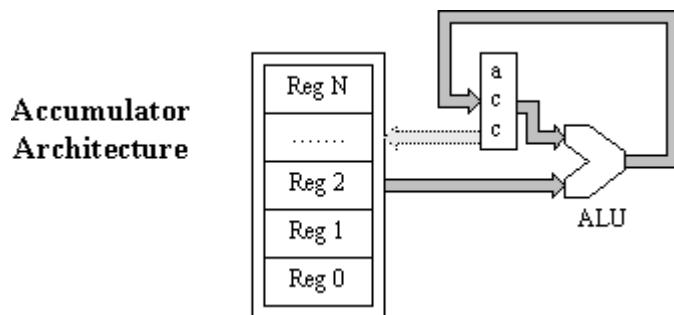
Register, Accumulator, or Stack?

There are three fundamental architectures in which the data memory and Arithmetic Logic Unit (ALU) can be organised. Hybrids of these also exist, but we will focus on the basic forms to make a selection that is most suitable for a PSM.



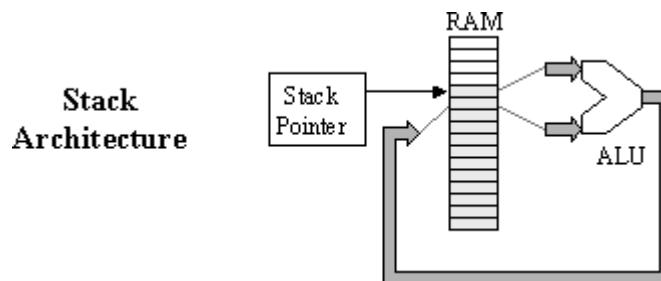
The **register architecture** has a finite number of registers that the program may select in any order. This tends to make manual programming (i.e., writing at assembler level) a straightforward task. It is desirable to have a large number of registers to perform complex tasks and hold multiple variables locally. Obviously, the more registers there are, the larger the implementation of a processor due to the number of storage elements and the multiplexer logic that is required to select the operands applied to the ALU.

Equally significant is the impact a register-based architecture has on instruction size. Some bits must be used to specify each register used in an instruction. For example, a 3-bit binary code is required to select one of 8 registers. Therefore, 6-bits would be required to specify the two operand registers. A further 3-bits may be required to specify a destination register for the result.



The **accumulator structure** is almost certainly associated with registers or memory to hold the various variables. The advantage of the accumulator structure is that one of the operands and the destination for the result is implied and does not need to be specified in the instruction encoding. Hence, the instruction encoding only needs to reserve bits to identify the remaining operand, and less logic is required to select the register or alternative source.

The disadvantage of this structure is that a program will tend to expend instructions and time simply moving an initial value into the accumulator or storing the accumulated value. Whilst long sequences involving the current accumulated value will be efficient, more esoteric programs, such as those implementing complex state machines, will become tedious to write and be slow to execute as the accumulator is continuously initialised and stored.



The **stack architecture** is probably the most efficient in terms of silicon resources, as it links directly to memory that is forming the stack and requires no data selection logic other than the stack pointer. The instruction encoding is also very efficient because the location of both operands and the result are implied as being the top of the stack.

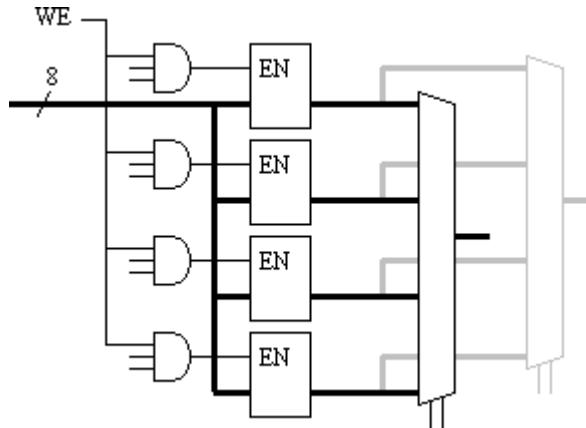
However, even more instructions (stack PUSH and stack POP) are expended to ensure that the correct data is located at the top of the stack. Correct sequencing of the ALU operations will result in excellent code density and execution speed, but this "reverse polish" style does not come naturally to most of us and greatly impacts the desire

to utilize a PSM that provides an easy methodology for implementing complex state machines.

Selecting the Register Structure

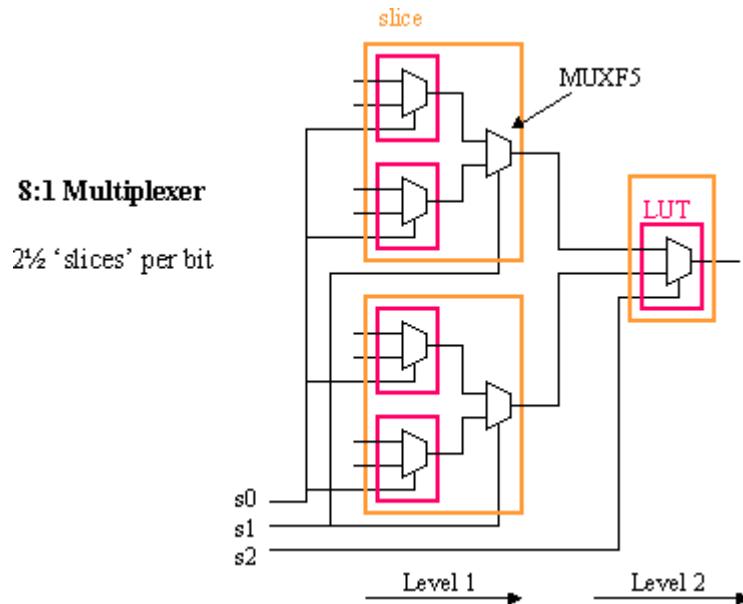
- The stack architecture is not the best choice, as the methodology is too cumbersome.
- The accumulator structure seems reasonable, but expending too many instructions to move data is a concern given limited program memory space.
- The register architecture appears to be the best for PSM applications, but it could be expensive.

The cost of implementing a register bank can be high, as the following diagram illustrates. Only four registers are being implemented; there are 32 flip-flops for 8-bit data, which would occupy 16 "slices" of a Spartan-II™ or Virtex™ device.



We must now add the operand selecting multiplexers and clock-enable gates to the registers. A 4-to-1 multiplexer would require a complete slice per bit by combining the two LUTs and the dedicated MUXF5. Hence the 8-bit multiplexer requires 8 slices.

To fetch both operands at the same time, a second multiplexer is required. This 16-slices of logic is free if the combinatorial logic is mapped into the same slices as the flip-flops. However, placing combinatorial logic between the registers and the ALU would compromise the performance of the processor. The clock-enable gates only require 2 slices, but again would add combinatorial delay.



Increasing the number of registers does not seem to be a good idea! The table below illustrates the number of slices required by a Spartan-II to implement a selection of register bank sizes. Whilst it is obvious that more flip-flops are required, the multiplexer logic dominates the size. The multiplexers for 8 and 16 registers also incur another level of logic delay as indicated by the 8:1 multiplexer above. Virtex-II has dedicated MUXF6 and MUXF7 components that help reduce the size of larger multiplexers and significantly increase performance.

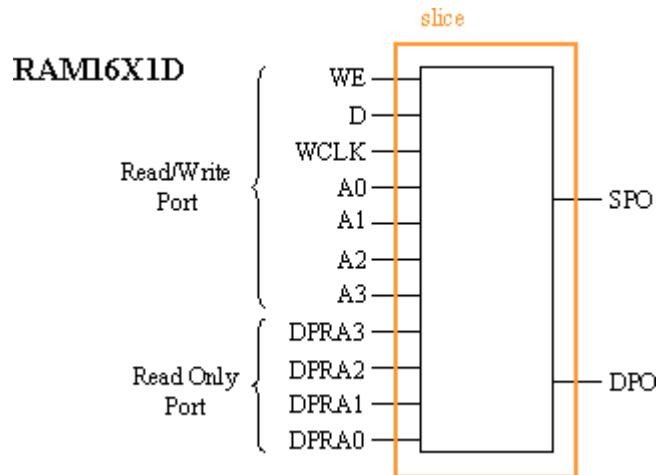
8-bit Register Bank Resources (all figures in 'slices')

Nº Registers	Flip Flops	1-bit Mux	Dual register select Mux's	Enable gates	Packed Size
4	16	1	16	2	18
8	32	2½	40	4	44
16	64	5	80	8	88

So, it appears that the desire for more registers that will make a PSM easy to use must be balanced with the expense of implementation in both size and performance. Fortunately, this is where one of the most powerful features of Xilinx devices comes into play...

Distributed RAM

The SRAM configuration cells normally used to set the "gate" functionality implemented by a LUT are also available within the design directly as RAM. Hence each LUT offers 16 bits of RAM, and a "slice" can provide 32 bits of RAM. Of particular interest to PSM development is the ability to trade 16 bits of RAM per slice in order to achieve a type of dual-port RAM that is ideally suited for implementing a register bank.



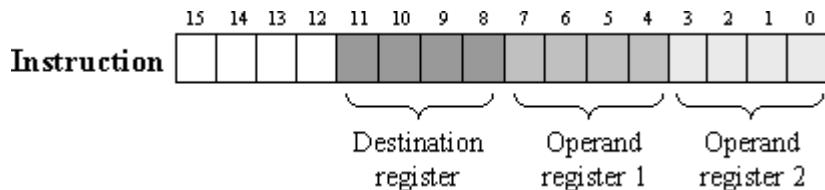
The 16 RAM cells provide all the functionality of a 1-bit, 16-register bank, complete with selective write enable. The second port enables a second operand to be accessed in the same way that a second multiplexer was used with discrete registers. With this great feature, an 8-bit, 16-register bank can be implemented in just 8 "slices" (compared to 88 "slices") and significantly increases performance.

Given the potential offered by these 8 "slices", it is easy to see why a register-based architecture should be considered and why it makes sense to include 16 general-purpose registers in the Virtex-E and Spartan-II PSM implementation.

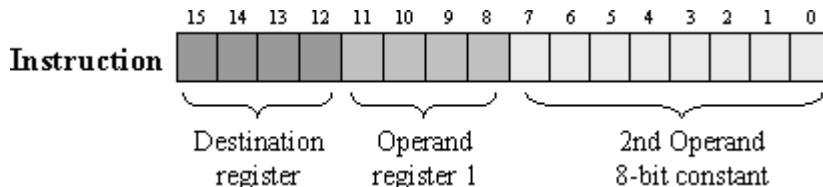
Instruction Considerations

In Part 1, I described how the 256x16 aspect ratio appeared to be advantageous for storing the program. Now that

we have selected a register-based architecture and seen the potential for 16 registers, we can consider the initial format of the instruction encoding.

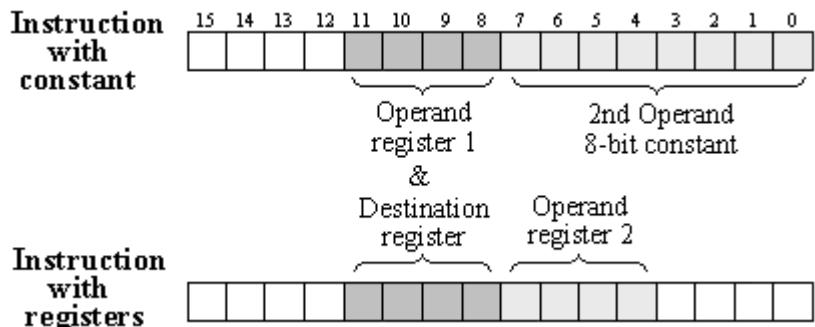


It is also necessary to consider operations involving a constant value. It is desirable to keep all instructions self-contained, so an 8-bit constant must be specified within the instruction encoding together with the register operands.



Clearly, the instruction now has to specify so much operand information that there is simply no space to define the operation. Although exploiting both ports of the block memory could form a wider instruction format, the even wider aspect ratio would further reduce the program length (128x[16+16] aspect ratio). Alternatively, we could consider using a fetch cycle to obtain constant values, but this again would reduce program length and lead to variable instruction size and duration.

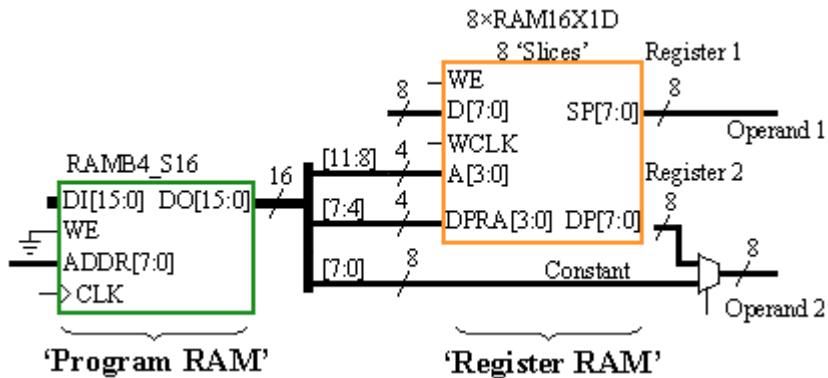
Given the desire to keep all instructions self-contained and maintain the natural 256x16 program ROM aspect ratio, a compromise must be made. In this case, the destination register was inferred by making it the same as the first operand register. This releases 4 valuable bits of instruction for encoding the operation and reveals the primary encoding for all ALU-related instructions.



It is clear that the programming may not be quite so flexible, and that additional instructions will be required when the first operand register contents must be preserved. However, program coding can often be organised such that this is not such a restriction, and can even be advantageous in the same way as an accumulator. The ability to specify a constant as the second operand (at no cost to program size or performance) will also be valuable when defining the instruction set. (I will study this later in the series.)

Memory Controlling Memory

The structure of a PSM is already emerging. Through this very simple and progressive analysis, we can see how important memory is to a processor and why the Virtex and Spartan-II devices are well-suited to this application.



The program memory acts as the controller and is formed using block RAM. The variable data will be held in registers, and these are made cost-effective via the highly efficient distributed dual-port RAM. Without this distributed RAM option, the 44 CLBs (88 "slices") required to implement the register bank alone would already exceed the size of the complete PSM implementation!

In Part 3

In the next article, I will define the program flow control instructions, then examine the details of implementing the program address counter and associated logic. Here, we will see a third way memory can be used in our time-sharing machine.

If you would like to have a look at KCPSM, you may download it at the address below. Full documentation and an assembler are also available.

XAPP213 "8-bit Microcontroller for Virtex Devices":

<http://www.xilinx.com/xapp/xapp213.pdf>

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

This is suitable for all Virtex, Virtex-E and Spartan-II devices. If you would like a PSM that is specially tuned for the Virtex-II architecture, drop me an e-mail at ken.chapman@xilinx.com, and I will be pleased to send it to you.

Creating Embedded Microcontrollers (Programmable State Machines) - Part 3

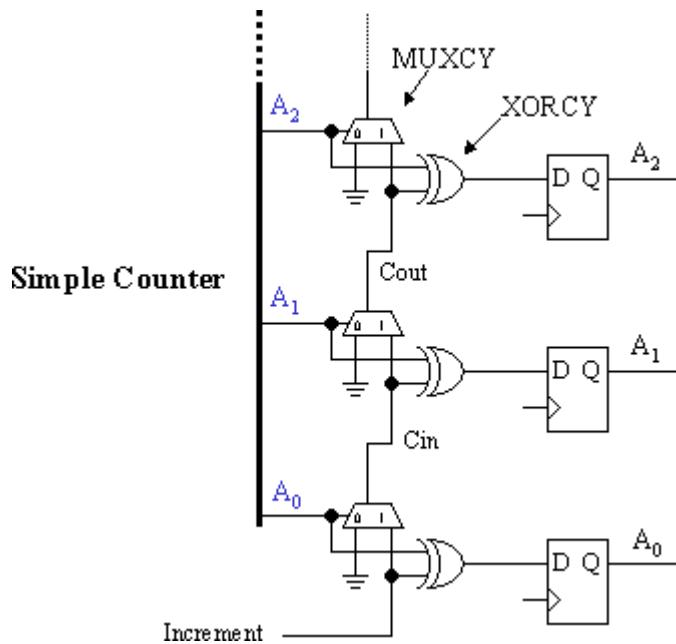
In Part 2, we decided that the program will be stored in a 256x16 aspect block RAM and that the fundamental processing architecture will be 16 registers implemented with the highly efficient distributed dual-port RAM. We will now consider the detail of the implementation.

Part 3 will focus on program flow control aspects and define the instructions required to control a program execution sequence. We will then consider the implementation of the program counter and its associated circuits and realise again that RAM is highly desirable in this "state machine" section of a PSM.

Programs are sequential most of the time!

A traditional processor executes instructions one at a time from sequential memory locations of the program memory. Execution normally starts at memory location zero. This is the reason for the requirement of a program counter (PC), which increments through the program memory locations.

The simple "increment" operation of a program counter is very efficient and easy to implement in the Xilinx Virtex™ and Spartan™-II devices. Dedicated carry logic components and flip-flops within each logic "slice" are all that is required to implement a simple incrementing counter.



The carry logic is able to implement an increment function without using the look up table (LUT). The operation of each bit is defined by the following truth tables. Observe how the MUXCY propagates the carry signal.

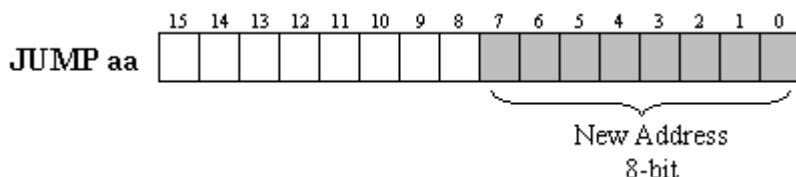
XORCY			MUXCY			
A_n	Cin	A_{n+1}	A_n	GND	Cout	Cin
0	0	0	0	0	0	0
0	1	1	0	0	0	1
1	0	1	1	0	0	0
1	1	0	1	0	1	1

Because each logic "slice" contains the carry logic components and flip-flops for 2-bits, the required 8-bit program counter to access 256 memory locations can be implemented in just four "slices". However, the program counter must do more than increment.

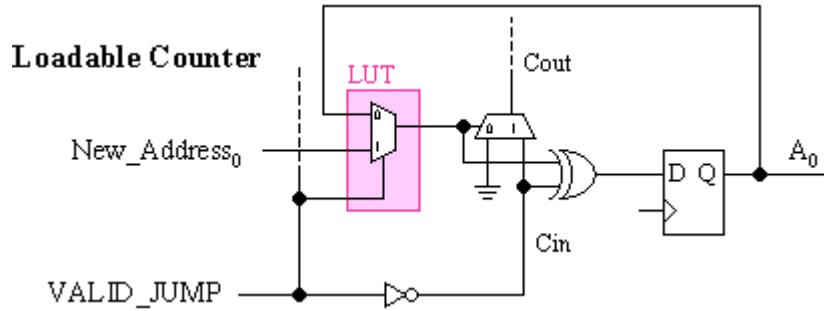
Jump with control

As a minimum, the PSM processor should have a way to repeat sections of code by enforcing the program counter to non-sequential address locations. The ability to jump (or branch) to any specified program memory location would allow for greater programming flexibility. To make a processor practical, the jump would be performed only under specified conditions (conditional jump) so that different sections of code are executed under different circumstances.

In Part 1, we decided that the 16-bit aspect ratio of the block RAM was more suitable for a PSM as it would enable the operands to be included within each instruction. With a JUMP instruction, the operand is the desired program counter value after execution; therefore, the JUMP instruction must specify an 8-bit address for the PSM program counter.



Now, the program counter must be loadable. When the instruction coding indicates that a JUMP must occur, the normal counter feedback must be replaced with the new value and the increment prevented. The LUT associated with each bit of the counter is now ideal for implementing the multiplexer to perform this selection.

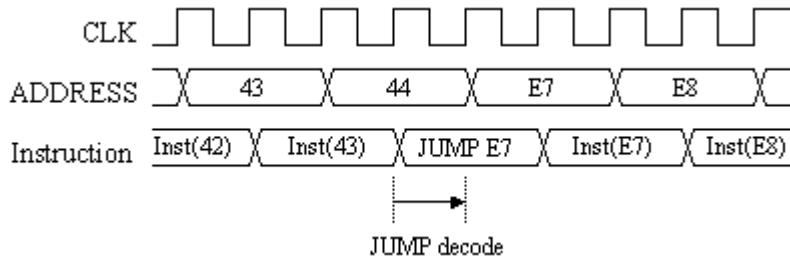


The VALID_JUMP signal will be active-High when a JUMP instruction is detected and when any conditions that are imposed on the execution are met. This will be a logical decode based on the remaining 8 bits of the instruction word and flags from the ALU. The inversion of the VALID_JUMP signal is used to drive the input to the carry chain to prevent the new address value from being incremented as it is loaded. This inverter is absorbed into the dedicated carry logic such that it requires no additional resources and does not impact performance.

Two Clock Cycles per Instruction

The JUMP instruction indicates that a decision will take place, and control over this decision implies the need for timing on the PSM. On the next clock edge, the program counter will either increment or load. The JUMP instruction therefore requires one clock cycle to determine which address of program memory will be accessed next.

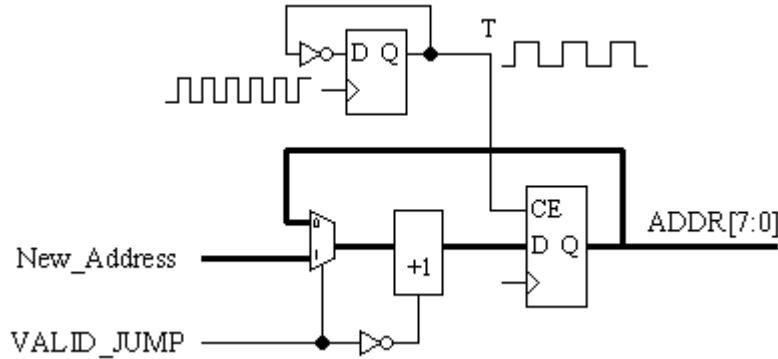
The program is stored in block RAM. It now becomes crucial to realise that the block RAM is a synchronous memory which must be clocked to both write and read data at a given address. Hence, if the program counter takes one clock cycle to determine the address, and the block RAM takes another clock cycle to access the instruction located at that address, the fundamental operation of a PSM takes two clock cycles per instruction.



In the above timing diagram you can see that most instructions result in the simple increment operation of the program counter. When the JUMP instruction located at address 44 is read from the block ROM, the program counter is loaded with the new address value supplied (E7) and the address jumps on the next clock edge.

The decoding of a JUMP instruction must take place in one clock cycle. All other instructions can be completed in two clock cycles; we will exploit this in the ALU and I/O instructions later. Although it is possible to execute all non-JUMP instructions at one-clock intervals, the handling of a JUMP instruction becomes more complex and detracts from the concept of the PSM being small and simple. The constant two clock cycles for every instruction also makes execution time easy to predict when writing a program. Processor terminology refers to the two cycles per instruction as T-states.

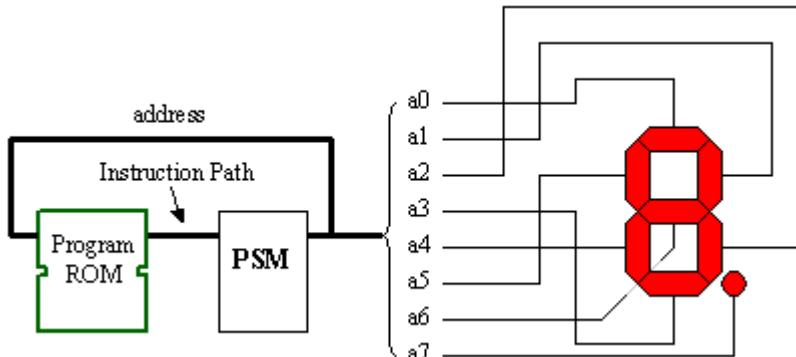
To make the program counter change only every two clock cycles, the dedicated clock enable on each of the counter flip-flops can be controlled using a simple toggle flip-flop.



The toggle flip-flop is essentially the only logic forming the "state machine" of the PSM. The counter logic exploits the capability of the four "slices" very well.

Our First Program -- Without an ALU!

With only a JUMP instruction, we can already see that a program can be written for the programmable state machine. In this simple example, the address signals are used to drive a 7-segment display directly. The program is written to ensure that the address sequence provides a decimal counter on the display. The display would change every two clock cycles.



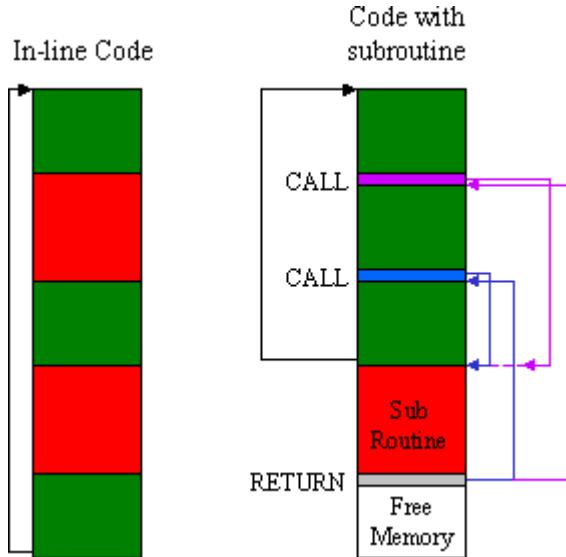
Addr	Instruction	Comment
03	JUMP 5B	; display '1' and go to '2'
07	JUMP 7F	; display '7' and go to '8'
3F	JUMP 03	; display '0' and go to '1'
4F	JUMP 66	; display '3' and go to '4'
5B	JUMP 4F	; display '2' and go to '3'
66	JUMP 6D	; display '4' and go to '5'
6D	JUMP 7D	; display '5' and go to '6'
6F	JUMP 3F	; display '9' and go to '0'
7D	JUMP 07	; display '6' and go to '7'
7F	JUMP 6F	; display '8' and go to '9'

Whilst this simple example may be an interesting concept, it is unlikely to be a practical way to work with a PSM. However, the advantage of a fully embedded processor is that all signals are available to interface with the programmable logic of an FPGA. By organising programs to have routines located at particular addresses, it is possible to decode these addresses and trigger events external to the processor (but still internal to the FPGA) without actually performing I/O operations.

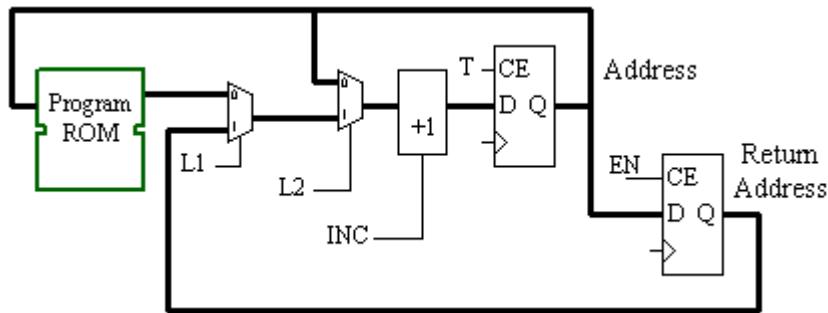
Subroutine for Real Programs

The PSM is intended to be a practical way to exploit time and share logic. A subroutine extends this concept by

enabling sections of common code stored in memory to be shared by different parts of the program. Code does not need to be "in line," and this makes it potentially more compact and easier to write. Given the restricted program space of a PSM, a feature that enables compact code is highly desirable. Anything that makes writing a program easier is always desirable!



The CALL instruction is similar to a JUMP instruction in that the operand will again provide a new address to be loaded into the program counter provided conditions have been met. The RETURN instruction also causes the program counter to load a new address at the end of the subroutine. However, unlike the JUMP or CALL instructions, there is no operand to specify the new address; this must be derived by the PSM itself. To achieve this address specification, the CALL instruction must also store the current program counter value.



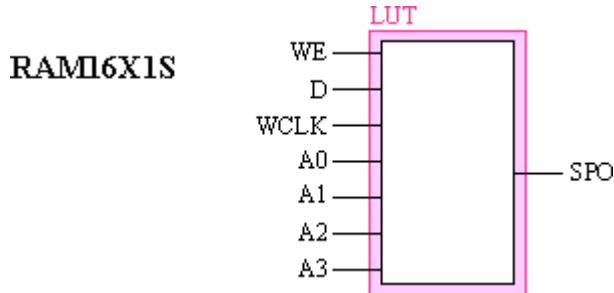
A register connected to the program counter is able to capture the address value when a CALL instruction is encountered. When the RETURN instruction is detected (at the end of the subroutine), the value from this register can be loaded back into the program counter. The program counter now has two sources of new address information, and a second multiplexer is required to make this selection, requiring four additional "slices". Closer inspection reveals that during a RETURN instruction, the return address value must be incremented as it is being loaded so that the instruction following the original CALL is executed.

Instruction	EN	L1	L2	INC
Normal	0	x	0	1
JUMP	0	0	1	0
CALL	1	0	1	0
RETURN	0	1	1	1

Careful allocation of instruction op-codes means that the decoding logic to control these signals can be very simple, and that sometimes an instruction bit can drive a control directly.

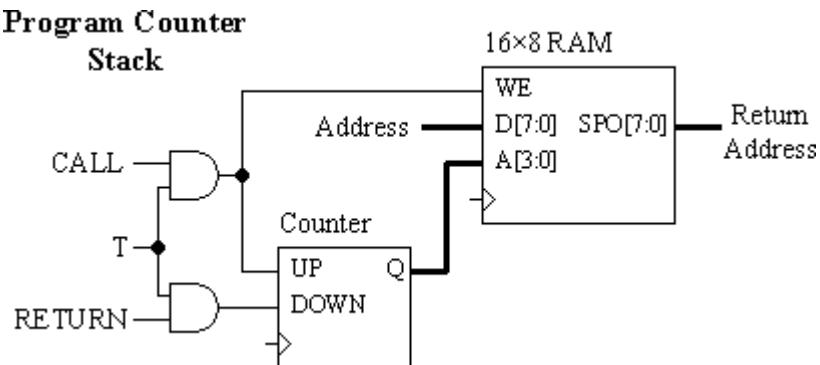
Don't just use it, exploit it!

The register used to preserve the address value is formed of eight flip-flops and therefore occupies another four "slices". In Part 2 we saw how distributed dual-port RAM could very efficiently provide a register bank, and now we can exploit distributed RAM in single-port mode to provide an address stack.



With a program address stack, nested subroutines can be executed, making programs even smaller and easier to write. Because the program address stack is implemented independently of the ALU, registers, and I/O, no special instructions or programming styles are required to initialise or control the stack operation.

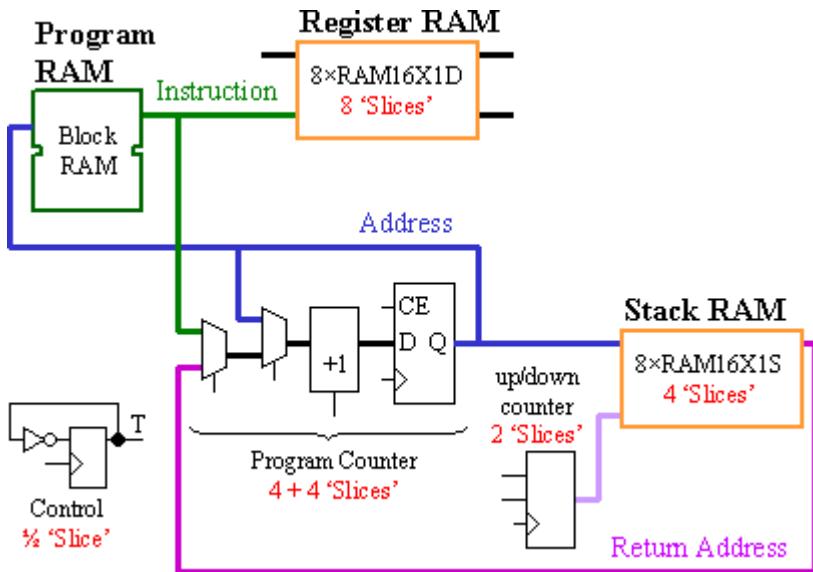
To form a stack, a 4-bit up/down counter requiring two "slices" is used to address the distributed RAM. When a CALL instruction is encountered, the current program address is pushed onto the stack by writing the memory and incrementing the stack counter. RETURN instructions will decrement the stack counter and read the program address from the distributed RAM.



Due to the two cycle per instruction format, the CALL and RETURN instruction decoding must be qualified by the T-state control such that the stack is only "pushed" or "popped" once. Although there are two clock cycles per instruction, only one cycle is available to decode a JUMP, CALL, or RETURN instruction and load the program counter. The operation of the stack should therefore be arranged such that the "top of stack" return address is presented at the output of the stack at the start of each instruction execution in case it is a RETURN operation.

RAM, RAM, and more RAM!

We can now see that the PSM employs RAM three times, each with independent address and data paths. Although the simple instruction decoding logic is not shown, the fundamental program flow control has been achieved using just 14½ "slices".



Distributed RAM has been used twice to provide a total of 256 bits of memory in 12 "slices". Using flip-flops and logic to replace these memory structures would require 152 "slices", which is more than double the size of the complete KCPSM (35 CLBs in Spartan-II provide 70 "slices"). Using block RAM to implement these small memories would be wasteful and would impact the flexibility of a PSM to be used multiple times in devices, especially small ones with limited block RAM.

In Part 4

In Part 4, we will define an instruction set for the PSM and discover that efficient multiplexers are the key to the implementation of the ALU.

To have a look at KCPSM, download it from the address below. Full documentation and an assembler are also available. More than 1000 copies of KCPSM are downloaded from the web site every month. If you already use KCPSM in your designs, it would be great to hear from you, and I look forward to your emails.

XAPP213 "8-bit Microcontroller for Virtex Devices"

<http://www.xilinx.com/xapp/xapp213.pdf>

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

This PSM is suitable for all Virtex, Virtex-E, and Spartan-II devices. If you would like a PSM specially tuned to the Virtex-II architecture, drop me an e-mail at ken.chapman@xilinx.com and I will be pleased to send it to you.

NOTE: Xilinx has recently given the KCPSM reference design the name "PicoBlaze," to indicate the complementary nature of the Programmable State Machines with the high performance 32-bit RISC soft processor called "MicroBlaze" that was released in October 2001. With PicoBlaze(TM) and MicroBlaze(TM), designers now can choose from a range of "right-sized" solutions, from 8 to 32 bits.

Creating Embedded Microcontrollers (Programmable State Machines) - Part 4

At last, we may define the ALU instruction set of our PSM. Whilst the ALU would appear to be the logical starting point for any processor design, I hope you have been able to see that it is better to design a PSM starting first with the considerations of the "life support system" surrounding the ALU. In this way, we have derived the format of the instruction words (16-bit), the number of registers (16), and the necessity to infer the destination register for operations requiring two input operands.

Programmable Logic = Flexible Instruction Set

Another reason for leaving the definition of the ALU instructions until the end of your design process is because this is an area of great deliberation and debate. Can you imagine the long discussions and arguments occurring at companies that design and manufacture microprocessor chips? There must be enough instructions so that the chips are easy to use, but every instruction added makes the processor more expensive to manufacture. Sometimes special instructions may be included to make the processor more suitable for certain applications (such as a hardware multiplier for improved DSP performance), but such specific tuning may then preclude that processor from being adopted for other applications. Of course, any special instructions are only valuable if they are actually used, which may not always be the case if the compiler or software engineer does not invoke them when the opportunity arises.

The advantage of creating a processor inside an FPGA is that the ALU instruction set does not have to be fixed until you decide it is right for you. You may actually define a different instruction set to suit each particular application. In practice, you will probably find that defining one personalised instruction set is adequate unless you implement a very diverse range of products. Even if you discover that you made a poor choice of instruction set well into your design phase, you still have the ability to change it. However, be careful not to make changes too often as this will impact the development time of your product and detract from the advantages the PSM concept offers in the first place.

Selecting Operations

When I defined the instruction set for my KCPSM (PicoBlaze™) macros, I wanted to provide a reasonably general-purpose machine that could be applied to a wide variety of applications. My focus was also to keep the size small, so I avoided including any logically expensive operations that would be infrequently used. It was always clear to me that whatever I chose could be changed in future, so I did not lose sleep over my decisions! Provided that you have a reasonable selection of basic instructions, it should be possible to write programs (or sub-routines) to perform more complex operations. For example, repeated shift and addition operations may be used to realise a multiplication.

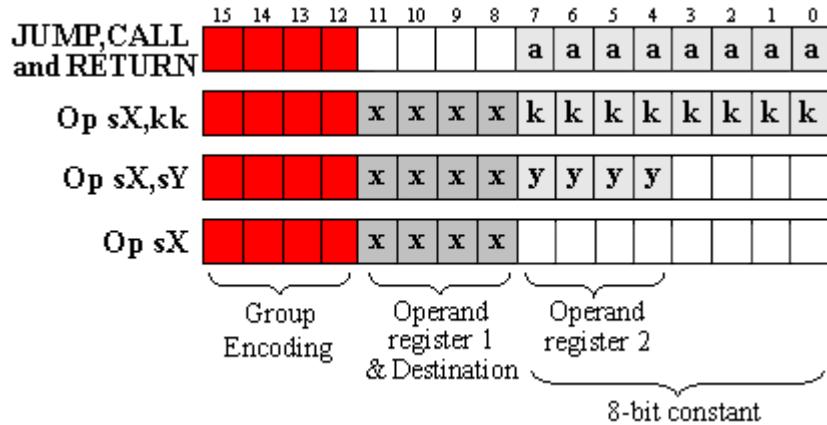
Electing to include a particular operation in the ALU will probably make the PSM larger. However, it will mean that your programs should become easier to write and will execute more quickly. Generally, the concept behind a PSM is to perform complex state machine tasks that are not particularly time critical; therefore, adding logic to the ALU so that programs will execute faster is not high on my priority list, but obviously your objectives may be different. The more significant advantage of adding a special operation is linked to the phrase that programs become "easier to write". The ability to specify one instruction instead of multiple operations not only makes programming easier, but implies a smaller program. Since the most restrictive aspect of a PSM is the length of the program stored in block RAM (256 locations in Virtex™-E or Spartan™-II, and 1024 locations in Virtex™-II), being able to write more compact code can be vital for a successful fit.

A program that nearly fills the available block RAM and executes at an acceptable rate indicates efficient use of the device via time-sharing of the logic resources. Use this to gauge whether it is necessary to include special instructions to make the code smaller and faster.

Some Constraints

The format of the ALU instructions for my PSM macro was partially defined in Part 2. The first operand will be a register (sX), which also implies the destination for any result. The second operand can be register (sY) or an 8-bit constant (kk). There are also instructions that only manipulate the contents of a single register (sX). Part 3 illustrated how JUMP, CALL, and RETURN instructions provide flexible flow control of the program. Although not strictly an ALU operation, we must also consider that the PSM processor will need to communicate externally using some form of input and output ports. My choice for these input/output (I/O) operations was to allow values to be passed to or from a register and a port specified by an 8-bit address. This has the same operands as other ALU operations except that the second operand is used to provide the port address.

Instructions may be formed into groups with common operand types:

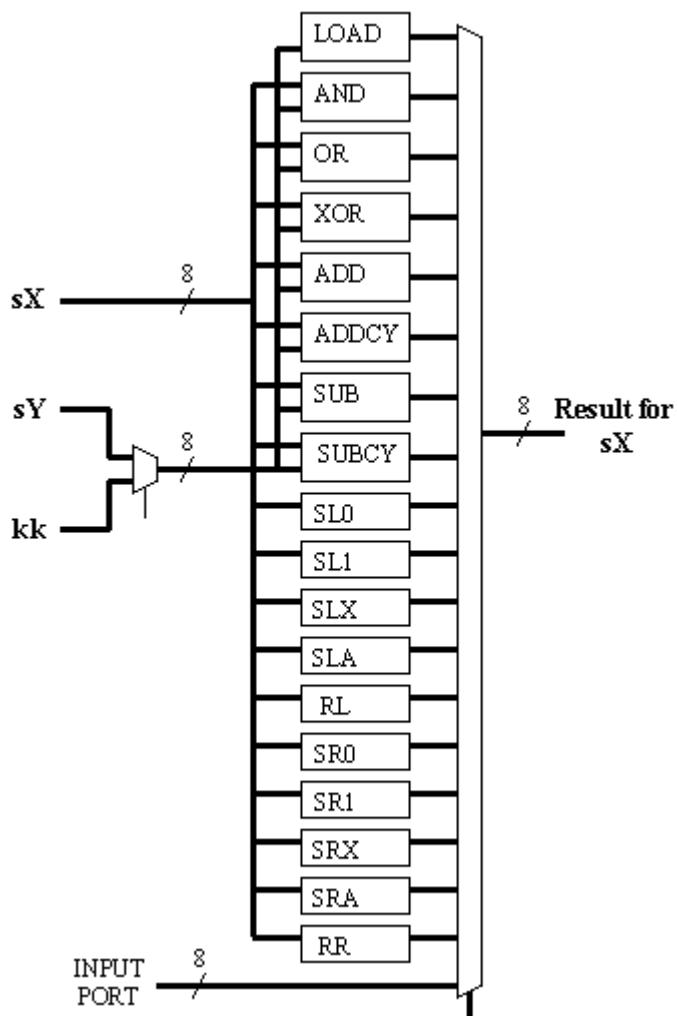


Given that the instruction words are 16-bits, it is clear that only 4 bits are available to uniquely encode the principle instruction groups. Fortunately, this does not mean that only 16 (2^4) instructions are allowed in total, because most of the groups contain "spare bits" that can be used to encode instructions within the same group. You can see that the "OPsX" group has 8 "spare bits" with the potential for up to 256 (2^8) instructions within the single operand group. The "OPsX,sY" and JUMP, CALL, and RETURN groups each have 4 "spare bits" with the potential for a further 16 (2^4) instructions in each group. Therefore, the greatest constraint comes in the "OPsX,kk" group where no "spare bits" exist.

The "OPsX,kk" is the original reason behind the decision to infer the destination register to be the same as the first operand (see Part 2). It is now obvious that this instruction group must be fully encoded by the primary 4 bits, which must also be used to identify the other groups. Therefore, the maximum number of instructions in this group is 13 (or 14 for those who really want to push it!).

Multiplexing

The key to an efficient ALU implementation is reducing the amount of multiplexing logic. In the most primitive implementation of an ALU, every operation is performed in parallel regardless of the instruction being executed at the time. The instruction decoding logic then selects the appropriate result. To make the following descriptions easier to understand, I will focus on my implementation of the KCSPM (PicoBlaze) instruction set. The 18 ALU-based instructions certainly require a great deal of multiplexing. There is also the simple 2:1 multiplexer to choose either a register or constant value for the second operand (introduced at the end of Part 2).

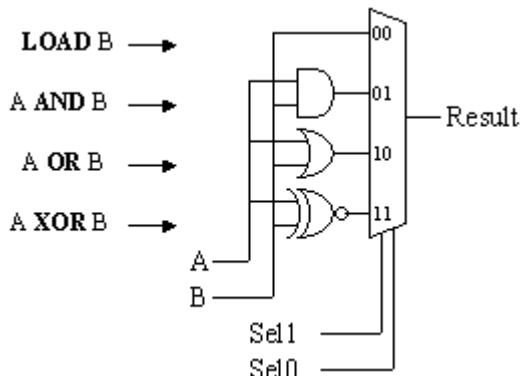


The apparent requirement for a 19-input multiplexer is alarming! Even with the dedicated multiplexers within the logic slices (MUXF5 etc), the size of this multiplexer for the 8-bit data width would be 48 "slices". This is very expensive, considering that each actual ALU function has yet to be implemented. This multiplexer would also require 5 select lines -- it would be an interesting task to encode these from the 4 primary bits and various "spare bits" of the instruction words.

Obviously, the amount of multiplexing may be reduced by decreasing the number of available ALU operations, but this hardly the most desirable solution! In Part 2, we saw that multiplexers could be very expensive in a register-based processor structure, but we avoided this situation by using the look-up tables in a RAM mode. We must now see how the look-up tables may be exploited to combine some multiplexing with the ALU operations that must be implemented.

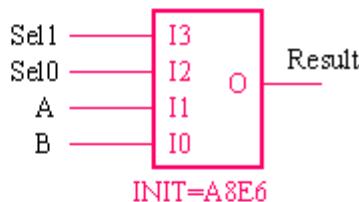
Logical Group

The logical group of instructions provide bit-wise logic operations between two operands. If the set of operations is drawn out in more detail together with a multiplexer, we make a powerful observation:



For each bit of the operands, we can see a 4-input function -- a perfect fit for a look-up table. In this way, the entire 8-bit logical group (including the multiplexer) can be realised in just 4 "slices" rather than 20 slices (4 "slices" for each AND, OR, and XOR, plus 8 for a 4:1 multiplexer). This reduction in size again provides the benefit of higher performance.

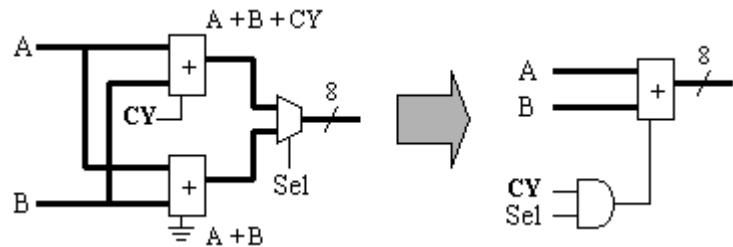
Logical Group						
I3	I2	I1	I0	O	Result	INIT=A8E6
Sel1	Sel0	A	B			
0	0	0	0	0	0	
0	0	0	1	1	1	6
0	0	1	0	1	1	
0	0	1	1	0	0	
0	1	0	0	0	0	
0	1	0	1	1	1	E
0	1	1	0	1	1	
0	1	1	1	1	1	
1	0	0	0	0	0	
1	0	0	1	0	0	
1	0	1	0	0	0	8
1	0	1	1	1	1	
1	1	0	0	0	0	
1	1	0	1	1	1	
1	1	1	0	0	0	
1	1	1	1	1	1	A



The above truth table derives the INIT value for each LUT forming the logical group. The same mapping should be achievable via synthesis tools now that we are breaking the description of the ALU down into manageable pieces.

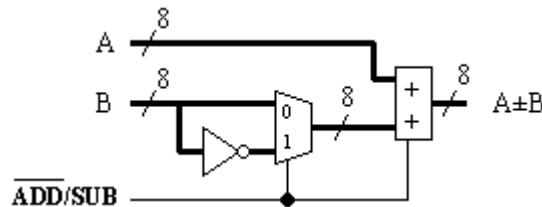
Arithmetic Group

The Arithmetic group performs an 8-bit addition or subtraction with or without the inclusion of a CARRY flag. The first reduction can be made simply by reordering the description of the optional CARRY operation:

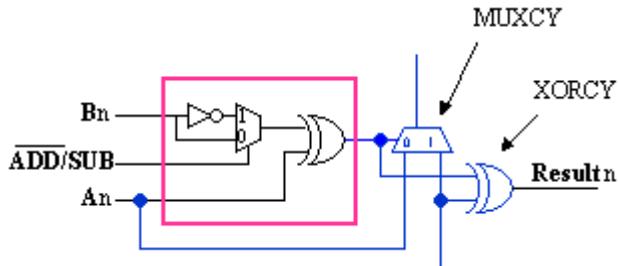


Rather than use the select signal as a multiplexer control, we now use it as a masking control to the CARRY (CY) signal. Since this is a single AND gate, it only costs a ½ "slice" compared with 4 "slices" for a multiplexer and a further 4 "slices" for a second adder. Although this is a simple observation, you must consider this when writing any HDL code.

More knowledge about the arithmetic capability of the Xilinx devices tells us that the addition and subtraction operations may also be combined. This is also a case of reordering the functions in order to place the multiplexer in front of the carry logic. Subtraction is performed in the "slices" -- the addition of the "A" input with the twos complement of the "B" input. To perform the twos complement of "B", each bit of "B" is inverted (the one's complement), and the addition of "1" is achieved by applying a carry input to the adder.

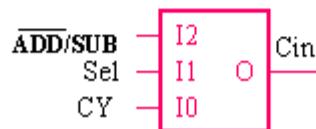


Making the complement of the "B" optional requires a multiplexer, but there is no problem absorbing this into the function generator, which performs the "half add" preceding the dedicated carry logic. This rather complex-looking function generator reduces to a 3-input function and is more typically represented as a 3-input XOR gate.



The input to the carry chain must combine the effects of the twos complement addition of "1" with the optional addition of the CARRY flag. This reduces to a 3-input function and therefore occupies no more space than the simple masking AND gate required for the optional carry input adder. The most complicated cases of this table involve subtracting. The simple SUB operation means that the input to the carry chain must be forced High to provide the twos complement addition of "1". When performing a SUBCY operation, the CY input must be inverted such that when CY=0, the twos complement addition of "1" still takes place. When performing a SUBCY operation with CY=1, the inverted CY prevents the twos complement addition of "1" so that the result is the required A-B less one (i.e., A-B-CY where CY=1).

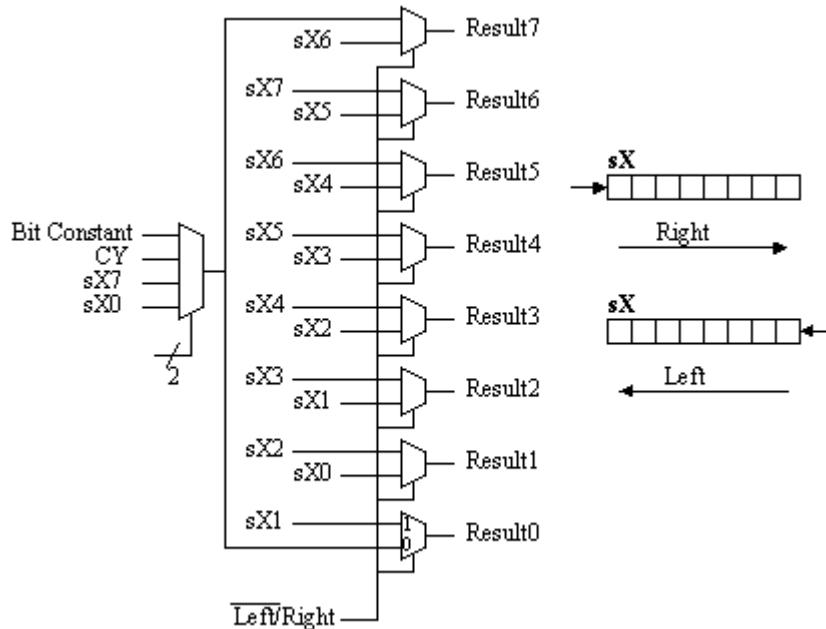
Instruction	$\overline{\text{ADD/SUB}}$	Include CARRY	Carry Chain Cin
ADD	0	0	0
ADDCY	0	1	CY
SUB	1	0	1
SUBCY	1	1	not CY



Now, the entire arithmetic group is implemented by just 4 ½ "slices" rather than 24 "slices" (4 "slices" for each add or subtract, plus 8 for a 4:1 multiplexer).

Shift and Rotate Group

The shift and rotate group is a simple case of selecting the correct bits via multiplexers; it is obviously reduced to the following structure:

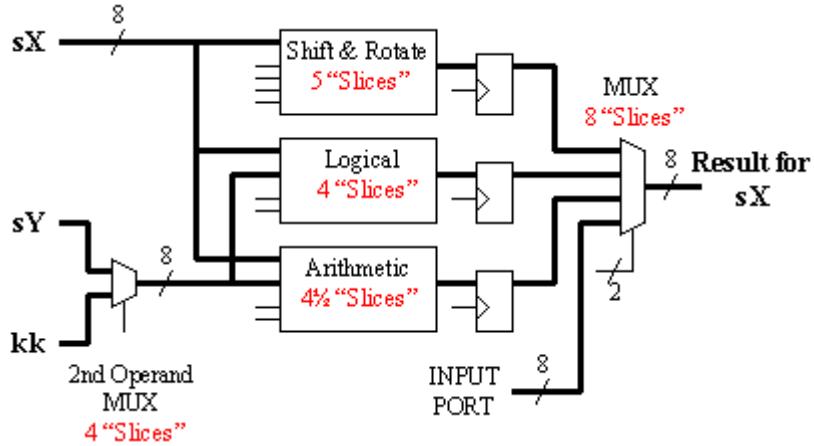


Anyone familiar with KCPSM (PicoBlaze) may have wondered why the shift and rotate group is so comprehensive. When examining the above structure, we can see that all the operations share the same 2:1 multiplexer created in 4 "slices". A single "slice" is then required to select the information to be injected into the MSB or LSB of the result. The multiplexer select lines and the "Bit Constant" are easily provided using 4 of the 8 "spare bits" available in the "OPsX" group.

Completing the ALU

Some final multiplexing is required to combine the three main ALU instruction groups and provide an input data port. This final 4:1 multiplexer is efficiently implemented by 8 "slices", including the MUXF5 dedicated multiplexers.

If you create your own ALU, this final multiplexer is always worth careful consideration. The 4:1 is a perfect fit in the Xilinx architecture. There is nothing to be gained from having a 3:1 multiplexer, but a 5:1 would be larger and add delay with associated lower performance. The next natural fit would be an 8:1 multiplexer at a cost of 16 "slices".

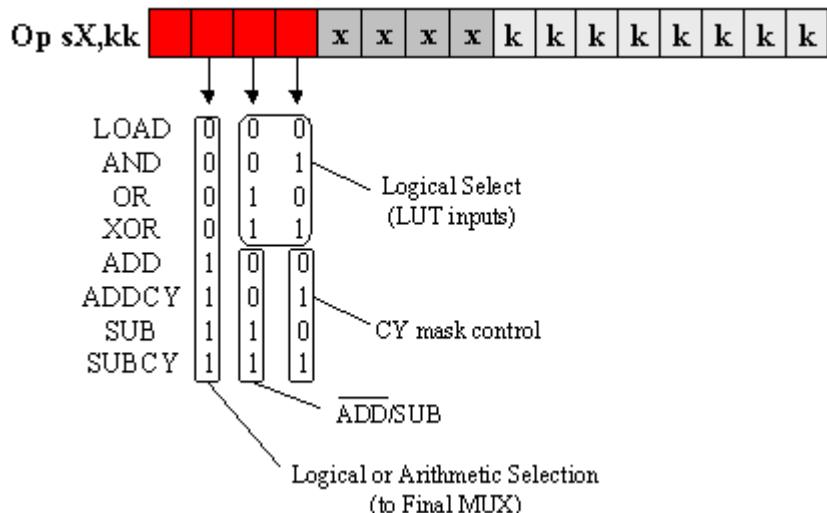


In Part 3, we saw that the PSM would work with a constant 2-clock cycles per instruction. This means that the ALU has 2 clock cycles in which to complete any operation. The path through the ALU would certainly become the critical delay path and set the performance of the the PSM as a whole. Although time specifications could be applied to indicate these 2-cycle paths, there are plenty of flip-flops available to insert a pipeline stage and simplify all time specifications to the normal clock period.

ALU Details

The size of an ALU will depend on the instruction set that you define. However, I hope you can apply similar techniques that I have used to implement the KCPSM (PicoBlaze) ALU in 25½ "slices". Remember to see which functions can be reordered and combined to minimise the actual number of logic blocks working in parallel and reduce the size of the final multiplexer.

All the reduction also helps to simplify the task of decoding the instruction words and controlling the select lines on each of the ALU blocks. Indeed, if you are allocate the instruction encoding carefully, many of the bits can be applied directly to the ALU blocks, as illustrated by the most constraining "OPsX,kk" operations of KCPSM (PicoBlaze).

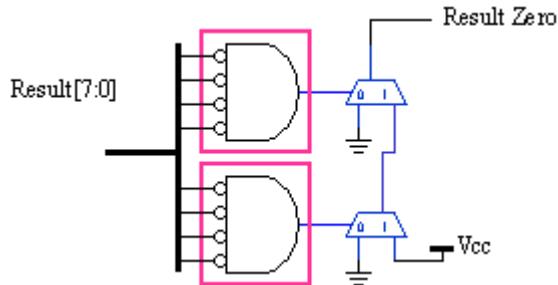


The ALU also generates flags used in some operations (i.e., ADDCY) and program flow control. KCPSM has a CARRY flag and a ZERO flag, but you may decide that other flags are more useful in your own PSM processors. Each flag may be implemented using a simple clock enable flip-flop to ensure that they are only updated during appropriate

ALU operations.

The CARRY flag is forced to zero during logical operations and captures either the carry chain output during arithmetic operations, or the bit "shifted out" during shift and rotate operations. The CY flag is therefore driven by a simple 1-bit multiplexer.

The ZERO flag is set when all bits of the result from any operation are low. This is a simple 8-bit NOR operation and can be efficiently implemented within a single "slice". As well as saving a LUT, the use of the carry chain in this way ensures a minimum delay. Delay in forming the ZERO flag value can be critical, as it follows the final MUX of the ALU.



In Part 5

In the final article of this series, I will present some alternative architectures to consider when implementing a PSM, make some comments about programming tools, and discuss a few possible applications of a PSM. It would be great to include some of your applications as examples, so do keep letting me know what you are using KCPSM (PicoBlaze) for yourself.

To have a look at KCPSM, download it from the address below. Full documentation and an assembler are also available. More than 1000 copies of KCPSM are downloaded from the web site every month. If you already use KCPSM in your designs, it would be great to hear from you, and I look forward to your emails.

XAPP213 "8-bit Microcontroller for Virtex Devices"

<http://www.xilinx.com/xapp/xapp213.pdf>

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

This PSM is suitable for all Virtex, Virtex-E, and Spartan-II devices. If you would like a PSM specially tuned to the Virtex-II architecture, drop me an e-mail at ken.chapman@xilinx.com and I will be pleased to send it to you.

NOTE: Xilinx has recently given the KCPSM reference design the name "PicoBlaze," to indicate the complementary nature of the Programmable State Machines with the high performance 32-bit RISC soft processor called "MicroBlaze" that was released in October 2001. With PicoBlaze and MicroBlaze™, designers now can choose from a range of "right-sized" solutions, from 8 to 32 bits.

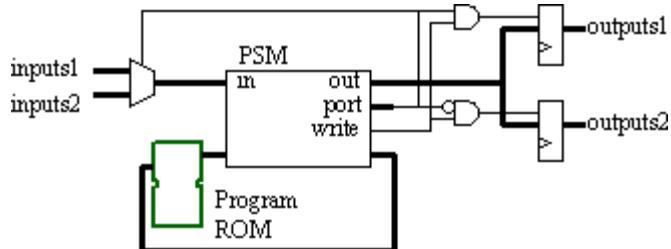
Creating Embedded Microcontrollers (Programmable State Machines) - Part 5

In the final article of this series, I will examine some alternative techniques to consider when implementing a PSM, make some comments about supporting programming tools, and mention just a few PSM applications.

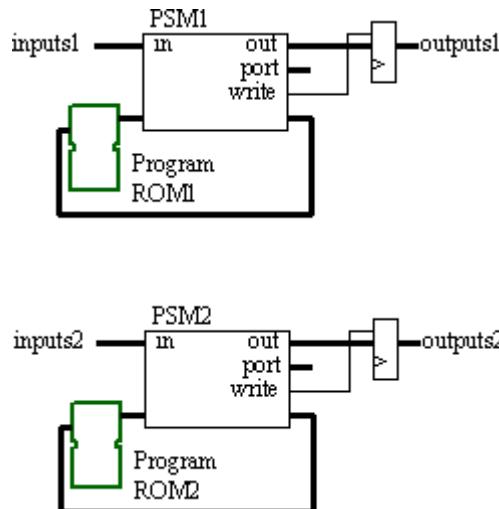
Extending Program Size

In my PSM macros, I chose to utilise a single block RAM to store the program. This obviously restricts the size of the program to 256 instructions in Virtex™-E and Spartan™-II devices when we use the 16-bit aspect ratio, which enables the operands to be included with each instruction. Many of you have told me how you have reached this memory limit; therefore, it seems likely that you will elect to have a larger program space if you implement your own PSMs. I will therefore focus on techniques suitable for supporting these larger programs.

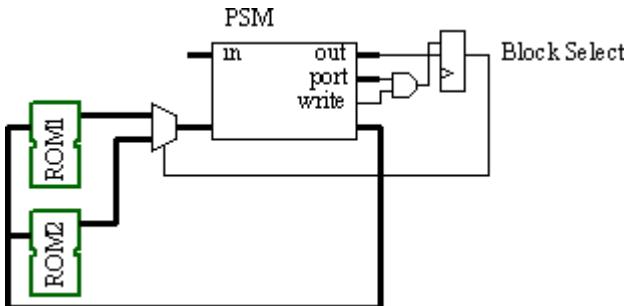
Before we take the "easy" option, let us consider ways to work with the PSM structure we have studied so far in this series. It is not unusual to hear from PicoBlaze™ (KCPSM) users who have included two or three processors in a single design. They realise that distributed processing is the solution, which means that each PSM processor then has its own program memory. Although this may sound extravagant at first, this approach reduces the amount of I/O interface logic and has the advantage of totally independent code, which is easier to develop and test. The following diagram illustrates the interface logic to work with just two simple inputs and outputs:



The diagram below shows how the use of two PSM macros simplifies the I/O interface. It may also help with the layout of a design in a large device. (After all, if you were implementing hardware state machines, they would be implemented separately.)



Others have also reported success through implementing a form of memory-swapping under software control. Typically, a sub-routine located at the same position in each program ROM controls the selection by writing to an output port. In practice, it is highly likely that other sections of code will need to be repeated in each of the memory blocks. Hence if two blocks are used, this will not actually yield twice the available program space. This technique does allow a common PSM macro and support tools to be used in a wide range of applications requiring small or "large" programs.

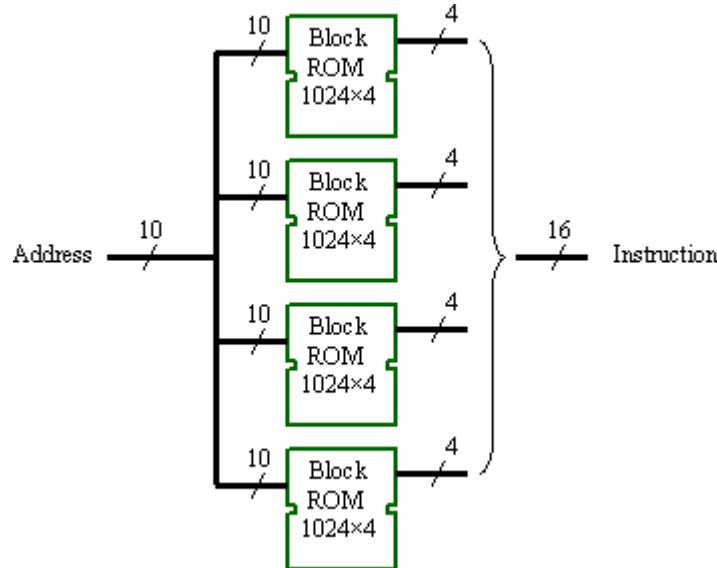


The multiplexer in the above diagram could be saved by using the block RAMs in 512x8 aspect ratio, then using the select bit to address the MSB (9th) address bit. However, the instructions will then be split across the RAM blocks, and programming the ROMs will become more challenging!

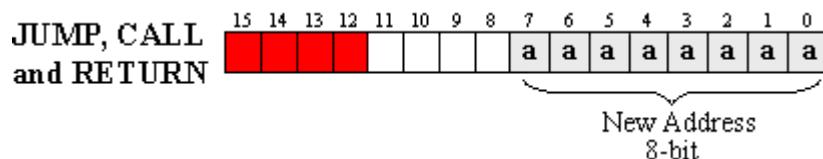
Larger Program Memory

Although block-swapping is possible, it does distract from the ease of use that a PSM should offer to a designer. So if a larger program memory really is going to make things easier, we should just consider the impact and use the best techniques.

Larger program memory may be constructed by combining several block RAMs. Using CORE Generator™ is probably the easiest way to construct such a memory, given the need to initialise the contents of the memory with the program. Each instruction code will be split across the blocks rather than using the memory in "block pages". Larger memories constructed in this way require no additional logic, as the multiplexing and address decoding is achieved directly by the block RAMs. In the diagram below, a memory of 1024 locations of 16-bits is implemented. Consider the impact of using the same block RAMs organised in 256x16.

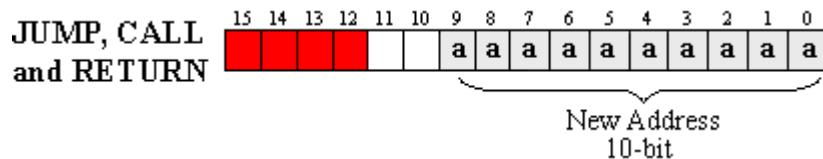


Increasing the size of the program memory sounds like an easy thing to do, but the PSM is now required to support the extended address range. To increase from 256 locations to 1204 requires that the program counter grow from 8 to 10 bits. Likewise, the width of the CALL/RETURN stack must grow. However, these only increase the size of the PSM by a few "slices". The real issue is determining how to specify the address operand in the JUMP and CALL instructions.



In Part 4, we examined the pressure that existed on the four "primary" bits of the 16-bit instruction word. For this reason, the "spare" bits associated with the flow control group will be used to further encode these instructions. Not only must we distinguish between JUMP, CALL, and RETURN, but we also must encode if the instruction is conditional. This was a pretty tight fit in my own KCSPM design, which allows unconditional and conditional instructions with conditional tests for zero, not zero, carry, and not carry. This leads to a total of 15 combinations, which obviously puts pressure on the 4 "spare" bits.

Now, consider what happens when the address operand grows to 10-bits...



It is clear that there would be no way to encode all the desired instructions in the two "spare" bits that remain. Retaining a 16-bit instruction format would either require a reduction in the number of conditional flow control instructions or place more pressure on the primary bits, causing a potential reduction in ALU instructions. Neither of these is desirable; we should also remember that the more encoded the instructions become, the larger and slower the PSM decoding logic will become.

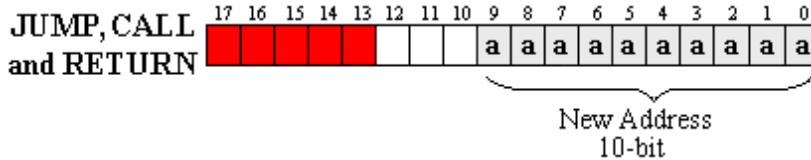
Traditional processors solve this same issue using a variety of methods:

Operand Fetch: Rather than keep each instruction fully self-contained, operands are obtained from the program memory as required by using subsequent memory locations. This uses up some of the additional memory space we are trying to provide and leads to fetch cycles that complicate the PSM internal state machine.

Relative JUMP: This limits the distance you can "jump" to an address relative to the current location. An 8-bit address operand is typically a two's complement value that allows the program counter to be increased by up to 127 and decreased by up to 128. This works well with small routines, but doesn't make movement between routines very nice. Calls to subroutines outside the range would be impractical. Programming of this type of processor really requires an assembler supporting labels, and the program counter logic must implement a signed addition.

Pre-Fetch Instruction: An instruction is used simply to provide operand information that is then available for use by a subsequent instruction. The operand is loaded into a holding register inside the processor. Once again, this requires a memory location and additional logic inside the processor.

These methods are ideal when the operands can be large in comparison to the available program memory width. They were the only sensible methods to use for full 8-bit data and 16-bit address range processors using external byte wide memory. However, a PSM is intended to be 100% embedded, and it is good to further exploit the flexibility of the devices and the "virtual pins" that the embedded state offers. Since we are only trying to make the address operand a few bits longer, a solution would be to make the program memory wider. At 18-bits wide, the additional 2-bits are provided, and the number of bits available to encode the instruction is restored.



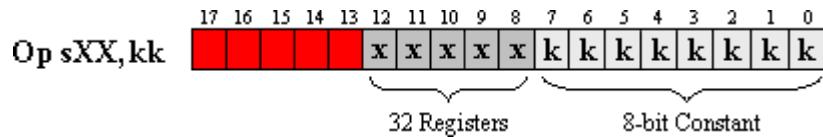
This is not an obvious solution unless you are familiar with Xilinx FPGAs, and it still may not be obvious now. People tend to think in terms of 8, 16 and 32 bits when it comes to processors and memory, so a memory width of 17, 18, 19, or 20 bits sounds strange. In Part 1, we opted to make all instructions 16-bits wide, as that was the widest aspect ratio of the Virtex-E and Spartan-II block RAM. To increase the width by a few more bits does not seem to be a naturally good fit until you combine this with the fact that a larger program memory is implemented by joining multiple block RAMs together, each configured in a deep but narrow aspect ratio.

Hence, five block RAMs in 1024x4 mode implement a 1024x20 program memory. The additional 4-bits now available to describe each instruction will also help reduce the pressure on the primary encoding and make the PSM smaller, faster, and easier to design. Alternatively, they may be used to provide the PSM with more features and instructions.

Virtex-II

Rather fortuitously for PSM macros, the block RAMs provided in Virtex-II devices are four times bigger than those in Virtex-E and Spartan-II devices, and generally provide adequate memory for PSM-based applications. But the good fortune doesn't stop there. In addition to supporting a 1024x16 aspect ratio on the main data port, these blocks provide an additional bit for each byte of data, with the intention of storing parity information. This means that the memory is actually 18-bits wide when 1024 locations are provided.

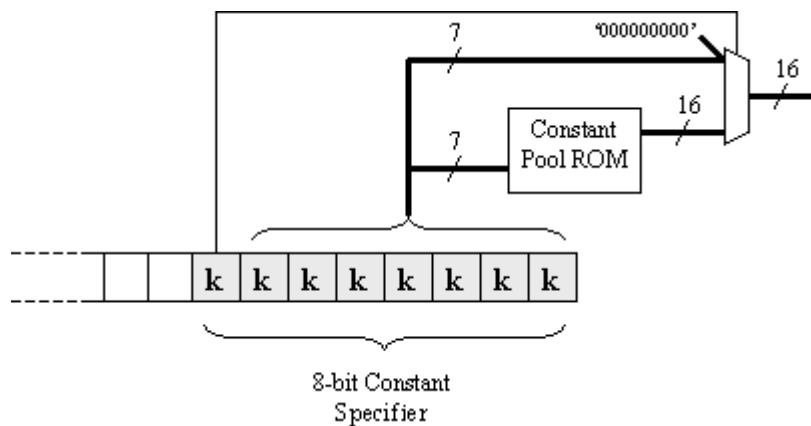
In my PSM macro that is specific to Virtex-II (KCPSM-II), which supports the 1024 locations of program space, I also used the additional bits to increase the number of internal registers to 32 (requiring 5 bits to identify) and make the instruction encoding easier. The most demanding "OpsXX,kk" instruction then had 5 primary bits remaining for instruction encoding.



Fetch without Fetch!

There will still be occasions when a PSM would benefit from the ability to specify large operands. Personally, I think it would cease to be a "programmable state machine" if the program length exceeded much more than 1024 instructions. However, it may make sense to implement a PSM with more than 8-bit data, and as previously discussed, that would probably mean a leap up to 16-bits. This really makes it impractical to include constant information within an instruction word because the program memory would have to become so wide that the additional bits would not be used in other instruction groups. It appears that we should return to the concept of a fetch cycle to obtain constant values when required, but we can modify this in the FPGA implementation.

CONSTANT POOL - Given the relatively small size of PSM programs, only a limited number of constants will be in use. Even when the data bus supports 16-bit values, further study of typical programs reveals that many constants are small values (such as "0" used for clearing registers and "1" used when a software counter is incremented). A "constant pool" is an additional memory in which constant values required by the program are stored separately from the program. The operand of the instruction is then used as an index address to this memory in order to locate the required constant. Small constants can still be contained directly in the instruction so that the constant pool may be kept small. Distributed memory is ideally suited to this function.

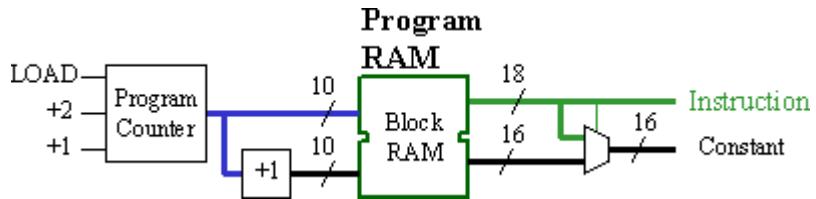


In the above diagram, the MSB of the constant operand is used to specify whether the lower 7 bits should be used directly to supply the constant value in the range 0-to-127, or used as an address to the constant pool to access 16-bit constants in the range 0-to-65535. It is useful to have an assembler to identify the constants that exceed the limit of 127 and build the indexed table of values automatically. Programming requires that both the program memory and the constant pool memory be initialised.

A trade-off occurs when the size of a constant pool is selected. In this example, although 7 bits are available for addressing the constant pool, it is likely that only 4 or 5 bits would be required to address 16 or 32 locations in a distributed memory. If the pool is too small, it is possible that certain programs will require more values than can be stored. If the pool is too large, the PSM will become larger, and there is a point at which the memory would be better used to make the instruction format wider.

PARALLEL ACCESS - This technique is very similar to an operand fetch cycle in that the operand is stored in the program memory location following the instruction. Although this uses program memory space, it has the flexibility to store any number of constants required by the program. As with the constant pool technique, it would still be a good idea to represent small constant values within the single instruction word and reserve the "fetch" for larger values.

The parallel access technique exploits two features of the PSM concept. First, it exploits the dual-port ability of the block RAM to read two locations of program memory simultaneously. Less obviously, but more significantly, it exploits the 100%-embedded nature in order to allow all the additional "virtual pins" that are required to connect the PSM to the program memory for a second time.



The above structure assumes that a 16-bit PSM is being formed in Virtex-II. An additional 26 "virtual pins" allow parallel access to the program memory to access operand information when required. An incrementer is used to access the location following the current instruction at all times. The multiplexer following the memory is used to select between small and large operand values. The program counter must be enhanced to allow it to jump forward by two addresses in the cases when a large operand is stored.

Software Support

If you design your own PSM, it won't be very long before you are frustrated with manually encoding machine code for it to execute. Besides the time consuming effort of programming in this way, it is also rather prone to errors. As with any complex state machine, a PSM has a potentially large number of "illegal states" that correspond to all the unused instruction word combinations. It is likely that you will need to develop and debug your software program anyway, so having to deal with incorrect machine code is unacceptable. It is therefore important that you also invest some time in providing an assembler to accompany your PSM macro.

Given the restricted size of the program memory, and the anticipation that a PSM works very closely with the hardware that surrounds it, it is my opinion that an assembler is highest level of abstraction that should be considered when programming a PSM. On all occasions that I have been asked if there is a C compiler available for PicoBlaze (KCPSM), a discussion about the application has rapidly revealed that a PSM would not be suitable and that MicroBlaze™ would be a much better choice. Since MicroBlaze is a 32-bit RISC processor it is fully supported with a C-compiler and indeed this would become the most suitable way to write programs for it.

So where do you get an assembler for the PSM you have created? Well I understand that there are some assemblers on the market that can be tuned to a given instruction set, but it may just be fun to write one yourself! Even though my software skills are limited and I am more comfortable writing assembler for a PSM, it only took a couple of days to produce a simple assembler for KCSPM (written in Microsoft QuickBASIC for DOS). If you don't feel like writing one yourself, try asking your software friends. You would be amazed how many of them can produce you something usable in a day and enjoy doing it (although it may cost you a couple of beers!).

I do not intend to explain how an assembler works here, but the principle is relatively straightforward. The majority of my assembler source code is used to identify syntax errors and provide constructive feedback. If you are prepared to be more careful about entering your PSM assembler code, the assembler can focus on the generation of machine code only and will be much easier to write.

TEMPLATE MANIPULATION - Once the assembler has done its job, you are faced with how to get the machine code into the block RAM of the design. Template manipulation is a very easy way to accelerate this process and allows the assembler to output a file that is immediately suitable for use by the Xilinx tools. The assembler reads a template text file of the required format except that the actual data values for the program memory are replaced by a special string of characters. The assembler then identifies these special strings and replaces them with the actual data before writing the modified file out.

The most simple template file for a PSM is a coefficient file for use with the Core Generator. This really only needs to have the "memory_initialisation_vector=" string at the start, but may also include other configuration information used to define the "single-port Block memory" or "dual-port Block memory" cores more completely (e.g., "width_a=16;" and "depth_a=1024;"). This flow requires that CORE Generator be executed each time a change is made to the program, but it is the easiest method when creating larger memories from multiple blocks.

If you are prepared to put a little more effort into formatting of the data values, then a template file could be of a type that is even more readily integrated into the Xilinx ISE tool flow. In this case, the template could be an EDIF net list or VHDL description in which a block RAM primitive component is instantiated. The initialisation strings would not be nice to construct manually and emphasises the benefit of CORE Generator in most cases. The following is an example of one of the 72 initialisation strings required for a Virtex-II block memory instantiated in VHDL (XST):

INIT_02 => X"3A01608354083A106083608B6120170A60C2C000607F1804400860D592185022",

I am currently investigating a template required to use the DATA2BRAM utility that would allow the PSM program to be changed directly in the configuration bit stream; this would allow very rapid code iteration cycles in the same way that MicroBlaze users enjoy already.

Applications of PSM Processors

I'm pleased to note that PicoBlaze (KCPSM) is very well-used and continues to be a popular download from the Xilinx Web site (over 9,000 in the first six months of 2002 alone). I am sure that PicoBlaze is not suitable in all cases, but this is a clear indication that there are many applications that fall into the "complex state machine" category for which timing is not so critical. It is so nice to know how many "boring" things PicoBlaze has been used to accomplish! However, other applications exist that really include "processing" as well as other innovative applications.

"Make it easy" applications - The vast majority of PSM-based applications are those in which a hardware design could be employed, but the complexity of the state machine design would make it difficult to enter in a schematic or HDL. A digital clock with a timer is not that complex in itself; however, when you begin to consider how the display should change, and how to allow the user to set the time and alarms using just two press switches...this makes it very ugly in hardware!

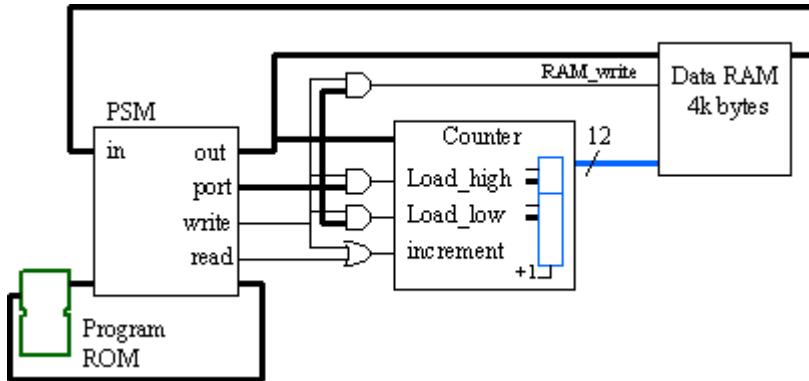
FPGAs are also used to "glue" many systems together. Spartan-II is an ideal device for this, given its low cost. However, so many devices to which you interface expect to be communicating with a processor of some kind. For example, I recently used an LCD display module of the type that can display one line of 16 characters. This module is able to respond to commands and fully understands ASCII characters.

However, the module expects commands and character data to be presented in a very specific order and with particular timing. There needed to be a delay of at least 40us between the writing of characters, but with a delay of more than 1.64ms after a command to clear the display was issued . This would all be very messy in pure hardware and would require some interesting counters in order to establish the timing. In contrast, this was very easy to write in assembler including software delay loops. So, a PSM was a natural interface to this LCD module and was an excellent way of interfacing a high-performance FPGA with a much slower component.

Including PSM macros in a design can also make other processing applications easier. MicroBlaze is a very capable 32-bit RISC processor, but if tasks are offloaded to a PSM, MicroBlaze is allowed to focus on the data processing at which it excels. As we look at Virtex™-II Pro devices with their embedded PowerPC processors, I can envision systems in which MicroBlaze and PSM processors are also included to provide a hierarchy of processing options each suited to their tasks. Distributed software processing is coming to an FPGA near you soon!

PROCESSING APPLICATIONS - It is important to remember that a PSM is a state machine and not really a data processor -- most significantly, it really doesn't have the concept of a memory map. PicoBlaze has an 8-bit port identifier that provides up to 256 input and 256 output ports. Although some of this port map could be used to access memory, again, this is a relatively limited space; therefore, data processing applications must generally be restricted to a minimal number of variables held in registers and small data sets in external memory (relative to the PSM but probably still inside the FPGA). Applications such as motor control fall nicely into this limited data space and also match well with the available performance.

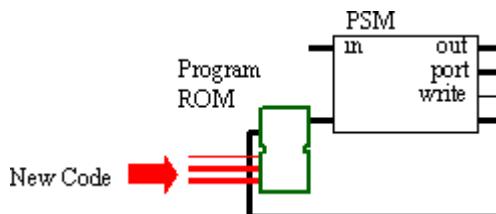
Access to larger data sets may be possible if we add hardware support and allow the PSM to act as a controller. In the diagram below, we see that a large data memory must be accessed by the PSM. Rather than directly address the memory, a hardware address counter that is controlled by the PSM is provided.



Registers can be used to compute start addresses from which data will be accessed (such as the beginning of text strings). This value is then loaded into the hardware address counter using a couple of output port locations. The hardware counter is then used to access the main data memory. As far as the PSM program is concerned, all reads and writes to the data memory occur at the same I/O port address. The added advantage of this technique is that the address counter can be made to increment automatically each time the PSM reads or writes information.

INNOVATIVE APPLICATIONS - These applications are innovative in their continued exploitation of the way in which a PSM is implemented inside the Xilinx FPGA. They build on the idea that there is now a software-programmable element inside a hardware-programmable device. The degree of flexibility that this offers is vast and is no longer limited to the larger and more costly devices. Now anyone with the most basic of Spartan-II evaluation boards may investigate these programmable options.

The most common observation is that the program for a PSM is stored in a block RAM. Typically, this is used as a ROM that is initialised by the configuration bit stream. This block RAM is, of course, RAM; therefore, the contents can be modified. The key factor here is that the memory is dual-port, which allows the PSM to read one port whilst the other is available to modify the code.



Although we could enter the scary world of self-modifying code, applications tend to be based on a completely new program being loaded into the memory for the PSM to execute. These may be the appropriate programs for handling different data types, protocols, or standards. This again means that the program space does not have to be very large and provides a method to support new protocols and standards in future.

Probably the most innovative concept is that of using the PSM as a sequencer. Each program becomes a one-shot event without the normal repeating loop. New execution sequences are then loaded into the program memory, possibly several hundred times per second. Since there would be so many different sequences to generate, and each sequence is by its very nature a sequential process, these sequences are much easier to develop in the software environment.

Closing comments

Thank you for following this series of articles -- I hope it has inspired you to either create your own programmable state machines or simply to use the ones that I have made available.

I have always found it interesting to study the implementation of a processor, as it consists of so many common digital logic building blocks. When memory, registers, multiplexers, adders, logical functions, and decoding logic can be efficiently implemented for a processor structure, it is a firm indication that similar techniques may be employed to implement virtually any digital function. It would be nice to think that you will also be able to exploit the Xilinx FPGA resources in similar ways in your own designs in future.

Although this is the last article of this series, please do continue to send me your e-mails to discuss this subject further, or simply to share your own PSM designs and applications of PicoBlaze (KCPSM) with me.

If you want to have a look at KCPSM (PicoBlaze), it is downloadable at the address below. (Full documentation and an assembler are also available for your use.) Over 1700 copies of KCPSM were downloaded from the web site in May, 2002.

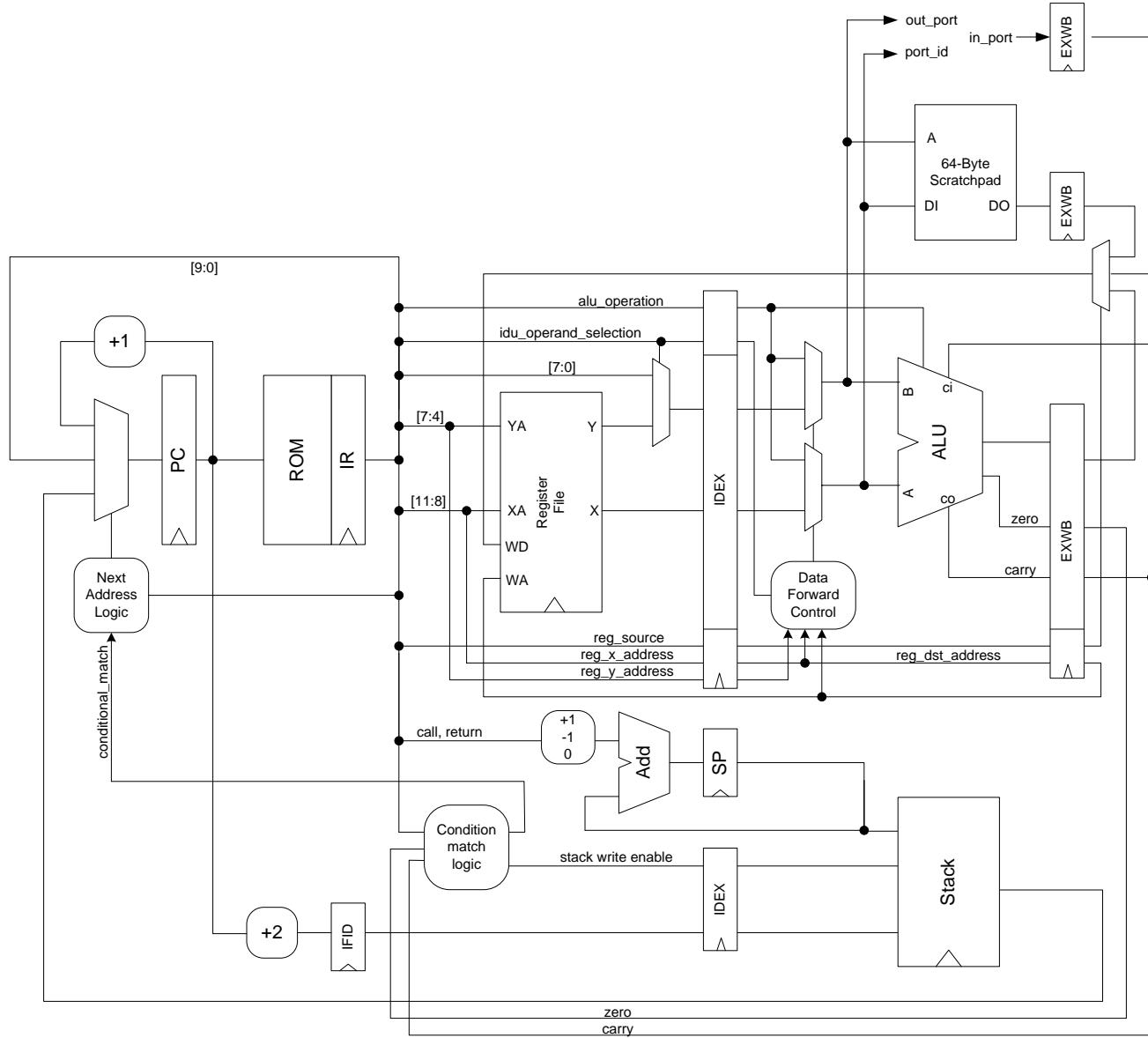
XAPP213 "8-bit Microcontroller for Virtex Devices"

<http://www.xilinx.com/xapp/xapp213.pdf>

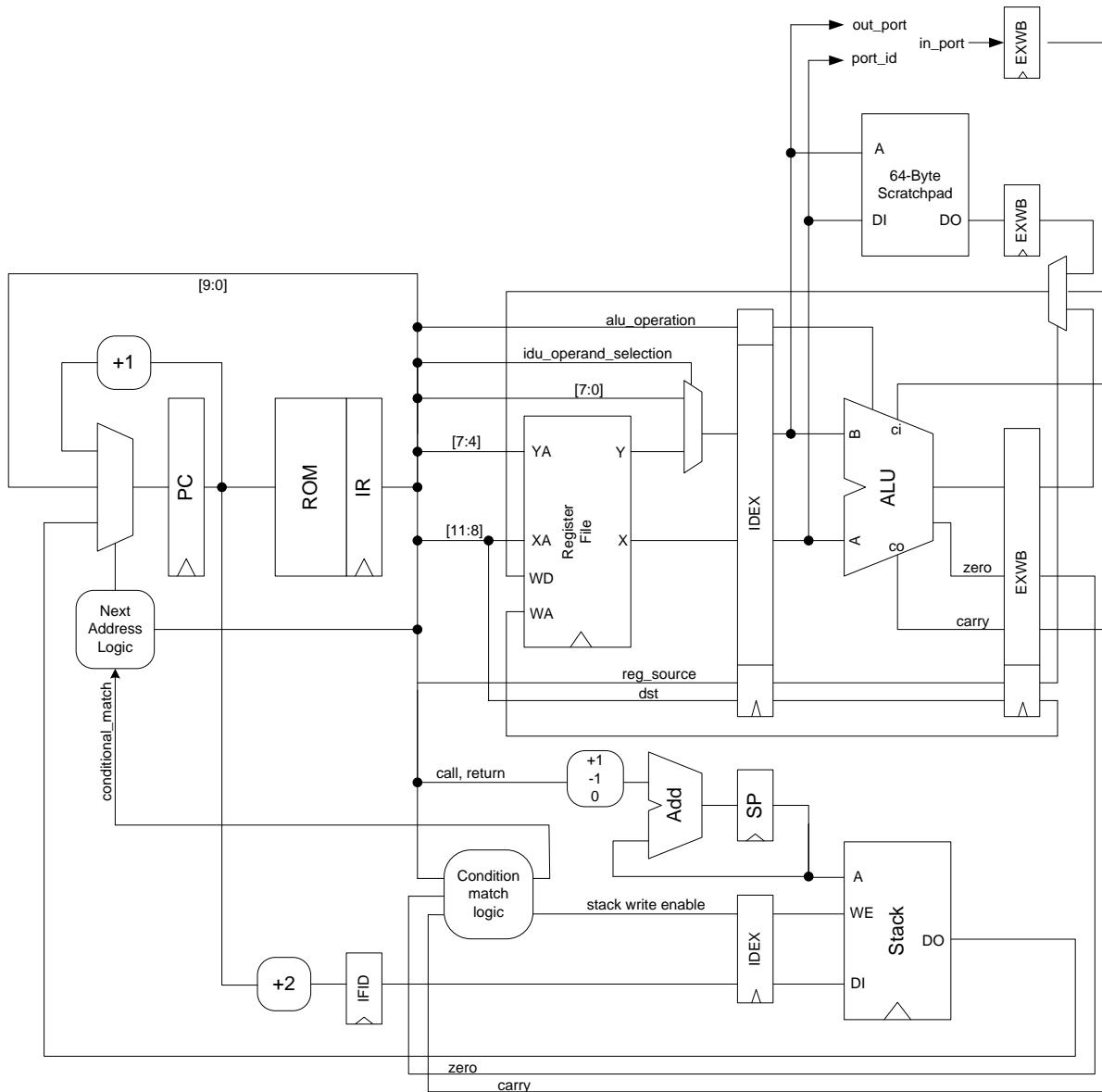
http://www.xilinx.com/ipcenter/processor_central/picoblaze/index.htm

This PSM is suitable for all Virtex, Virtex-E, and Spartan-II devices. If you would like a PSM specially tuned to the Virtex-II architecture, drop me an e-mail at ken.chapman@xilinx.com and I will be pleased to send it to you.

NOTE: Xilinx has recently given the KCPSM reference design the name "PicoBlaze," to indicate the complementary nature of the Programmable State Machines with the high performance 32-bit RISC soft processor called "MicroBlaze" that was released in October 2001. With PicoBlaze and MicroBlaze™, designers now can choose from a range of "right-sized" solutions, from 8 to 32 bits.



Note: Interrupt handling logic not shown



Note: Interrupt handling
logic not shown

PicoBlaze 8-bit Embedded Microcontroller User Guide

*for Spartan-3, Virtex-II, and
Virtex-II Pro FPGAs*

UG129 (v1.1.1) November 21, 2005

PicoBlazeTM

 XILINX®



"Xilinx" and the Xilinx logo shown above are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, and XC5210 are registered trademarks of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLNX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLNX, NanoBlaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, RocketIO, SelectIO, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Virtex-II EasyPath, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

All other trademarks are the property of their respective owners.

Xilinx, Inc. does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx, Inc. reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx, Inc. will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such 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 any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx, Inc. devices and products are protected under U.S. Patents. Other U.S. and foreign patents pending. Xilinx, Inc. does not represent that devices shown or products described herein are free from patent infringement or from any other third party right. Xilinx, Inc. assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx, Inc. will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

The contents of this manual are owned and copyrighted by Xilinx. Copyright 1994-2004 Xilinx, Inc. All Rights Reserved. Except as stated herein, none of the material may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Any unauthorized use of any material contained in this manual may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.

PicoBlaze 8-bit Embedded Microcontroller User Guide UG129 (v1.1.1) November 21, 2005

The following table shows the revision history for this document..

	Version	Revision
05/20/04	1.0	Initial Xilinx release.
06/10/04	1.1	Various minor corrections, updates, and enhancements throughout.

Limitations

Limited Warranty and Disclaimer

These designs are provided to you "as-is". Xilinx and its licensors make and you receive no warranties or conditions, express, implied, statutory or otherwise, and Xilinx specifically disclaims any implied warranties of merchantability, non-infringement, or fitness for a particular purpose. Xilinx does not warrant that the functions contained in these designs will meet your requirements, or that the operation of these designs will be uninterrupted or error free, or that defects in the Designs will be corrected. Furthermore, Xilinx does not warrant or make any representations regarding use or the results of the use of the designs in terms of correctness, accuracy, reliability, or otherwise.

Limitation of Liability

In no event will Xilinx or its licensors be liable for any loss of data, lost profits, cost or procurement of substitute goods or services, or for any special, incidental, consequential, or indirect damages arising from the use or operation of the designs or accompanying documentation, however caused and on any theory of liability. This limitation will apply even if Xilinx has been advised of the possibility of such damage. This limitation shall apply notwithstanding the failure of the essential purpose of any limited remedies herein.

Technical Support Limitations

This module is not supported by general Xilinx Technical support as an official Xilinx product. Please refer any issues initially to the provider of the module. The author will gratefully receive any issues or potential continued improvements of the PicoBlaze microcontroller.

Ken Chapman

Senior Staff Engineer -Spartan FPGA Applications Specialist

E-mail: picoblaze@xilinx.com

The author will also be pleased to hear from anyone using the PicoBlaze™ microcontroller with information about your application and how these macros have been useful.

Acknowledgments

Xilinx thanks the following individuals for their contribution to the PicoBlaze microcontroller cause:

- **Henk van Kampen, Mediatronix**
Developer of the pBlazIDE graphical, integrated development environment.
- **Prof. Dr.-Ing. Bernhard Lang, University of Applied Sciences, Osrambrueck, Germany**
Concept of using VHDL simulation variables to display disassembled op-code instructions.
- **Kris Chaplin, Xilinx Ltd.**
JTAG-based program loader, update function.



MAKE IT YOUR ASIC

About This Guide

The PicoBlaze™ embedded microcontroller is an efficient, cost-effective embedded processor core for Spartan-3, Virtex-II, and Virtex-II Pro FPGAs. This user guide describes the capabilities, features, and benefits of PicoBlaze hardware design and how to effectively use the PicoBlaze instruction set and tools to create software applications.

Guide Contents

This manual contains the following chapters:

- [Chapter 1, "Introduction,"](#) describes the features and functional blocks of the PicoBlaze microcontroller.
- [Chapter 2, "PicoBlaze Interface Signals,"](#) defines the PicoBlaze signals.
- [Chapter 3, "PicoBlaze Instruction Set,"](#) summarizes the instruction set of the PicoBlaze microcontrollers.
- [Chapter 4, "Interrupts,"](#) describes how the PicoBlaze microcontroller uses interrupts.
- [Chapter 5, "Scratchpad RAM,"](#) describes the 64-byte scratchpad RAM.
- [Chapter 6, "Input and Output Ports,"](#) describes the input and output ports supported by the PicoBlaze microcontroller.
- [Chapter 7, "Instruction Storage Configurations,"](#) provides several examples of instruction storage with the PicoBlaze microcontroller.
- [Chapter 8, "Performance,"](#) provides performance values for the PicoBlaze microcontroller.
- [Chapter 9, "Using the PicoBlaze Microcontroller in an FPGA Design,"](#) describes the design flow process with the PicoBlaze microcontroller.
- [Chapter 10, "PicoBlaze Development Tools,"](#) describes the available development tools.
- [Chapter 11, "Assembler Directives,"](#) describes the assembler directives that provide advanced control.
- [Chapter 12, "Simulating PicoBlaze Code,"](#) describes the tools that simulate PicoBlaze code.
- [Appendix A, "Related Materials and References,"](#) provides additional resources useful for the PicoBlaze microcontroller design.
- [Appendix B, "Example Program Templates,"](#) provides example KCPSM3 and pBlazIDE code templates for use in application programs.
- [Appendix C, "PicoBlaze Instruction Set and Event Reference,"](#) summarizes the PicoBlaze instructions and events in alphabetical order.
- [Appendix D, "Instruction Codes,"](#) provides the 18-bit instruction codes for all PicoBlaze instructions.
- [Appendix E, "Register and Scratchpad RAM Planning Worksheets,"](#) provides worksheets to use for the PicoBlaze microcontroller design.

Table of Contents

Preface: Limitations

Limited Warranty and Disclaimer	3
Limitation of Liability	3
Technical Support Limitations	3

Preface: Acknowledgments

About This Guide

Guide Contents	5
----------------------	---

Chapter 1: Introduction

PicoBlaze Microcontroller Features	13
PicoBlaze Microcontroller Functional Blocks	14
General-Purpose Registers	14
1,024-Instruction Program Store	14
Arithmetic Logic Unit (ALU)	15
Flags	15
64-Byte Scratchpad RAM	15
Input/Output	15
Program Counter (PC)	16
Program Flow Control	16
CALL/RETURN Stack	16
Interrupts	16
Reset	16
Why the PicoBlaze Microcontroller?	17
Why Use a Microcontroller within an FPGA?	17

Chapter 2: PicoBlaze Interface Signals

Chapter 3: PicoBlaze Instruction Set

Address Spaces	24
Processing Data	26
Logic Instructions	26
Bitwise AND, OR, XOR	26
Complement/Invert Register	27
Invert or Toggle Bit	27
Clear Register	27
Set Bit	27
Clear Bit	28
Arithmetic Instructions	28
ADD and ADDCY Add Instructions	28
SUB and SUBCY Subtract Instructions	29

Increment/Decrement	29
Negate.....	30
Multiplication	30
Division	32
No Operation (NOP)	33
Setting and Clearing CARRY Flag.....	34
Clear CARRY Flag	34
Set CARRY Flag	34
Test and Compare	34
Test	34
Compare	36
Shift and Rotate Instructions	36
Shift.....	36
Rotate	37
Moving Data	38
Program Flow Control	38
JUMP	39
CALL/RETURN.....	40

Chapter 4: Interrupts

Example Interrupt Flow	44
------------------------------	----

Chapter 5: Scratchpad RAM

Address Modes	47
Direct Addressing	47
Indirect Addressing	47
Implementing a Look-Up Table	48
Stack Operations	49
FIFO Operations	49

Chapter 6: Input and Output Ports

POR T_ID Port	51
INPUT Operations	52
Applications with Few Input Sources.....	54
READ_STROBE Interaction with FIFOs.....	54
OUTPUT Operations	55
Simple Output Structure for Few Output Destinations	56
Pipelining for Maximum Performance	58
Repartitioning the Design for Maximum Performance	60

Chapter 7: Instruction Storage Configurations

Standard Configuration – Single 1Kx18 Block RAM	61
Standard Configuration with UART or JTAG Programming Interface	62
Two PicoBlaze Microcontrollers Share a 1Kx18 Code Image	62
Two PicoBlaze Microcontrollers with Separate 512x18 Code Images in a Block RAM	63
Distributed ROM Instead of Block RAM	63

Chapter 8: Performance

Input Clock Frequency	65
Predicting Executing Performance	65

Chapter 9: Using the PicoBlaze Microcontroller in an FPGA Design

VHDL Design Flow	67
KCPSM3 Module	67
Connecting the Program ROM	68
Black Box Instantiation of KCPSM3 using KCPSM3.ngc	69
Generating the Program ROM using prog_rom.coe.....	69
Generating an ESC Schematic Symbol	69
Verilog Design Flow	69

Chapter 10: PicoBlaze Development Tools

KCPSM3.....	71
Assembler	71
Assembly Errors.....	72
Input and Output Files	72
Mediatronix pBlazIDE	73
Configuring pBlazIDE for the PicoBlaze Microcontroller	73
Importing KCPSM3 Code into pBlazIDE	74
Differences Between the KCPSM3 Assembler and pBlazIDE.....	75
Directives.....	75

Chapter 11: Assembler Directives

Locating Code at a Specific Address	77
Naming or Aliasing Registers	77
Defining Constants	78
Naming the Program ROM Output File.....	78
KCPSM3	78
pBlazIDE	78
Defining I/O Ports (pBlazIDE).....	78
Input Ports.....	79
Output Ports	79
Input/Output Ports	80

Chapter 12: Simulating PicoBlaze Code

Instruction Set Simulation with pBlazIDE	84
Simulator Control Buttons	85
Using the pBlazIDE Instruction Set Simulator with KCPSM3 Programs	86
Simulating FPGA Interaction with the pBlazIDE Instruction Set Simulator	86
Turbocharging Simulation using FPGAs!.....	87

Appendix A: Related Materials and References

Appendix B: Example Program Templates

KCPSM3 Syntax	91
pBlazIDE Syntax	92

Appendix C: PicoBlaze Instruction Set and Event Reference

ADD sX, Operand —Add Operand to Register sX.....	93
ADDCY sX, Operand —Add Operand to Register sX with Carry	94
AND sX, Operand — Logical Bitwise AND Register sX with Operand.....	95
CALL [Condition,] Address — Call Subroutine at Specified Address, Possibly with Conditions	96
COMPARE sX, Operand — Compare Operand with Register sX.....	97
DISABLE INTERRUPT — Disable External Interrupt Input.....	98
ENABLE INTERRUPT — Enable External Interrupt Input	99
FETCH sX, Operand — Read Scratchpad RAM Location to Register sX.....	99
INPUT sX, Operand — Set PORT_ID to Operand, Read value on IN_PORT into Register sX	100
INTERRUPT Event, When Enabled.....	101
JUMP [Condition,] Address — Jump to Specified Address, Possibly with Conditions	102
LOAD sX, Operand — Load Register sX with Operand.....	103
OR sX, Operand — Logical Bitwise OR Register sX with Operand	103
OUTPUT sX, Operand — Write Register sX Value to OUT_PORT, Set PORT_ID to Operand	104
RESET Event.....	105
RETURN [Condition] — Return from Subroutine Call, Possibly with Conditions	106
RETURNI [ENABLE/DISABLE] — Return from Interrupt Service Routine and Enable or Disable Interrupts.....	107
RL sX — Rotate Left Register sX	108
RR sX — Rotate Right Register sX	108
SL[0 1 X A] sX — Shift Left Register sX	109
SR[0 1 X A] sX — Shift Right Register sX.....	110
STORE sX, Operand — Write Register sX Value to Scratchpad RAM Location	112
SUB sX, Operand —Subtract Operand from Register sX.....	113
SUBCY sX, Operand —Subtract Operand from Register sX with Borrow	114
TEST sX, Operand — Test Bit Location in Register sX, Generate Odd Parity .	115
XOR sX, Operand — Logical Bitwise XOR Register sX with Operand.....	117

Appendix D: Instruction Codes

Appendix E: Register and Scratchpad RAM Planning Worksheets

Registers.....	123
Scratchpad RAM.....	124

Introduction

The PicoBlaze™ microcontroller is a compact, capable, and cost-effective fully embedded 8-bit RISC microcontroller core optimized for the Spartan™-3, Virtex™-II, and Virtex-II Pro™ FPGA families. The PicoBlaze microcontroller provides cost-efficient microcontroller-based control and simple data processing.

The PicoBlaze microcontroller is optimized for efficiency and low deployment cost. It occupies just 96 FPGA slices, or only 12.5% of an XC3S50 FPGA and a minuscule 0.3% of an XC3S5000 FPGA. In typical implementations, a single FPGA block RAM stores up to 1024 program instructions, which are automatically loaded during FPGA configuration. Even with such resource efficiency, the PicoBlaze microcontroller performs a respectable 44 to 100 million instructions per second (MIPS) depending on the target FPGA family and speed grade.

The PicoBlaze microcontroller core is totally embedded within the target FPGA and requires no external resources. The PicoBlaze microcontroller is extremely flexible. The basic functionality is easily extended and enhanced by connecting additional FPGA logic to the microcontroller's input and output ports.

The PicoBlaze microcontroller provides abundant, flexible I/O at much lower cost than off-the-shelf controllers. Similarly, the PicoBlaze peripheral set can be customized to meet the specific features, function, and cost requirements of the target application. Because the PicoBlaze microcontroller is delivered as synthesizable VHDL source code, the core is future-proof and can be migrated to future FPGA architectures, effectively eliminating product obsolescence fears. Being integrated within the FPGA, the PicoBlaze microcontroller reduces board space, design cost, and inventory.

The PicoBlaze FPC is supported by a suite of development tools including an assembler, a graphical integrated development environment (IDE), a graphical instruction set simulator, and VHDL source code and simulation models. Similarly, the PicoBlaze microcontroller is also supported in the Xilinx System Generator development environment.

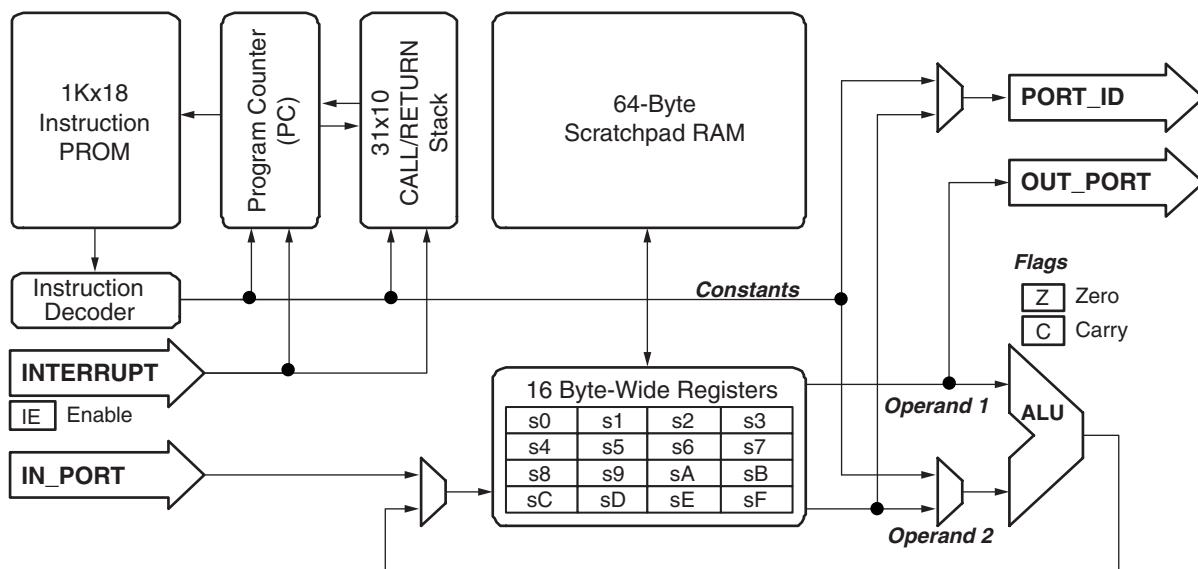
The various PicoBlaze code examples throughout this application note are written for the Xilinx KCPSM3 assembler. The Mediatronix pBlazIDE assembler has a code import function that reads the KCPSM3 syntax.

PicoBlaze Microcontroller Features

As shown in the block diagram in [Figure 1-1](#), the PicoBlaze microcontroller supports the following features:

- 16 byte-wide general-purpose data registers
- 1K instructions of programmable on-chip program store, automatically loaded during FPGA configuration

- Byte-wide Arithmetic Logic Unit (ALU) with CARRY and ZERO indicator flags
- 64-byte internal scratchpad RAM
- 256 input and 256 output ports for easy expansion and enhancement
- Automatic 31-location CALL/RETURN stack
- Predictable performance, always two clock cycles per instruction, up to 200 MHz or 100 MIPS in a Virtex-II Pro FPGA
- Fast interrupt response; worst-case 5 clock cycles
- Optimized for Xilinx Spartan-3, Virtex-II, and Virtex-II Pro FPGA architectures—just 96 slices and 0.5 to 1 block RAM
- Assembler, instruction-set simulator support



UG129_c1_01_051204

Figure 1-1: PicoBlaze Embedded Microcontroller Block Diagram

PicoBlaze Microcontroller Functional Blocks

General-Purpose Registers

The PicoBlaze microcontroller includes 16 byte-wide general-purpose registers, designated as registers **s0** through **sF**. For better program clarity, registers can be renamed using an assembler directive. All register operations are completely interchangeable; no registers are reserved for special tasks or have priority over any other register. There is no dedicated accumulator; each result is computed in a specified register.

1,024-Instruction Program Store

The PicoBlaze microcontroller executes up to 1,024 instructions from memory within the FPGA, typically from a single block RAM. Each PicoBlaze instruction is 18 bits wide. The instructions are compiled within the FPGA design and automatically loaded during the FPGA configuration process.

Other memory organizations are possible to accommodate more PicoBlaze controllers within a single FPGA or to enable interactive code updates without recompiling the FPGA design. See [Chapter 7, "Instruction Storage Configurations,"](#) for more information.

Arithmetic Logic Unit (ALU)

The byte-wide Arithmetic Logic Unit (ALU) performs all microcontroller calculations, including:

- basic arithmetic operations such as addition and subtraction
- bitwise logic operations such as AND, OR, and XOR
- arithmetic compare and bitwise test operations
- comprehensive shift and rotate operations

All operations are performed using an operand provided by any specified register (sX). The result is returned to the same specified register (sX). If an instruction requires a second operand, then the second operand is either a second register (sY) or an 8-bit immediate constant ($k8$).

Flags

ALU operations affect the ZERO and CARRY flags. The ZERO flag indicates when the result of the last operation resulted in zero. The CARRY flag indicates various conditions, depending on the last instruction executed.

The INTERRUPT_ENABLE flag enables the INTERRUPT input.

64-Byte Scratchpad RAM

The PicoBlaze microcontroller provides an internal general-purpose 64-byte scratchpad RAM, directly or indirectly addressable from the register file using the STORE and FETCH instructions.

The STORE instruction writes the contents of any of the 16 registers to any of the 64 RAM locations. The complementary FETCH instruction reads any of the 64 memory locations into any of the 16 registers. This allows a much greater number of variables to be held within the boundary of the processor and tends to reserve all of the I/O space for real inputs and output signals.

The six-bit scratchpad RAM address is specified either directly (ss) with an immediate constant, or indirectly using the contents of any of the 16 registers (sY). Only the lower six bits of the address are used; the address should not exceed the 00 - 3F range of the available memory.

Input/Output

The Input/Output ports extend the PicoBlaze microcontroller's capabilities and allow the microcontroller to connect to a custom peripheral set or to other FPGA logic. The PicoBlaze microcontroller supports up to 256 input ports and 256 output ports or a combination of input/output ports. The PORT_ID output provides the port address. During an INPUT operation, the PicoBlaze microcontroller reads data from the IN_PORT port to a specified register, sX . During an OUTPUT operation, the PicoBlaze microcontroller writes the contents of a specified register, sX , to the OUT_PORT port.

See [Chapter 6, "Input and Output Ports,"](#) for more information.

Program Counter (PC)

The Program Counter (PC) points to the next instruction to be executed. By default, the PC automatically increments to the next instruction location when executing an instruction. Only the JUMP, CALL, RETURN, and RETURNI instructions and the Interrupt and Reset Events modify the default behavior. The PC cannot be directly modified by the application code; computed jump instructions are not supported.

The 10-bit PC supports a maximum code space of 1,024 instructions (000 to 3FF hex). If the PC reaches the top of the memory at 3FF hex, it rolls over to location 000.

Program Flow Control

The default execution sequence of the program can be modified using conditional and non-conditional program flow control instructions.

The JUMP instructions specify an absolute address anywhere in the 1,024-instruction program space.

CALL and RETURN instructions provide subroutine facilities for commonly used sections of code. A CALL instruction specifies the absolute start address of a subroutine, while the return address is automatically preserved on the CALL/RETURN stack.

If the interrupt input is enabled, an Interrupt Event also preserves the address of the preempted instruction on the CALL/RETURN stack while the PC is loaded with the interrupt vector, 3FF hex. Use the RETURNI instruction instead of the RETURN instruction to return from the interrupt service routine (ISR).

CALL/RETURN Stack

The CALL/RETURN hardware stack stores up to 31 instruction addresses, enabling nested CALL sequences up to 31 levels deep. 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. Consequently, there are no instructions to control the stack or the stack pointer. No program memory is required for the stack.

Interrupts

The PicoBlaze microcontroller has an optional INTERRUPT input, allowing the PicoBlaze microcontroller to handle asynchronous external events. In this context, “asynchronous” relates to interrupts occurring at any time during an instruction cycle. However, recommended design practice is to synchronize all inputs to the PicoBlaze controller using the clock input.

The PicoBlaze microcontroller responds to interrupts quickly in just five clock cycles.

See [Chapter 4, “Interrupts,”](#) for more information.

Reset

The PicoBlaze microcontroller is automatically reset immediately after the FPGA configuration process completes. After configuration, the RESET input forces the processor into the initial state. The PC is reset to address 0, the flags are cleared, interrupts are disabled, and the CALL/RETURN stack is reset.

The data registers and scratchpad RAM are not affected by Reset.

See “[RESET Event](#)” in [Appendix C](#) for more information.

Why the PicoBlaze Microcontroller?

There are literally dozens of 8-bit microcontroller architectures and instruction sets. Modern FPGAs can efficiently implement practically any 8-bit microcontroller, and available FPGA soft cores support popular instruction sets such as the PIC, 8051, AVR, 6502, 8080, and Z80 microcontrollers. Why use the PicoBlaze microcontroller instead of a more popular instruction set?

The PicoBlaze microcontroller is specifically designed and optimized for the Spartan-3, Virtex-II, and Virtex-II Pro FPGA architectures. Its compact yet capable architecture consumes considerably less FPGA resources than comparable 8-bit microcontroller architectures within an FPGA. Furthermore, the PicoBlaze microcontroller is provided as a free, source-level VHDL file with royalty-free re-use within Xilinx FPGAs.

Some standalone microcontroller variants have a notorious reputation for becoming obsolete. Because it is delivered as VHDL source, the PicoBlaze microcontroller is immune to product obsolescence as the microcontroller can be retargeted to future generations of Xilinx FPGAs, exploiting future cost reductions and feature enhancements. Furthermore, the PicoBlaze microcontroller is expandable and extendable.

Before the advent of the PicoBlaze and MicroBlaze™ embedded processors, the microcontroller resided externally to the FPGA, limiting the connectivity to other FPGA functions and restricting overall interface performance. By contrast, the PicoBlaze microcontroller is fully embedded in the FPGA with flexible, extensive on-chip connectivity to other FPGA resources. Signals remain within the FPGA, improving overall performance. The PicoBlaze microcontroller reduces system cost because it is a single-chip solution, integrated within the FPGA and sometimes only occupying leftover FPGA resources.

The PicoBlaze microcontroller is resource efficient. Consequently, complex applications are sometimes best portioned across multiple PicoBlaze microcontrollers with each controller implementing a particular function, for example, keyboard and display control, or system management.

Why Use a Microcontroller within an FPGA?

Microcontrollers and FPGAs both successfully implement practically any digital logic function. However, each has unique advantages in cost, performance, and ease of use. Microcontrollers are well suited to control applications, especially with widely changing requirements. The FPGA resources required to implement the microcontroller are relatively constant. The same FPGA logic is re-used by the various microcontroller instructions, conserving resources. The program memory requirements grow with increasing complexity.

Programming control sequences or state machines in assembly code is often easier than creating similar structures in FPGA logic.

Microcontrollers are typically limited by performance. Each instruction executes sequentially. As an application increases in complexity, the number of instructions required to implement the application grows and system performance decreases accordingly. By contrast, performance in an FPGA is more flexible. For example, an algorithm can be implemented sequentially or completely in parallel, depending on the

performance requirements. A completely parallel implementation is faster but consumes more FPGA resources.

A microcontroller embedded within the FPGA provides the best of both worlds. The microcontroller implements non-timing crucial complex control functions while timing-critical or data path functions are best implemented using FPGA logic. For example, a microcontroller cannot respond to events much faster than a few microseconds. The FPGA logic can respond to multiple, simultaneous events in just a few to tens of nanoseconds. Conversely, a microcontroller is cost-effective and simple for performing format or protocol conversions.

Table 1-1: PicoBlaze Microcontroller Embedded within an FPGA Provides the Optimal Balance between Microcontroller and FPGA Solutions

	PicoBlaze Microcontroller	FPGA Logic
Strengths	<ul style="list-style-type: none"> Easy to program, excellent for control and state machine applications Resource requirements remain constant with increasing complexity Re-uses logic resources, excellent for lower-performance functions 	<ul style="list-style-type: none"> Significantly higher performance Excellent at parallel operations Sequential vs. parallel implementation trade-offs optimize performance or cost Fast response to multiple, simultaneous inputs
Weaknesses	<ul style="list-style-type: none"> Executes sequentially Performance degrades with increasing complexity Program memory requirements increase with increasing complexity Slower response to simultaneous inputs 	<ul style="list-style-type: none"> Control and state machine applications more difficult to program Logic resources grow with increasing complexity

PicoBlaze Interface Signals

The top-level interface signals to the PicoBlaze microcontroller appear in [Figure 2-1](#) and are described in [Table 2-1](#). [Figure 7-1](#) provides additional detail on the internal structure of the PicoBlaze controller.

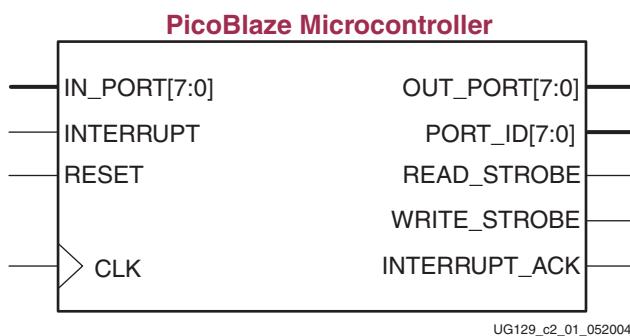


Figure 2-1: PicoBlaze Interface Connections

Table 2-1: PicoBlaze Interface Signal Descriptions

Signal	Direction	Description
IN_PORT[7:0]	Input	Input Data Port: Present valid input data on this port during an INPUT instruction. The data is captured on the rising edge of CLK.
INTERRUPT	Input	Interrupt Input: If the INTERRUPT_ENABLE flag is set by the application code, generate an INTERRUPT Event by asserting this input High for at least two CLK cycles. If the INTERRUPT_ENABLE flag is cleared, this input is ignored.
RESET	Input	Reset Input: To reset the PicoBlaze microcontroller and to generate a RESET Event, assert this input High for at least one CLK cycle. A Reset Event is automatically generated immediately following FPGA configuration.
CLK	Input	Clock Input: The frequency may range from DC to the maximum operating frequency reported by the Xilinx ISE development software. All PicoBlaze synchronous elements are clocked from the rising clock edge. There are no clock duty-cycle requirements beyond the minimum pulse width requirements of the FPGA.
OUT_PORT[7:0]	Output	Output Data Port: Output data appears on this port for two CLK cycles during an OUTPUT instruction. Capture output data within the FPGA at the rising CLK edge when WRITE_STROBE is High.

Table 2-1: PicoBlaze Interface Signal Descriptions (*Continued*)

Signal	Direction	Description
PORT_ID[7:0]	Output	Port Address: The I/O port address appears on this port for two CLK cycles during an INPUT or OUTPUT instruction.
READ_STROBE	Output	Read Strobe: When asserted High, this signal indicates that input data on the IN_PORT[7:0] port was captured to the specified data register during an INPUT instruction. This signal is asserted on the second CLK cycle of the two-cycle INPUT instruction. This signal is typically used to acknowledge read operations from FIFOs.
WRITE_STROBE	Output	Write Strobe: When asserted High, this signal validates the output data on the OUT_PORT[7:0] port during an OUTPUT instruction. This signal is asserted on the second CLK cycle of the two-cycle OUTPUT instruction. Capture output data within the FPGA on the rising CLK edge when WRITE_STROBE is High.
INTERRUPT_ACK	Output	Interrupt Acknowledge: When asserted High, this signal acknowledges that an INTERRUPT Event occurred. This signal is asserted during the second CLK cycle of the two-cycle INTERRUPT Event. This signal is optionally used to clear the source of the INTERRUPT input.

PicoBlaze Instruction Set

Table 3-1 summarizes the entire PicoBlaze instruction set, which appears alphabetically. Instructions are listed using the KCPSM3 syntax. If different, the pBlazIDE syntax appears in parentheses. Each instruction includes an overview description, a functional description, and how the ZERO and CARRY flags are affected. For more details on each instruction, see [Appendix C, “PicoBlaze Instruction Set and Event Reference.”](#)

Table 3-1: PicoBlaze Instruction Set (alphabetical listing)

Instruction	Description	Function	ZERO	CARRY
ADD sX, kk	Add register sX with literal kk	$sX \leftarrow sX + kk$?	?
ADD sX, sY	Add register sX with register sY	$sX \leftarrow sX + sY$?	?
ADDCY sX, kk (ADDC)	Add register sX with literal kk with CARRY bit	$sX \leftarrow sX + kk + \text{CARRY}$?	?
ADDCY sX, sY (ADDC)	Add register sX with register sY with CARRY bit	$sX \leftarrow sX + sY + \text{CARRY}$?	?
AND sX, kk	Bitwise AND register sX with literal kk	$sX \leftarrow sX \text{ AND } kk$?	0
AND sX, sY	Bitwise AND register sX with register sY	$sX \leftarrow sX \text{ AND } sY$?	0
CALL aaa	Unconditionally call subroutine at aaa	$\text{TOS} \leftarrow \text{PC}$ $\text{PC} \leftarrow \text{aaa}$	-	-
CALL C, aaa	If CARRY flag set, call subroutine at aaa	If CARRY=1, { $\text{TOS} \leftarrow \text{PC}$, $\text{PC} \leftarrow \text{aaa}$ }	-	-
CALL NC, aaa	If CARRY flag not set, call subroutine at aaa	If CARRY=0, { $\text{TOS} \leftarrow \text{PC}$, $\text{PC} \leftarrow \text{aaa}$ }	-	-
CALL NZ, aaa	If ZERO flag not set, call subroutine at aaa	If ZERO=0, { $\text{TOS} \leftarrow \text{PC}$, $\text{PC} \leftarrow \text{aaa}$ }	-	-
CALL Z, aaa	If ZERO flag set, call subroutine at aaa	If ZERO=1, { $\text{TOS} \leftarrow \text{PC}$, $\text{PC} \leftarrow \text{aaa}$ }	-	-
COMPARE sX, kk (COMP)	Compare register sX with literal kk. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If $sX=kk$, $\text{ZERO} \leftarrow 1$ If $sX < kk$, $\text{CARRY} \leftarrow 1$?	?
COMPARE sX, sY (COMP)	Compare register sX with register sY. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If $sX=sY$, $\text{ZERO} \leftarrow 1$ If $sX < sY$, $\text{CARRY} \leftarrow 1$?	?
DISABLE INTERRUPT (DINT)	Disable interrupt input	$\text{INTERRUPT_ENABLE} \leftarrow 0$	-	-

Table 3-1: PicoBlaze Instruction Set (alphabetical listing)

Instruction	Description	Function	ZERO	CARRY
ENABLE INTERRUPT (EINT)	Enable interrupt input	INTERRUPT_ENABLE \leftarrow 1	-	-
Interrupt Event	Asynchronous interrupt input. Preserve flags and PC. Clear INTERRUPT_ENABLE flag. Jump to interrupt vector at address 3FF.	Preserved ZERO \leftarrow ZERO Preserved CARRY \leftarrow CARRY INTERRUPT_ENABLE \leftarrow 0 TOS \leftarrow PC PC \leftarrow 3FF	-	-
FETCH sX, (sY) (FETCH sX, sY)	Read scratchpad RAM location pointed to by register sY into register sX	sX \leftarrow RAM[(sY)]	-	-
FETCH sX, ss	Read scratchpad RAM location ss into register sX	sX \leftarrow RAM[ss]	-	-
INPUT sX, (sY) (IN sX, sY)	Read value on input port location pointed to by register sY into register sX	PORT_ID \leftarrow sY sX \leftarrow IN_PORT	-	-
INPUT sX, pp (IN)	Read value on input port location pp into register sX	PORT_ID \leftarrow pp sX \leftarrow IN_PORT	-	-
JUMP aaa	Unconditionally jump to aaa	PC \leftarrow aaa	-	-
JUMP C, aaa	If CARRY flag set, jump to aaa	If CARRY=1, PC \leftarrow aaa	-	-
JUMP NC, aaa	If CARRY flag not set, jump to aaa	If CARRY=0, PC \leftarrow aaa	-	-
JUMP NZ, aaa	If ZERO flag not set, jump to aaa	If ZERO=0, PC \leftarrow aaa	-	-
JUMP Z, aaa	If ZERO flag set, jump to aaa	If ZERO=1, PC \leftarrow aaa	-	-
LOAD sX, kk	Load register sX with literal kk	sX \leftarrow kk	-	-
LOAD sX, sY	Load register sX with register sY	sX \leftarrow sY	-	-
OR sX, kk	Bitwise OR register sX with literal kk	sX \leftarrow sX OR kk	?	0
OR sX, sY	Bitwise OR register sX with register sY	sX \leftarrow sX OR sY	?	0
OUTPUT sX, (sY) (OUT sX, sY)	Write register sX to output port location pointed to by register sY	PORT_ID \leftarrow sY OUT_PORT \leftarrow sX	-	-
OUTPUT sX, pp (OUT sX, pp)	Write register sX to output port location pp	PORT_ID \leftarrow pp OUT_PORT \leftarrow sX	-	-
RETURN (RET)	Unconditionally return from subroutine	PC \leftarrow TOS+1	-	-
RETURN C (RET C)	If CARRY flag set, return from subroutine	If CARRY=1, PC \leftarrow TOS+1	-	-
RETURN NC (RET NC)	If CARRY flag not set, return from subroutine	If CARRY=0, PC \leftarrow TOS+1	-	-
RETURN NZ (RET NZ)	If ZERO flag not set, return from subroutine	If ZERO=0, PC \leftarrow TOS+1	-	-
RETURN Z (RET Z)	If ZERO flag set, return from subroutine	If ZERO=1, PC \leftarrow TOS+1	-	-

Table 3-1: PicoBlaze Instruction Set (alphabetical listing)

Instruction	Description	Function	ZERO	CARRY
RETURNI DISABLE (RETI DISABLE)	Return from interrupt service routine. Interrupt remains disabled.	$PC \leftarrow TOS$ $ZERO \leftarrow$ Preserved ZERO $CARRY \leftarrow$ Preserved CARRY $INTERRUPT_ENABLE \leftarrow 0$?	?
RETURNI ENABLE (RETI ENABLE)	Return from interrupt service routine. Re-enable interrupt.	$PC \leftarrow TOS$ $ZERO \leftarrow$ Preserved ZERO $CARRY \leftarrow$ Preserved CARRY $INTERRUPT_ENABLE \leftarrow 1$?	?
RL sX	Rotate register sX left	$sX \leftarrow \{sX[6:0],sX[7]\}$ $CARRY \leftarrow sX[7]$?	?
RR sX	Rotate register sX right	$sX \leftarrow \{sX[0],sX[7:1]\}$ $CARRY \leftarrow sX[0]$?	?
SL0 sX	Shift register sX left, zero fill	$sX \leftarrow \{sX[6:0],0\}$ $CARRY \leftarrow sX[7]$?	?
SL1 sX	Shift register sX left, one fill	$sX \leftarrow \{sX[6:0],1\}$ $CARRY \leftarrow sX[7]$	0	?
SLA sX	Shift register sX left through all bits, including CARRY	$sX \leftarrow \{sX[6:0],CARRY\}$ $CARRY \leftarrow sX[7]$?	?
SLX sX	Shift register sX left. Bit sX[0] is unaffected.	$sX \leftarrow \{sX[6:0],sX[0]\}$ $CARRY \leftarrow sX[7]$?	?
SR0 sX	Shift register sX right, zero fill	$sX \leftarrow \{0,sX[7:1]\}$ $CARRY \leftarrow sX[0]$?	?
SR1 sX	Shift register sX right, one fill	$sX \leftarrow \{1,sX[7:1]\}$ $CARRY \leftarrow sX[0]$	0	?
SRA sX	Shift register sX right through all bits, including CARRY	$sX \leftarrow \{CARRY,sX[7:1]\}$ $CARRY \leftarrow sX[0]$?	?
SRX sX	Arithmetic shift register sX right. Sign extend sX. Bit sX[7] Is unaffected.	$sX \leftarrow \{sX[7],sX[7:1]\}$ $CARRY \leftarrow sX[0]$?	?
STORE sX, (sY) (STORE sX, sY)	Write register sX to scratchpad RAM location pointed to by register sY	$RAM[(sY)] \leftarrow sX$	-	-
STORE sX, ss	Write register sX to scratchpad RAM location ss	$RAM[ss] \leftarrow sX$	-	-
SUB sX, kk	Subtract literal kk from register sX	$sX \leftarrow sX - kk$?	?
SUB sX, sY	Subtract register sY from register sX	$sX \leftarrow sX - sY$?	?
SUBCY sX, kk (SUBC)	Subtract literal kk from register sX with CARRY (borrow)	$sX \leftarrow sX - kk - CARRY$?	?
SUBCY sX, sY (SUBC)	Subtract register sY from register sX with CARRY (borrow)	$sX \leftarrow sX - sY - CARRY$?	?

Table 3-1: PicoBlaze Instruction Set (alphabetical listing)

Instruction	Description	Function	ZERO	CARRY
TEST sX, kk	Test bits in register sX against literal kk. Update CARRY and ZERO flags. Registers are unaffected.	If (sX AND kk) = 0, ZERO \leftarrow 1 CARRY \leftarrow odd parity of (sX AND kk)	?	?
TEST sX, sY	Test bits in register sX against register sY. Update CARRY and ZERO flags. Registers are unaffected.	If (sX AND sY) = 0, ZERO \leftarrow 1 CARRY \leftarrow odd parity of (sX AND kk)	?	?
XOR sX, kk	Bitwise XOR register sX with literal kk	sX \leftarrow sX XOR kk	?	0
XOR sX, sY	Bitwise XOR register sX with register sY	sX \leftarrow sX XOR sY	?	0

sX = One of 16 possible register locations ranging from s0 through sF or specified as a literal

sY = One of 16 possible register locations ranging from s0 through sF or specified as a literal

aaa = 10-bit address, specified either as a literal or a three-digit hexadecimal value ranging from 000 to 3FF or a labeled location

kk = 8-bit immediate constant, specified either as a literal or a two-digit hexadecimal value ranging from 00 to FF or specified as a literal

pp = 8-bit port address, specified either as a literal or a two-digit hexadecimal value ranging from 00 to FF or specified as a literal

ss = 6-bit scratchpad RAM address, specified either as a literal or a two-digit hexadecimal value ranging from 00 to 3F or specified as a literal

RAM[n] = Contents of scratchpad RAM at location n

TOS = Value stored at Top Of Stack

Address Spaces

As shown in [Table 3-2](#), the PicoBlaze microcontroller has five distinct address spaces. Specific instructions operate on each of the address spaces.

Table 3-2: PicoBlaze Address Spaces and Related Instructions

Address Space	Size (Depth x Width)	Addressing Modes	Instructions that Operate on Address Space
Instruction	1Kx18	Direct	<ul style="list-style-type: none"> • JUMP • CALL • RETURN • RETURNI • INTERRUPT event • RESET event <p>All others increment the PC to the next location</p>
Register File	16x8	Direct	<ul style="list-style-type: none"> • LOAD • AND • OR • XOR • TEST (read only) • ADD • ADDCY • SUB • SUBCY • COMPARE (read only) • SR0 • SR1 • SRX • SRA • RR • SL0 • SL1 • SLX • SLA • RL • INPUT • OUTPUT (read only) • STORE (read only) • FETCH
Scratchpad RAM	64x8	Direct Indirect	<ul style="list-style-type: none"> • STORE • FETCH
I/O	256x8	Direct Indirect	<ul style="list-style-type: none"> • INPUT • OUTPUT
CALL/RETURN Stack	31x10	N/A	<ul style="list-style-type: none"> • CALL • Enabled INTERRUPT event • RETURN • RETURNI • RESET event

Processing Data

All data processing instructions operate on any of the 16 general-purpose registers. Only the data processing instructions modify the ZERO or CARRY flags as appropriate for the instruction. The data processing instructions consists of the following types:

- Logic instructions
- Arithmetic instructions
- Test and Compare instructions
- Shift and Rotate instructions

Logic Instructions

The logic instructions perform a bitwise logical AND, OR, or XOR between two operands. The first operand is a register location. The second operand is either a register location or a literal constant. Besides performing pure AND, OR, and XOR operations, the logic instructions provide a means to:

- complement or invert a register
- clear a register
- set or clear specific bits within a register

Bitwise AND, OR, XOR

All logic instructions are bitwise operations. The AND operation, illustrated in Figure 3-1, shows that corresponding bit locations in both operands are logically ANDed together and the result is placed back into register sX. If the resulting value in register sX is zero, then the ZERO flag is set. The CARRY flag is always cleared by a logic instruction.

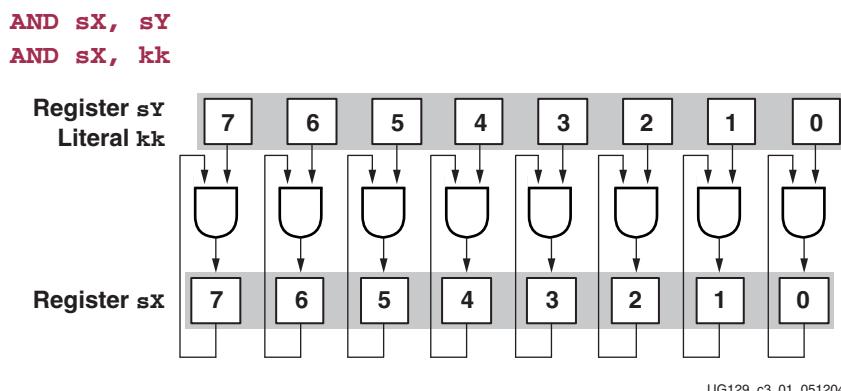


Figure 3-1: Bitwise AND Instruction

The OR and XOR instructions are similar to the AND instruction illustrated in Figure 3-1 except that they perform an OR or XOR logical operation, respectively.

See also:

- “AND sX, Operand — Logical Bitwise AND Register sX with Operand,” page 95
- “OR sX, Operand — Logical Bitwise OR Register sX with Operand,” page 104
- “XOR sX, Operand — Logical Bitwise XOR Register sX with Operand,” page 118

Complement/Invert Register

The PicoBlaze microcontroller does not have a specific instruction to invert individual bits within register `sX`. However, the `XOR sX, FF` instruction performs the equivalent operation, as shown in [Figure 3-2](#).



If reading this document in Adobe Acrobat,
use the Select Text tool to select code snippets,
then copy and paste the text into your text editor.

```
complement:  
; XOR sX, FF invert all bits in register sX, same as one's complement  
  
LOAD s0, AA ; load register s0 = 10101010  
XOR s0, FF ; invert contents s0 = 01010101
```

Figure 3-2: Complementing a Register Value

Invert or Toggle Bit

The PicoBlaze microcontroller does not have a specific instruction to invert or toggle an individual bit or bits within a specific register. However, the `XOR` instruction performs the equivalent operation. XORing register `sX` with a bit mask inverts or toggles specific bits, as shown in [Figure 3-3](#). A '1' in the bit mask inverts or toggles the corresponding bit in register `sX`. A '0' in the bit mask leaves the corresponding bit unchanged.

```
toggle_bit:  
; XOR sX, <bit_mask>  
  
XOR s0, 01 ; toggle the least-significant bit in register sX
```

Figure 3-3: Inverting an Individual Bit Location

Clear Register

The PicoBlaze microcontroller does not have a specific instruction to clear a specific register. However, the `XOR sX, sX` instruction performs the equivalent operation. XORing register `sX` with itself clears registers `sX` and sets the ZERO flag, as shown in [Figure 3-4](#).

```
XOR sX, sX ; clear register sX, set ZERO flag
```

Figure 3-4: Clearing a Register and Setting the ZERO Flag

The `LOAD sX, 00` instruction also clears register `sX`, but it does not affect the ZERO flag, as shown in [Figure 3-5](#).

```
LOAD sX, 00 ; clear register sX, ZERO flag unaffected
```

Figure 3-5: Clearing a Register without Modifying the ZERO Flag

Set Bit

The PicoBlaze microcontroller does not have a specific instruction to set an individual bit or bits within a specific register. However, the `OR` instruction performs the equivalent

operation. ORing register sX with a bit mask sets specific bits, as shown in [Figure 3-6](#). A '1' in the bit mask sets the corresponding bit in register sX . A '0' in the bit mask leaves the corresponding bit unchanged.

```
set_bit:
;   OR sX, <bit_mask>

OR s0, 01 ; set bit 0 of register s0
```

Figure 3-6: 16-Setting a Bit Location

Clear Bit

The PicoBlaze microcontroller does not have a specific instruction to clear an individual bit or bits within a specific register. However, the AND instruction performs the equivalent operation. ANDing register sX with a bit mask clears specific bits, as shown in [Figure 3-7](#). A '0' in the bit mask clears the corresponding bit in register sX . A '1' in the bit mask leaves the corresponding bit unchanged.

```
clear_bit:
;   AND sX, <bit_mask>

AND s0, FE ; clear bit 0 of register s0
```

Figure 3-7: Clearing a Bit Location

Arithmetic Instructions

The PicoBlaze microcontroller provides basic byte-wide addition and subtraction instructions. Combinations of instructions perform multi-byte arithmetic plus multiplication and division operations. If the end application requires significant arithmetic performance, consider using the 32-bit MicroBlaze RISC processor core for Xilinx FPGAs (see Reference 4).

ADD and ADDCY Add Instructions

The PicoBlaze microcontroller provides two add instructions, ADD and ADDCY, that compute the sum of two 8-bit operands, either without or with CARRY, respectively. The first operand is a register location. The second operand is either a register location or a literal constant. The resulting operation affects both the CARRY and ZERO flags. If the resulting sum is greater than 255, then the CARRY flag is set. If the resulting sum is either 0 or 256 (register sX is zero with CARRY set), then the ZERO flag is set.

The ADDCY instruction is an add operation with carry. If the CARRY flag is set, then ADDCY adds an additional one to the resulting sum.

The ADDCY instruction is commonly used in multi-byte addition. [Figure 3-8](#) demonstrates a subroutine that adds two 16-bit integers and produces a 16-bit result. The upper byte of each 16-bit value is labeled as MSB for most-significant byte; the lower byte of each 16-bit value is labeled LSB for least-significant byte.

```

ADD16:
    NAMEREG s0, a_lsb ; rename register s0 as "a_lsb"
    NAMEREG s1, a_msb ; rename register s1 as "a_msb"
    NAMEREG s2, b_lsb ; rename register s2 as "b_lsb"
    NAMEREG s3, b_msb ; rename register s3 as "b_msb"

    ADD a_lsb, b_lsb ; add LSBs, keep result in a_lsb
    ADDCY a_msb, b_msb ; add MSBs, keep result in a_msb
    RETURN

```

Figure 3-8: 16-Bit Addition Using ADD and ADDCY Instructions

See also:

- “[ADD sX, Operand —Add Operand to Register sX](#),” page 93
- “[ADDCY sX, Operand —Add Operand to Register sX with Carry](#),” page 94

SUB and SUBCY Subtract Instructions

The PicoBlaze microcontroller provides two subtract instructions, SUB and SUBCY, that compute the difference of two 8-bit operands, either without or with CARRY (borrow), respectively. The CARRY flag indicates if the subtract operation generates a borrow condition. The first operand is a register location. The second operand is either a register location or a literal constant. The resulting operation affects both the CARRY and ZERO flags. If the resulting difference is less than 0, then the CARRY flag is set. If the resulting difference is 0 or -256, then the ZERO flag is set.

The SUBCY instruction is a subtract operation with borrow. If the CARRY flag is set, then SUBCY subtracts an additional one from the resulting difference.

The SUBCY instruction is commonly used in multi-byte subtraction. [Figure 3-9](#) demonstrates a subroutine that subtracts two 16-bit integers and produces a 16-bit difference. The upper byte of each 16-bit value is labeled as MSB for most-significant byte; the lower byte of each 16-bit value is labeled LSB for least-significant byte.

```

SUB16:
    NAMEREG s0, a_lsb ; rename register s0 as "a_lsb"
    NAMEREG s1, a_msb ; rename register s1 as "a_msb"
    NAMEREG s2, b_lsb ; rename register s2 as "b_lsb"
    NAMEREG s3, b_msb ; rename register s3 as "b_lsb"

    SUB a_lsb, b_lsb ; subtract LSBs, keep result in a_lsb
    SUBCY a_msb, b_msb ; subtract MSBs, keep result in a_msb
    RETURN

```

Figure 3-9: 16-Bit Subtraction Using SUB and SUBCY Instructions

See also:

- “[SUB sX, Operand —Subtract Operand from Register sX](#),” page 114
- “[SUBCY sX, Operand —Subtract Operand from Register sX with Borrow](#),” page 115

Increment/Decrement

The PicoBlaze microcontroller does not have a dedicated increment or decrement instruction. However, adding or subtracting one using the ADD or SUB instructions provides the equivalent operation, as shown in [Figure 3-10](#).

```
ADD sX,01 ; increment register sX
SUB sX,01 ; decrement register sX
```

Figure 3-10: Incrementing and Decrementing a Register

If incrementing or decrementing a multi-register value—i.e., a 16-bit value—perform the operation using multiple instructions. Incrementing or decrementing a multi-byte value requires using the add or subtract instructions with carry, as shown in Figure 3-11.

```
inc_16:
; increment low byte
ADD lo_byte,01

; increment high byte only if CARRY bit set when incrementing low byte
ADDCY hi_byte,00
```

Figure 3-11: Incrementing a 16-bit Value

Negate

The PicoBlaze microcontroller does not have a dedicated instruction to negate a register value, taking the two's complement. However, the instructions in Figure 3-12 provide the equivalent operation.

```
Negate:
; invert all bits in the register performing a one's complement
XOR sX,FF
; add one to sX
ADD sX,01
RETURN
```

Figure 3-12: Destructive Negate (2's Complement) Function Overwrites Original Value

Another possible implementation that does not overwrite the value appears in Figure 3-13.

```
Negate:
NAMEREG sY, value
NAMEREG sX, complement
; Clear 'complement' to zero
LOAD complement, 00
; subtract value from 0 to create two's complement
SUB complement, value
RETURN
```

Figure 3-13: Non-destructive Negate Function Preserves Original Value

Multiplication

The PicoBlaze microcontroller core does not have a dedicated hardware multiplier. However, the PicoBlaze microcontroller performs multiplication using the available arithmetic and shift instructions. Figure 3-14 demonstrates an 8-bit by 8-bit multiply routine that produces a 16-bit multiplier product in 50 to 57 instruction cycles, or 100 to 114 clock cycles. By contrast, the 8051 microcontroller performs the same multiplication in eight instruction cycles or 96 clock cycles on a the standard 12-cycle 8051.

```

; Multiplier Routine (8-bit x 8-bit = 16-bit product)
; =====
; Shift and add algorithm
;
mult_8x8:
    NAMEREG s0, multiplicand      ; preserved
    NAMEREG s1, multiplier       ; preserved
    NAMEREG s2, bit_mask         ; modified
    NAMEREG s3, result_msb      ; most-significant byte (MSB) of result,
                                  ; modified
    NAMEREG s4, result_lsb      ; least-significant byte (LSB) of result,
                                  ; modified
;
    LOAD bit_mask, 01           ; start with least-significant bit (lsb)
    LOAD result_msb, 00          ; clear product MSB
    LOAD result_lsb, 00          ; clear product LSB (not required)
;
    ; loop through all bits in multiplier
mult_loop: TEST multiplier, bit_mask ; check if bit is set
    JUMP Z, no_add             ; if bit is not set, skip addition
;
    ADD result_msb, multiplicand ; addition only occurs in MSB
;
no_add: SRA result_msb          ; shift MSB right, CARRY into bit 7,
                                ; lsb into CARRY
    SRA result_lsb            ; shift LSB right,
                                ; lsb from result_msb into bit 7
;
    SL0 bit_mask               ; shift bit_mask left to examine
                                ; next bit in multiplier
;
    JUMP NZ, mult_loop         ; if all bit examined, then bit_mask = 0,
                                ; loop if not 0
    RETURN                   ; multiplier result now available in
                                ; result_msb and result_lsb

```

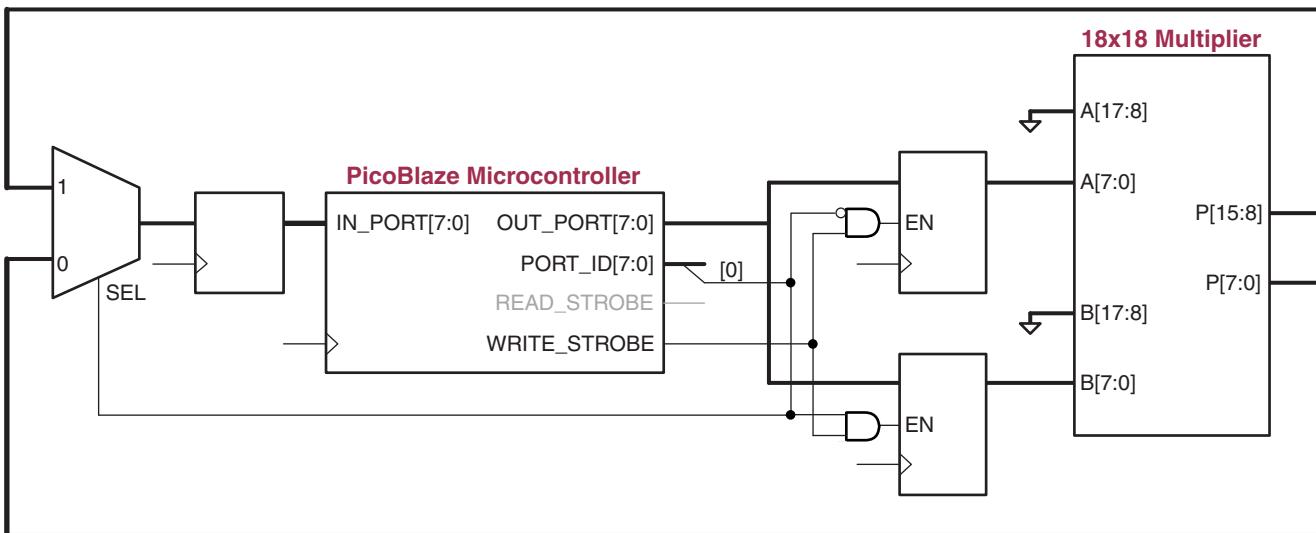
Figure 3-14: 8-bit by 8-bit Multiply Routine Produces a 16-bit Product

If multiplication performance is important to the application, connect one of the FPGA's 18x18 hardware multipliers the PicoBlaze I/O ports, as shown in [Figure 3-15](#). The hardware multiplier computes the 16-bit result in less than one instruction cycle.

[Figure 3-16](#) shows the routine required to multiply two 8-bit values using the hardware multiplier. This same technique can be expanded to multiply two 16-bit values to produce a 32-bit result. This example also illustrates how to use FPGA logic attached to the PicoBlaze microcontroller to accelerate algorithms.



If reading this document in Adobe Acrobat,
use the Select Text tool to select code snippets,
then copy and paste the text into your text editor.



UG129_c3_02_052004

Figure 3-15: 8-bit by 8-bit Hardware Multiplier Using the FPGA’s 18x18 Multipliers

```

; Multiplier Routine (8-bit x 8-bit = 16-bit product)
; =====
; Connects to embedded 18x18 Hardware Multiplier via ports
;
mult_8x8io:
    NAMEREG s0, multiplicand      ; preserved
    NAMEREG s1, multiplier        ; preserved
    NAMEREG s3, result_msb       ; most-significant byte (MSB) of result, modified
    NAMEREG s4, result_lsb       ; least-significant byte (LSB) of result, modified
;
; Define the port ID numbers as constants for better clarity
CONSTANT multiplier_lsb, 00
CONSTANT multiplier_msb, 01
;
; Output multiplicand and multiplier to FPGA registers connected to the inputs of
; the embedded multiplier.
OUTPUT multiplicand, multiplier_lsb
OUTPUT multiplier, multiplier_msb
;
; Input the resulting product from the embedded multiplier.
INPUT result_lsb, multiplier_lsb
INPUT result_msb, multiplier_msb
;
RETURN                      ; multiplier result now available in result_msb
                            ; and result_lsb

```

Figure 3-16: 8-bit by 8-bit Multiply Routine Using Hardware Multiplier

Division

The PicoBlaze microcontroller core does not have a dedicated hardware divider. However, the PicoBlaze microcontroller performs division using the available arithmetic and shift instructions. Figure 3-17 demonstrates a subroutine that divides an unsigned 8-bit number by another unsigned 8-bit number to produce an 8-bit quotient and an 8-bit remainder in 60 to 74 instruction cycles, or 120 to 144 clock cycles.

```

; Divide Routine (8-bit / 8-bit = 8-bit result, remainder)
; =====
; Shift and subtract algorithm
;
div_8by8:
    NAMEREG s0, dividend      ; preserved
    NAMEREG s1, divisor       ; preserved
    NAMEREG s2, quotient      ; preserved
    NAMEREG s3, remainder     ; modified
    NAMEREG s4, bit_mask      ; used to test bits in dividend,
                                ; one-hot encoded, modified
;
    LOAD remainder, 00         ; clear remainder
    LOAD bit_mask, 80          ; start with most-significant bit (msb)
div_loop:
    TEST dividend, bit_mask   ; test bit, set CARRY if bit is '1'
    SLA remainder             ; shift CARRY into lsb of remainder
    SLO quotient              ; shift quotient left (multiply by 2)
;
    COMPARE remainder, divisor; is remainder > divisor?
    JUMP C, no_sub            ; if divisor is greater, continue to next bit
    SUB remainder, divisor     ; if remainder > divisor, then subtract
    ADD quotient, 01           ; add one to quotient
no_sub:
    SR0 bit_mask               ; shift to examine next bit position
    JUMP NZ, div_loop          ; if bit_mask=0, then all bits examined
                                ; otherwise, examine next bit
RETURN

```

Figure 3-17: 8-bit Divided by 8-bit Routine



If reading this document in Adobe Acrobat,
use the Select Text tool to select code snippets,
then copy and paste the text into your text editor.

No Operation (NOP)

The PicoBlaze instruction set does not have a specific NOP instruction. Typically, a NOP instruction is completely benign, does not affect register contents or flags, and performs no operation other than requiring an instruction cycle to execute. A NOP instruction is therefore sometimes useful to balance code trees for more predictable execution timing.

There are a few possible implementations of an equivalent NOP operation, as shown in [Figure 3-18](#) and [Figure 3-19](#). Loading a register with itself does not affect the register value or the status flags.

```

nop:
    LOAD SX, SX

```

Figure 3-18: Loading a Register with Itself Acts as a NOP Instruction

A similar NOP technique is to simply jump to the next instruction, which is equivalent to the default program flow. The JUMP instruction consumes an instruction cycle (two clock cycles) without affecting register contents.

```
JUMP next  
next: <next instruction>
```

Figure 3-19: Alternative NOP Method Using JUMP Instructions

Setting and Clearing CARRY Flag

Sometimes, application programs need to specifically set or clear the CARRY flag, as shown in the following examples.

Clear CARRY Flag

ANDing a register with itself clears the CARRY flag without affecting the register contents, as shown in Figure 3-20.

```
clear_carry_bit:  
    AND SX, SX ; register SX unaffected, CARRY flag cleared
```

Figure 3-20: ANDing a Register with Itself Clears the CARRY Flag

Set CARRY Flag

There are various methods for setting the CARRY flag, one of which appears in Figure 3-21. Generally, these methods affect a register location.

```
set_carry:  
    LOAD SX, 00  
    COMPARE SX, 01 ; set CARRY flag and reset ZERO flag
```

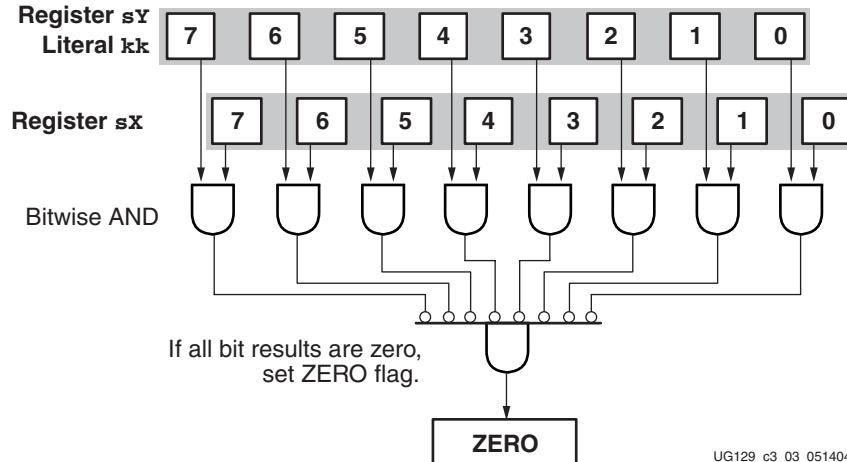
Figure 3-21: Example Operation that Sets the CARRY Flag

Test and Compare

The PicoBlaze microcontroller introduces two new instructions not available on previous PicoBlaze variants. The PicoBlaze microcontroller provides the ability to test individual bits within a register and the ability to compare a register value against another register or an immediate constant. The TEST or COMPARE instructions only affect the ZERO and CARRY flags; neither instruction affects register contents.

Test

The TEST instruction performs bit testing via a bitwise logical AND operation between two operands. Unlike the AND instruction, only the ZERO and CARRY flags are affected; no registers are modified. The ZERO flag is set if all the bitwise AND results are Low, as shown in Figure 3-22.



UG129_c3_03_051404

Figure 3-22: The TEST Instruction Affects the ZERO Flag

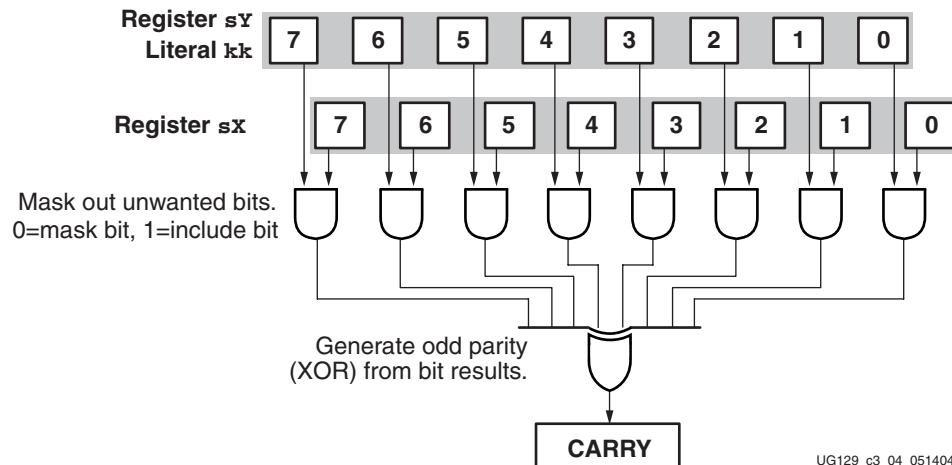
Each bit of register **sX** is logically ANDed with either the contents of register **sY** or a literal constant, **kk**. The operation sets the **ZERO** flag if the result of all bitwise AND operations is zero.

If the second operand contains a single '1' bit, then the CARRY flag tests if the corresponding bit in register **sX** is '1' as shown in the example in [Figure 3-23](#).

```
LOAD s0, 05 ; s0 = 00000101
TEST s0, 04 ; mask = 00000100
; CARRY = 1, ZERO = 0
```

Figure 3-23: Generate Parity for a Register Using the TEST Instruction

In a broader application, the CARRY bit generates the odd parity for the included bits in register **sX**, as shown in [Figure 3-24](#). The second operand acts as a mask. If a bit in the second operand is '0', then the corresponding bit in register **sX** is not included in the generated parity value. If a bit in the second operand is '1', then the corresponding bit in register **sX** is included in the final parity value.



UG129_c3_04_051404

Figure 3-24: The TEST Instruction Affects the CARRY Flag

The example in [Figure 3-25](#) demonstrates how to generate parity for all eight bits in a register.

```
generate_parity:
    TEST sX, FF ; include all bits in parity generation
```

Figure 3-25: Generate Parity for a Register Using the TEST Instruction

See also “[TEST sX, Operand — Test Bit Location in Register sX, Generate Odd Parity](#),” page 116.

Compare

The COMPARE instruction performs an 8-bit subtraction of two operands but only affects the ZERO and CARRY flags, as shown in [Table 3-3](#). No registers are modified.

The ZERO flag is set when both input operands are identical. When set, the CARRY flag indicates that the second operand is greater than the first operand.

Table 3-3: COMPARE Instruction Flag Operations

Flag	When Flag=0	When Flag=1
ZERO	Operand_1 ≠ Operand_2	Operand_1 = Operand_2
CARRY	Operand_1 ≥ Operand_2	Operand_1 < Operand_2

See also “[COMPARE sX, Operand — Compare Operand with Register sX](#),” page 97.

Shift and Rotate Instructions

Shift

The PicoBlaze microcontroller supports a rich set of shift instructions, summarized in [Table 3-4](#), that modify the contents of a single register. All shift instructions affect the CARRY and ZERO flags.

The SL0 sX instruction shift the contents of register sX left by one bit position. The most-significant bit, bit 7, shifts into the CARRY flag. The least-significant bit position is filled with a ‘0’. The SR0 instruction is similar except the least-significant bit, bit 0, shifts into the CARRY flag and the most-significant bit is filled with a ‘0’.

The SL1 and SR1 shift instructions are similar to SL0 and SR0 except that the empty bit location is filled with a ‘1’. The ZERO flag is always ‘0’ when using SL1 and SR1 because there is always a ‘1’ shifted into the affected register, making the register non-zero.

The SRX sX instruction performs an arithmetic shift right operation and sign extends register sX, preserving the sign bit. The most-significant bit, bit 7, is unaffected during the shift operation and is copied back into bit 7. The SLX sX instruction is similar but shifts the register sX contents to the left, replicating bit 0 and filling the register with the bit 0 value.

The SLA sX instruction left shifts the contents of register sX through the CARRY bit, the CARRY bit feeding back into the least-significant bit, bit 0, of register sX. The SRA sX instruction is similar to SLA but with a right shift.

Table 3-4: PicoBlaze Shift Instructions

	Shift Left		Shift Right
SL0	Shift Left with '0' fill. CARRY Register sX 	SR0	Shift Right with '0' fill. Register sX CARRY 
SL1	Shift Left with '1' fill. CARRY Register sX 	SR1	Shift Right with '1' fill. Register sX CARRY 
SLX	Shift Left, eXtend bit 0. CARRY Register sX 	SRX	Shift Right, sign eXtend. Register sX CARRY 
SLA	Shift Left through All bits, including CARRY. CARRY Register sX 	SRA	Shift Right through All bits, including CARRY. Register sX CARRY 

See also:

- “**SL[0 | 1 | X | A] sX** — Shift Left Register sX,” page 110
- “**SR[0 | 1 | X | A] sX** — Shift Right Register sX,” page 111

Rotate

The rotate instructions, shown in Table 3-5, rotate the contents of the specified register left or right. The **RL sX** instruction shifts the contents of register sX left with the most-significant bit, bit 7, feeding the least-significant bit, bit 0. The most-significant bit, bit 7, also shifts into the CARRY flag. The **RR sX** instruction is similar but shifts the contents of register sX to the right and copies the least-significant bit, bit 0, into the CARRY flag.

Table 3-5: PicoBlaze Rotate Instructions

	Rotate Left		Rotate Right
RL	CARRY Register sX 	RR	Register sX CARRY 

See also:

- “**RL sX** — Rotate Left Register sX,” page 109
- “**RR sX** — Rotate Right Register sX,” page 109

Moving Data

Data movement between various resources is an essential microcontroller function. Figure 3-26 shows the various PicoBlaze instructions to move data.

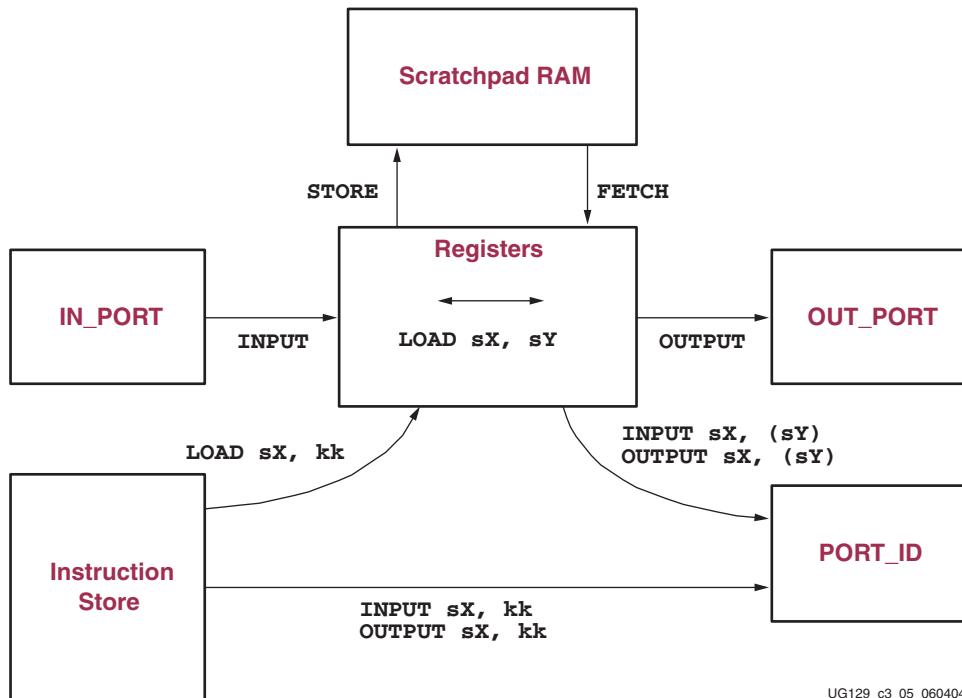


Figure 3-26: Data Movement Instructions

The **LOAD SX, SY** instruction moves data between two PicoBlaze registers; Register **sX** receives the data. The **LOAD SX, kk** instruction loads an immediate byte-wide constant into the specified register. See also “[LOAD SX, Operand — Load Register sX with Operand](#),” page 104. The **LOAD** instructions do not affect the CARRY or ZERO flags.

The **STORE** and **FETCH** instructions move data between the register file and the scratchpad RAM. See “[Chapter 5, ‘Scratchpad RAM’](#)” for more information.

During an **INPUT** operation, data from the **IN_PORT** input port is always read to one of the registers. Likewise, during an **OUTPUT** instruction, data written to the **OUT_PORT** output port always originates from one of the registers. The input/output address, provided on the **PORT_ID** output, originates either from one of the registers or as a literal constant from the instruction store. See [Chapter 6, “Input and Output Ports,”](#) for more information.

Program Flow Control

The Program Counter (PC) points to the next instruction to be executed and directly controls the PicoBlaze program flow. By default, the PicoBlaze microcontroller proceeds to the next instruction in the instruction store ($PC=PC+1$). The PC cannot be directly accessed. However, three different PicoBlaze instructions, **JUMP** and the **CALL/RETURN** pair potentially modify the default program flow by loading the PC with a different value. An enabled interrupt event also modifies program flow but this case is described in [Chapter 4, “Interrupts.”](#) Likewise, a Reset Event resets the PC to zero, restarting program execution.

The JUMP, CALL, and RETURN instructions are all conditionally executed, depending if a condition is specified and specifically whether the CARRY or ZERO flags are set or cleared. **Table 3-6** summarizes the possible conditions. The condition is specified as an instruction operand. The instruction is unconditionally executed if no condition is specified.

Table 3-6: Instruction Conditional Execution

Condition	Description
<none>	Always true. Execute instruction unconditionally.
C	CARRY = 1. Execute instruction if CARRY flag is set.
NC	CARRY = 0. Execute instruction if CARRY flag is cleared.
Z	ZERO = 1. Execute instruction if ZERO flag is set.
NZ	ZERO = 0. Execute instruction if ZERO flag is cleared.

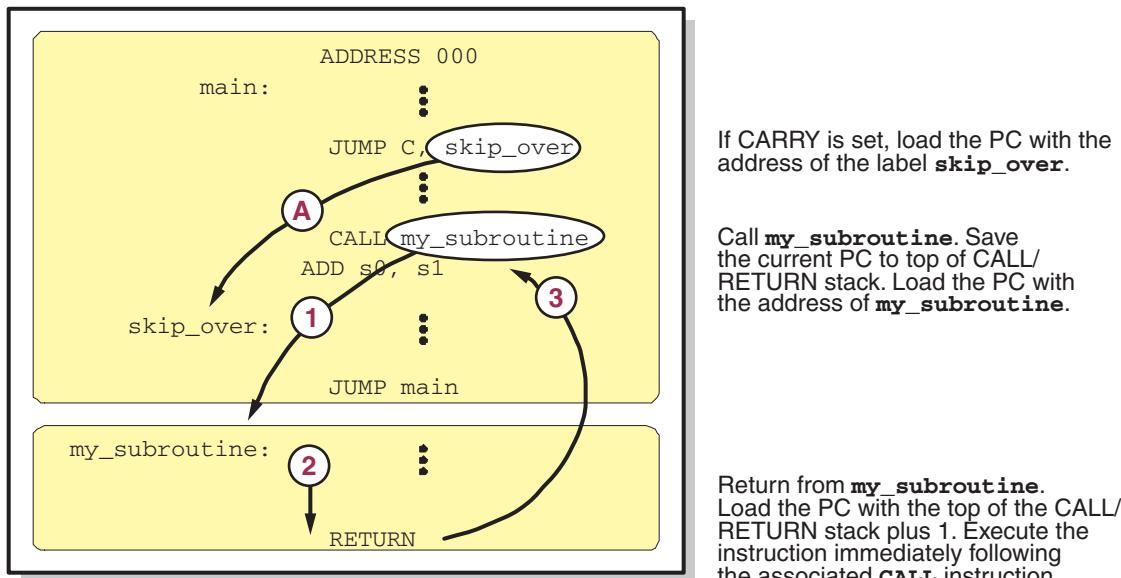
JUMP

The JUMP instruction is conditional and executes only if the specified condition, listed in **Table 3-6**, is met. If the condition is false, then the conditional JUMP instruction has no effect other than requiring two clock cycles to execute. No registers or flags are affected.

If the conditional JUMP instruction is executed, then the PC is loaded with the address of the specified label, which is computed and assigned by the assembler. The PicoBlaze processor then executes the instruction at the specified label.

The JUMP instruction does not interact with the CALL/RETURN stack.

Arrow 'A' in [Figure 3-27](#) illustrates the program flow during a JUMP instruction. When the PicoBlaze microcontroller executes the JUMP C, skip_over instruction, it first checks if the CARRY bit is set. If the CARRY bit is set, then the address of the skip_over label is loaded into the PC. The PicoBlaze microcontroller then jumps to and executes the instruction located at that address. Program flow continues normally from that point.



UG129_c3_06_051404

Figure 3-27: Example JUMP and CALL/RETURN Procedures

The JUMP instruction does not affect the ZERO and CARRY flags. All jumps are absolute; there are no relative jumps. Likewise, computed jumps are not supported.

See also “[JUMP \[Condition,\] Address — Jump to Specified Address, Possibly with Conditions](#),” page 103.

CALL/RETURN

The CALL instruction differs from the JUMP instruction in that program flow temporarily jumps to a subroutine function and then returns to the instruction following the CALL instruction, as shown in [Figure 3-27](#). The CALL instruction is conditional and executes only if the specified condition, listed in [Table 3-6](#), is met. If the condition is false, then the conditional CALL instruction has no effect other than requiring two clock cycles to execute. No registers or flags are affected.

If the conditional CALL instruction is executed, then the current PC value is pushed on top of the CALL/RETURN stack. The address of the specified label, which is computed and assigned by the assembler, is loaded into the PC. The PicoBlaze microcontroller then executes the instruction at the specified label. See arrow ‘1’ in [Figure 3-27](#).

The PicoBlaze microcontroller continues executing instructions in the subroutine call until it encounters a RETURN instruction. See arrow ‘2’ in [Figure 3-27](#).

Every CALL instruction should have a corresponding RETURN instruction. The RETURN instruction is also conditional and executes only if the specified condition, listed in [Table 3-6](#), is met. The RETURN instruction terminates the subroutine call, pops the top of the CALL/RETURN stack, increments the value, and loads the value into the PC, which returns the program flow to the instruction immediately following the original CALL instruction. See arrow ‘3’ in [Figure 3-27](#).

If the conditional CALL instruction is executed, the ZERO and CARRY flags are potentially modified by the instructions within the called subroutine, but not directly by the CALL or

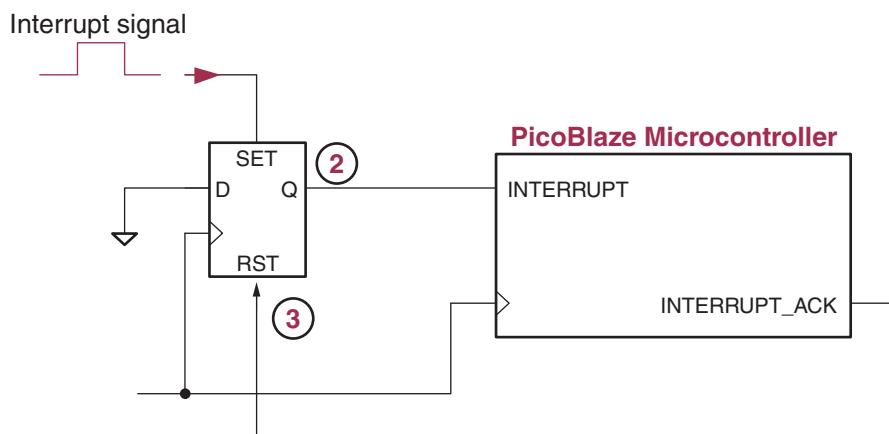
RETURN instructions themselves. If the CALL instruction is not executed, then the flags are unaffected.

See also:

- “CALL [Condition,] Address — Call Subroutine at Specified Address, Possibly with Conditions,” page 96“
- “RETURN [Condition] — Return from Subroutine Call, Possibly with Conditions,” page 107

Interrupts

The PicoBlaze processor provides a single interrupt input signal. If the application requires multiple interrupt signals, combine the signals using simple FPGA logic to form a single INTERRUPT input signal. After reset, the INTERRUPT input is disabled and must be enabled via the `ENABLE_INTERRUPT` instruction. To disable interrupts at any point in the program, issue a `DISABLE_INTERRUPT` instruction.



UG129_c4_01_060404

Figure 4-1: Simple Interrupt Logic

Once enabled, the INTERRUPT input signal must be applied for at least two clock cycles to guarantee that it is recognized, generating an INTERRUPT Event.

An active interrupt forces the PicoBlaze processor to immediately execute the `CALL 3FF` instruction immediately after completing the instruction currently executing. The `CALL 3FF` instruction is a subroutine call to the last program memory location. The instruction in the last location defines how the application code should handle the interrupt. Typically, the instruction at location 3FF is a jump location to an interrupt service routine (ISR).

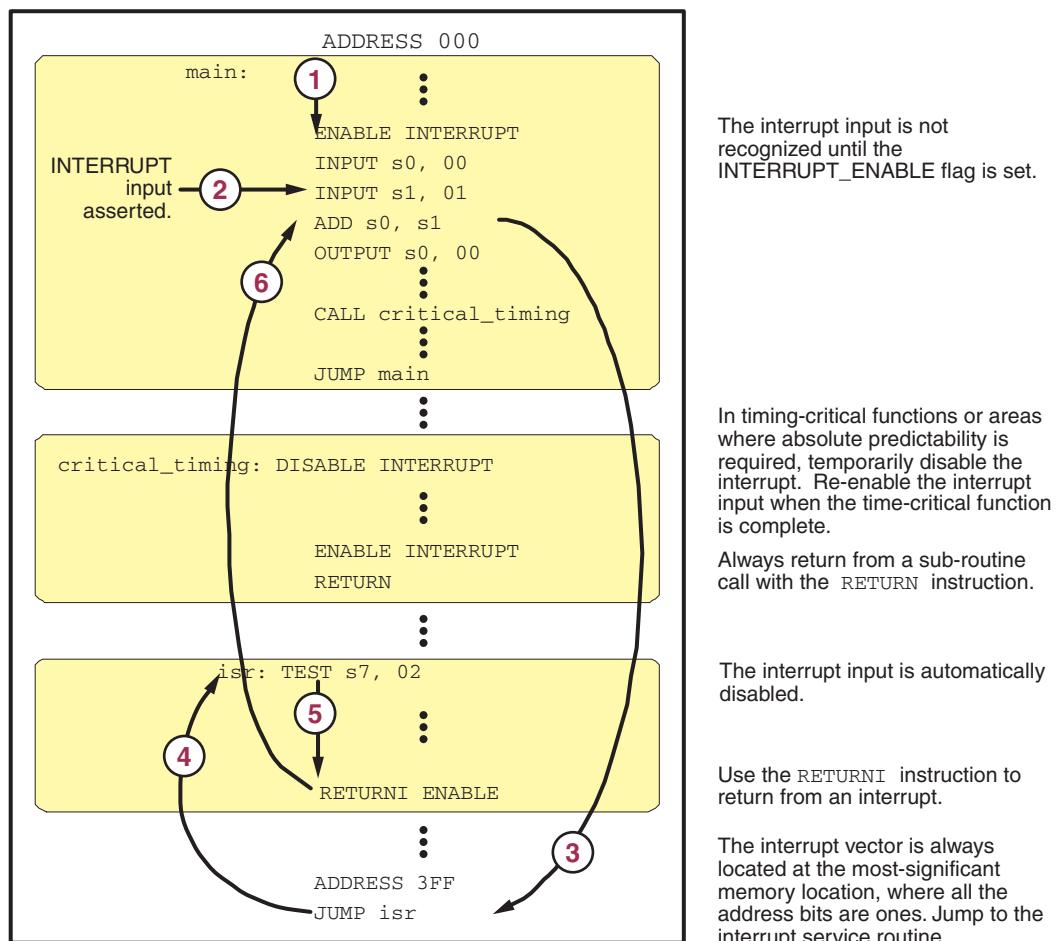
The PicoBlaze microcontroller automatically performs other functions. The interrupt process preserves the current ZERO and CARRY flag contents and disables any further interrupts. Likewise, the current program counter (PC) value is pushed onto the CALL/RETURN stack. Interrupts must remain disabled throughout the interrupt handling process.

As shown in [Figure 4-3](#), the PicoBlaze microcontroller asserts its `INTERRUPT_ACK` signal during the second cycle of the two-cycle Interrupt Event to indicate that the interrupt was recognized. The `INTERRUPT_ACK` signal may be used to clear external interrupts, as shown in [Figure 4-1](#).

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. When the RETURNI instruction is executed, the PC values saved onto the CALL/RETURN stack is automatically reloaded to the PC register. Likewise, the ZERO and CARRY flags are restored and program flow returns to the instruction following the instruction where the interrupt occurred.

If the application does not require an interrupt, tie the INTERRUPT signal Low. Consequently, all 1,024 instruction locations are available.

Example Interrupt Flow



UG129_c4_02_051404

Figure 4-2: Example Interrupt Flow

Figure 4-2 shows an example program flow during an interrupt event.

1. By default, the INTERRUPT input is disabled. The ENABLE_INTERRUPT instruction must execute before the interrupt is recognized.
2. In this example, interrupts are enabled and the PicoBlaze microcontroller is executing the INPUT s1, 01 instruction. Simultaneously to executing this instruction, an interrupt arrives on the INTERRUPT input. The PicoBlaze microcontroller does not act on the interrupt until it finishes executing the INPUT s1, 01 instruction.

3. The PicoBlaze microcontroller recognizes the interrupt and preempts the ADD s0, s1 instruction. The current PC, which points to the ADD s0 s1 instruction, is pushed onto the CALL/RETURN stack. Likewise, the ZERO and CARRY flags are preserved. Furthermore, the INTERRUPT_ENABLE flag is cleared disabling any further interrupts. Finally, the PC is loaded with all ones (3FF) and the PicoBlaze microcontroller performs an interrupt service routine call to the last location in the instruction store. If using a 1Kx18 block RAM for instruction store, the last location is 3FF. If using a smaller instruction store, then the interrupt vector is still located in the last instruction location. The PicoBlaze microcontroller also asserts the INTERRUPT_ACK output, indicating that the interrupt is being acknowledged.
4. The interrupt vector is always located in the last location in the instruction store. In this example, the program jumps to the interrupt service routine (ISR) via the JUMP isr instruction.
5. When completed, exit the interrupt service routine (ISR) using the special RETURNI instruction. Do not use the RETURN instruction, which is used with normal subroutine calls. The RETURNI ENABLE instruction returns from the interrupt service routine and re-enables the INTERRUPT input, which was automatically disabled when the interrupt was recognized. Using RETURNI DISABLE also returns from the interrupt service routine but leaves the INTERRUPT input disabled.
6. The RETURNI instruction restores the preserved ZERO and CARRY flags saved during Step (3). Likewise, the RETURNI instruction pops the top of the CALL/RETURN stack into the PC, which causes the PicoBlaze microcontroller to resume program executing the instruction that was preempted by the interrupt, ADD s0, s1 in this example.

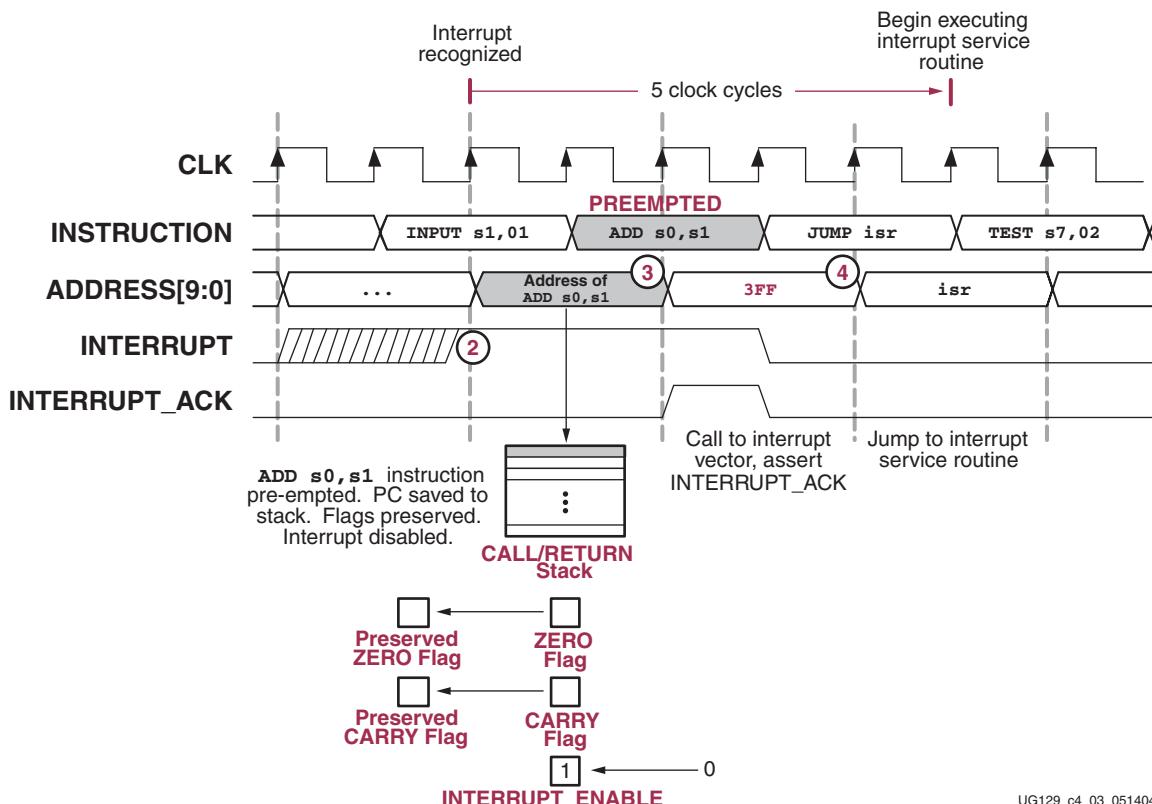


Figure 4-3: Interrupt Timing Diagram

UG129_c4_03_051404

[Figure 4-3](#) shows the same interrupt procedure but as a timing diagram. With the interrupt enabled, the INTERRUPT input is recognized at Step (2), the same clock cycle where the ADDRESS bus changes value. The address for the instruction ADD s0, s1 appears on the ADDRESS bus and is pushed onto the CALL/RETURN stack. Simultaneously, the interrupt is disabled and the ZERO and CARRY flags are preserved. The ADD s0, s1 instruction is preempted and does not yet execute. Instead, the PicoBlaze microcontroller performs a call to the interrupt vector at location 0x3FF.

An interrupt is undesirable in timing-critical procedures or when predictable timing is a must. Temporarily disable the INTERRUPT input using the DISABLE_INTERRUPT instruction, as demonstrated in the `critical_timing` subroutine in [Figure 4-2](#). Once the critical procedure completes, re-enable the INTERRUPT input with the ENABLE_INTERRUPT instruction.

Scratchpad RAM

The PicoBlaze microcontroller contains a 64-byte scratchpad RAM. Two instructions, STORE and FETCH, move data between any data register and the scratchpad RAM. Both direct and indirect addressing are supported. The scratchpad RAM is only supported on PicoBlaze microcontrollers for Spartan-3, Virtex-II, and Virtex-II Pro FPGAs.

The scratchpad RAM is unaffected by a RESET Event.

Address Modes

The STORE and FETCH instructions support both direct and indirect addressing modes to access scratchpad RAM data.

Direct Addressing

An immediate constant value directly addresses a specific scratchpad RAM location. In the example in [Figure 5-1](#), register sX directly writes to and reads from scratchpad RAM location 04.

```
scratchpad_transfers:  
    STORE sX, 04 ; Write register sX to RAM location 04  
    FETCH sX, 04 ; Read RAM location 04 into register sX
```

Figure 5-1: Directly Addressing Scratchpad RAM Locations

Indirect Addressing

Using indirect address, the actual RAM address is the value contained in a specified register. Whereas direct addressing requires the RAM address to be known before assembly, indirect addressing provides additional program flexibility. The application code can compute or modify the RAM address based on other program data. The code in [Figure 5-2](#), for example, initializes all the scratchpad RAM locations to 0 using a simple loop.

```

NAMEREG s0, ram_data
NAMEREG s1, ram_address

CONSTANT ram_locations, 40      ; there are 64 locations
CONSTANT initial_value, 00       ; initialize to zero

LOAD ram_data, initial_value   ; load initial value
LOAD ram_address, ram_locations ; fill from top to bottom

ram_fill: SUB ram_address, 01      ; decrement address
STORE ram_data, (ram_address)   ; initialize location
JUMP NZ, ram_fill              ; if not address 0, goto
                                ; ram_fill

```

Figure 5-2: Indirect Addressing Initializes All of RAM with a Simple Subroutine

Implementing a Look-Up Table

The next few examples demonstrate both the flexibility of the scratchpad RAM and indirect addressing. The example code in Figure 5-3 uses Scratchpad RAM as a look-up table (LUT) to convert four binary inputs to the equivalent hexadecimal character display on a 7-segment LED. The code reads four external switches, resulting in a binary value between 0000 and 1111. The PicoBlaze microcontroller converts each four-bit switch value into the equivalent hexadecimal character as displayed on a 7-segment LED. The scratchpad RAM holds the LED output patterns in the first 16 locations. The input switch value is the address input to the RAM.

```

CONSTANT switches, 00           ; read switch values at port 0
CONSTANT LEDs, 01             ; write 7-seg LED at port 1
; Define 7-segment LED pattern {dp,g,f,e,d,c,b,a}
CONSTANT LED_0, C0            ; display '0' on 7-segment display
CONSTANT LED_1, F9            ; display '1' on 7-segment display
;
CONSTANT LED_F, 8E            ; display 'F' on 7-segment display

NAMEREG s0, switch_value      ; read switches into register s0
NAMEREG s1, LED_output        ; load LED output data in register s1

; Load 7-segment LED patterns into scratchpad RAM
LOAD LED_output, LED_0        ; grab LED pattern for switches = 0000
STORE LED_output, 00           ; store in RAM[0]
LOAD LED_output, LED_1        ; grab LED pattern for switches = 0001
STORE LED_output, 01           ; store in RAM[1]
;
LOAD LED_output, LED_F        ; grab LED pattern for switches = 1111
STORE LED_output, 0F           ; store in RAM[F]

; Read switch values and display value on 7-segment LED
loop: INPUT switch_value, switches ; read value on switches
AND switch_value, F0           ; mask upper bits to guarantee < 15
FETCH LED_output, (switch_value) ; look up LED pattern in RAM

OUTPUT LED_output, LEDs       ; display switch value on 7-segment LED
JUMP loop

```

Figure 5-3: Using Scratchpad RAM as a Look-Up Table

Stack Operations

Although the PicoBlaze microcontroller has a CALL/RETURN stack, it does not have a dedicated data stack. In some controller architectures, register values are preserved during subroutine calls or interrupts by pushing them or popping them onto a data stack. The equivalent operation is possible in the PicoBlaze microcontroller by reserving some locations in scratchpad RAM.

In the example shown in [Figure 5-4](#), the my_subroutine function uses register s0. The value of register s0 is preserved onto a “stack”, which is emulated using scratchpad RAM. When the my_subroutine function completes, the preserved value of register s0 is restored from the stack.

```

NAMEREG sF, stack_ptr ; reserve register sF for the stack pointer

; Initialize stack pointer to location 32 in the scratchpad RAM
LOAD sF, 20

my_subroutine:
; preserve register s0
CALL push_s0

; *** remainder of subroutine algorithm ***

; restore register s0
CALL pop_s0
RETURN

push_s0:
STORE s0, stack_ptr ; preserve register s0 onto "stack"
ADD stack_ptr, 01 ; increment stack pointer
RETURN

pop_s0:
SUB stack_ptr, 01 ; decrement stack pointer
FETCH s0, stack_ptr ; restore register s0 from "stack"
RETURN

```

Figure 5-4: Use Scratchpad RAM to Emulate PUSH and POP Stack Operations

FIFO Operations

In a similar vein, FIFOs can be created using two separate pointers into scratchpad RAM. One pointer tracks data being written into RAM; the other tracks data being read from RAM.

See also:

- “[STORE sX, Operand — Write Register sX Value to Scratchpad RAM Location](#),” page 112.
- “[FETCH sX, Operand — Read Scratchpad RAM Location to Register sX](#),” page 99.

Input and Output Ports

The PicoBlaze microcontroller supports up to 256 input ports and 256 output ports that can also be combined to create input/output ports. The interface signals from [Figure 2-1](#) involved in INPUT and OUTPUT operations are described below.

- The PORT_ID[7:0] output port presents the port identifier number or port address for both INPUT and OUTPUT operations.
- The IN_PORT[7:0] input port captures input data during INPUT operations.
- The OUT_PORT[7:0] output port presents output data during OUTPUT operations.
- The READ_STROBE output is asserted High during the second cycle of the two-cycle INPUT operation.
- The WRITE_STROBE output is asserted High during the second cycle of the two-cycle OUTPUT operation.

In timing critical designs, set timing constraints for the PORT_ID and data paths allowing two clock cycles. Only the read and write strobes need to be constrained to a single clock cycle. For maximum performance and to simplify timing constraints, insert a pipeline register where possible, as described in the following sections.

Thought-out design keeps the interface logic compact with good performance. The following diagrams show circuits suitable for output ports, input ports, and for connecting memory. When using a logic synthesis tool, check that the source code is not describing a circuit that is more complex than is actually required and that the synthesis tool is implementing the intended logic.

PORT_ID Port

The 8-bit PORT_ID port supplies the port identifier or port address for the associated INPUT or OUTPUT operation. The PORT_ID port is valid for two clock cycles, allowing sufficient time for any interface decoding logic and for connections to asynchronous RAM. Similarly, the two-cycle operation allows read operations from synchronous RAM, such as block RAM.

INPUT and OUTPUT operations support both direct and indirect addressing. The port address is supplied as either as an 8-bit immediate constant or specified indirectly as the contents of any of the 16 data registers. Indirect addressing is ideal when accessing a block of memory, either a peripheral at contiguous port addresses or some form of block or distributed memory within or external to the FPGA.

Adding external peripherals to the PicoBlaze microcontroller is relatively straightforward. The only challenge is decoding the PORT_ID value using the minimum required logic for the application. The decoding challenge depends on the number of input, output, or bidirectional ports, as described in [Table 6-1](#) and subsequent text.

Table 6-1: Decoding PORT_ID Depending on Number of Ports

Number of Ports	INPUT	OUTPUT
0 to 1	No multiplexing required	No decoding required
2 to 8	Single input multiplexer Binary encode PORT_ID	"One hot" encode PORT_ID
9 to 256	Cascaded multiplexer tree Binary encode PORT_ID	Binary encode PORT_ID Hybrid "one hot"/binary encoded

INPUT Operations

An INPUT operation transfers the data supplied on the IN_PORT input port to any one of the 16 data registers, defined by register sX , as shown in Figure 6-1. The PORT_ID output port, defined either by register sY or an 8-bit immediate constant, selects the desired input source. Input sources are generally selected via a multiplexer, using a portion of the bits from the PORT_ID output port to select a specific source. The size of the multiplexer is proportional to the number of possible input sources, which has direct implications on performance.

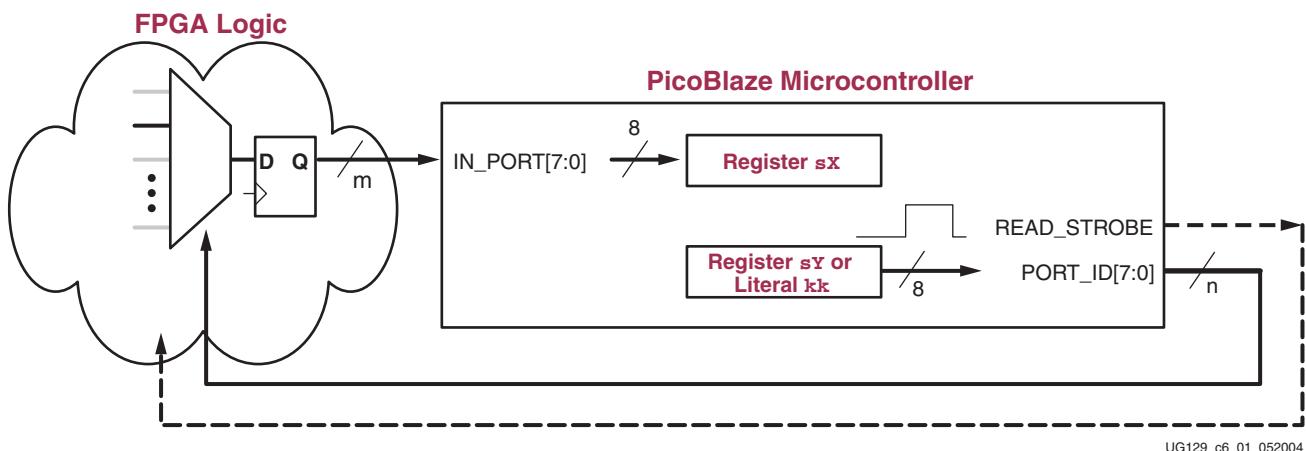


Figure 6-1: INPUT Operation and FPGA Interface Logic

The INPUT operation asserts the associated READ_STROBE output pulse on the second cycle of the two-cycle INPUT cycle, as shown in Figure 6-2. The READ_STROBE signal is seldom used in applications but it indicates that the PicoBlaze microcontroller has acquired the data. READ_STROBE is critical when reading data from a FIFO, acknowledging receipt of data as shown in Figure 6-4.

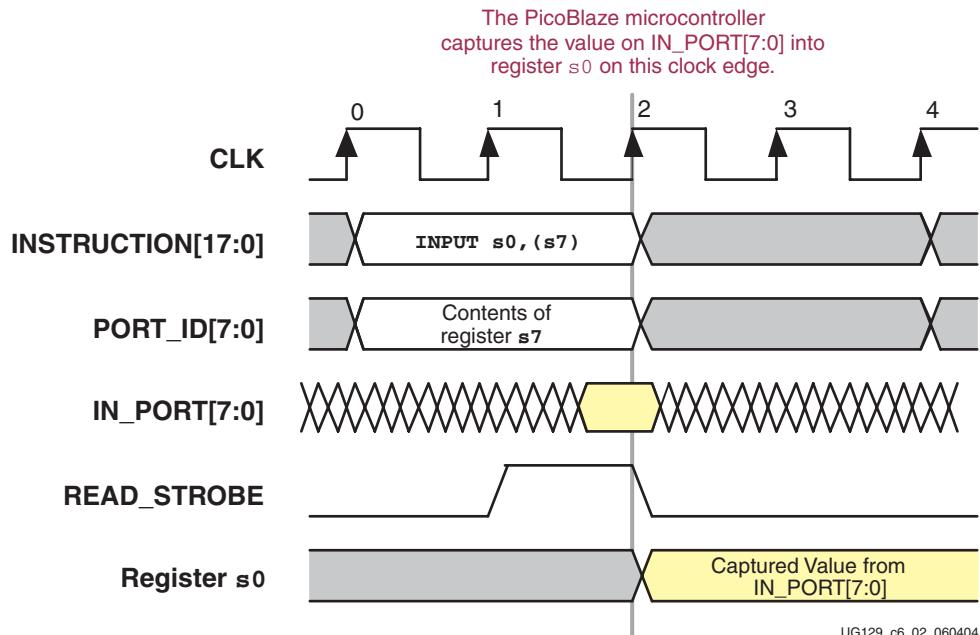
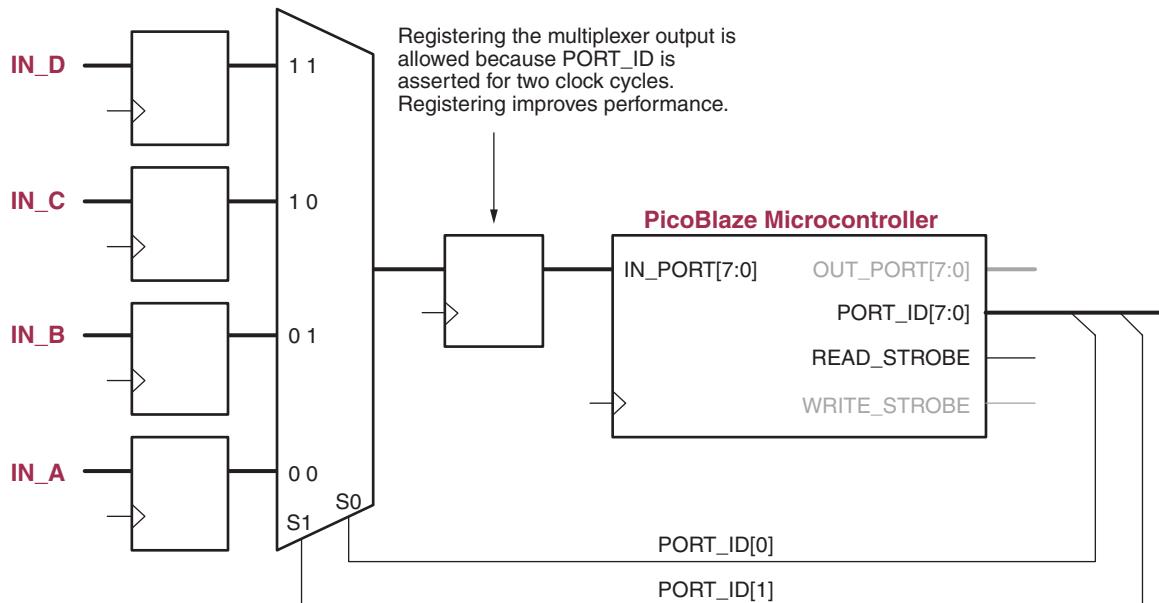


Figure 6-2: Port Timing for INPUT Instruction

In this example, the PicoBlaze microcontroller is reading data from the port address defined by the contents of register s7. The read data is captured in register s0. When the instruction executes, the contents of register S7 appear on the PORT_ID port. The PORT_ID is then decoded by FPGA logic external to the PicoBlaze microcontroller and the requested data is eventually presented on the IN_PORT port. The READ_STROBE signal goes High during the second clock cycle of the instruction, although the READ_STROBE signal is primarily used only by FIFOs so that the FIFO can update its read pointer. The data presented on the IN_PORT port is captured on rising clock edge 2, marking the end of the INPUT instruction. Data needs only be present with sufficient setup time to this clock edge. After rising clock edge 2, the data on the IN_PORT port is captured and available in the target register, register s0 in this case.

Because the PORT_ID is valid for two clock cycles, the input data multiplexer can be registered to maintain performance, as shown in Figure 6-3. In most applications, the actual clock cycle when the PicoBlaze microcontroller reads an input is not critical. Therefore the paths from the various sources can typically be registered. For example, signals arriving from the FPGA pins can be captured using input flip-flops. Registering the input path simplifies timing specifications, avoids reports of 'false paths' and leads to more reliable designs.



UG129_c6_03_060404

Figure 6-3: Multiplex Multiple Input Sources to Form a Single IN_PORT Port

Failure to include a register anywhere in the path from PORT_ID to IN_PORT is the most common reason for decreased system clock rates. Consequently, make sure that this path is registered at some point.

Applications with Few Input Sources

If the application has 32 or less input ports, then a single multiplexer is ideal to connect the various input signals to the IN_PORT input port, as shown in [Figure 6-3](#). Check the results of synthesis to ensure that the special MUXF5, MUXF6, MUXF7, and MUXF8 are being employed to make the most efficient multiplexer structure.

Refer to XAPP466: Using Dedicated Multiplexers in Spartan-3 FPGAs (see Reference 5).

READ_STROBE Interaction with FIFOs

Occasionally, the circuit providing data to the PicoBlaze microcontroller needs to know that it was successfully read. [Figure 6-4](#) shows an example using a FIFO buffer. The FIFO only updates its read pointer once the PicoBlaze microcontroller successfully captures data, indicated by the READ_STROBE signal.

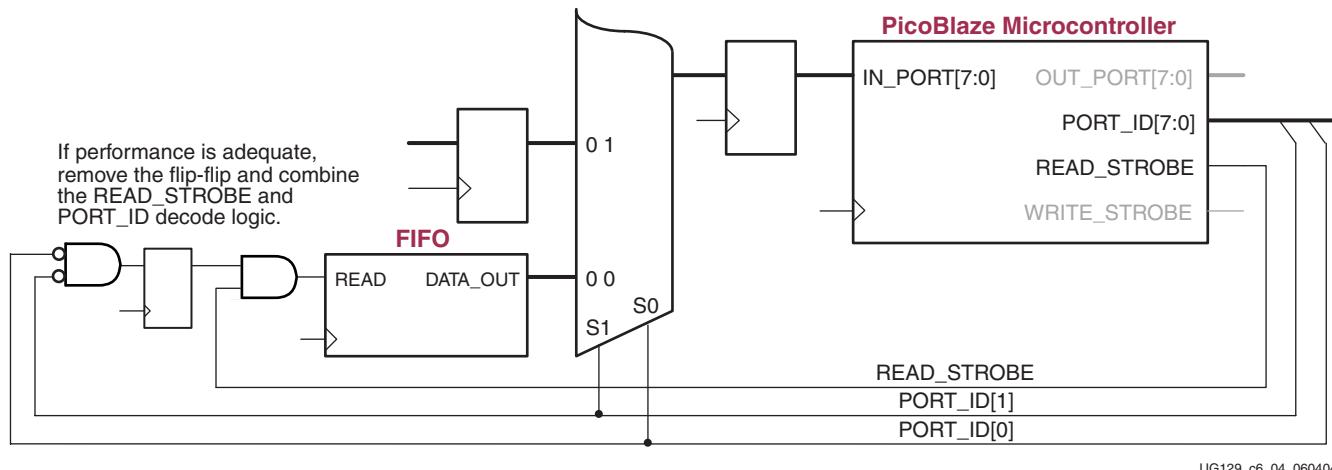


Figure 6-4: **READ_STROBE** Indicates a Successful INPUT Operation

OUTPUT Operations

As shown in [Figure 6-5](#), an OUTPUT operation presents the contents of any of the 16 registers to the OUT_PORT output port. The PORT_ID output port, defined either by register sY or an 8-bit immediate constant, selects the desired output destination. The WRITE_STROBE output pulse indicates that data on the OUT_PORT port is valid and ready for capture. Typically, the WRITE_STROBE signal, combined with the decoded PORT_ID port, is used as either a clock enable or a write enable signal to other FPGA logic that captures the output data.

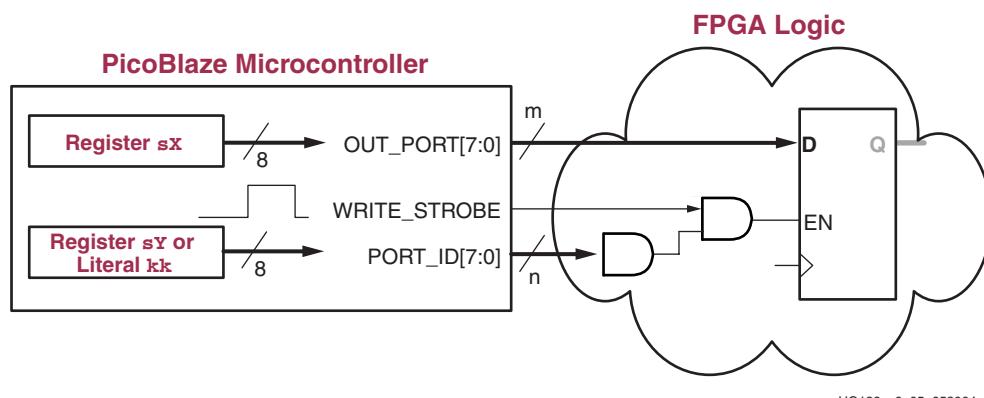


Figure 6-5: **OUTPUT Operation and FPGA Interface**

The OUTPUT operation asserts the associated WRITE_STROBE output pulse beginning on rising CLK edge 1 of the two-cycle OUTPUT instruction , as shown in [Figure 6-6](#). In this particular example, the PicoBlaze microcontroller writes the contents of register $s0$ to hexadecimal port address 65. The contents of register $s0$ appear on the OUT_PORT port; the port address appears on the PORT_ID port. The WRITE_STROBE goes High on the second clock cycle to indicate that data is valid.

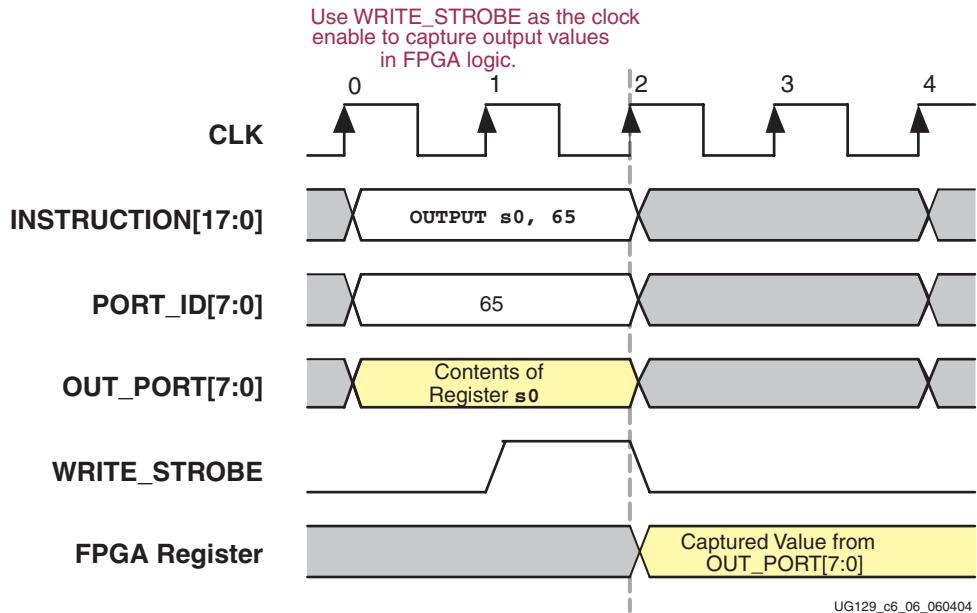


Figure 6-6: Port Timing for OUTPUT Instruction

Simple Output Structure for Few Output Destinations

For eight or less simple output ports, use “one-hot” port addresses and only decode the appropriate **PORT_ID** signal, as shown in [Figure 6-7](#). This technique greatly reduces the address decode logic which lowers cost and maximizes performance. This approach also reduces the loading on the **PORT_ID** bus, which is often critical to overall system performance.

If the number of decoded **PORT_ID** bits is three or less, then the decode logic fits in a single level of FPGA logic, maximizing performance.

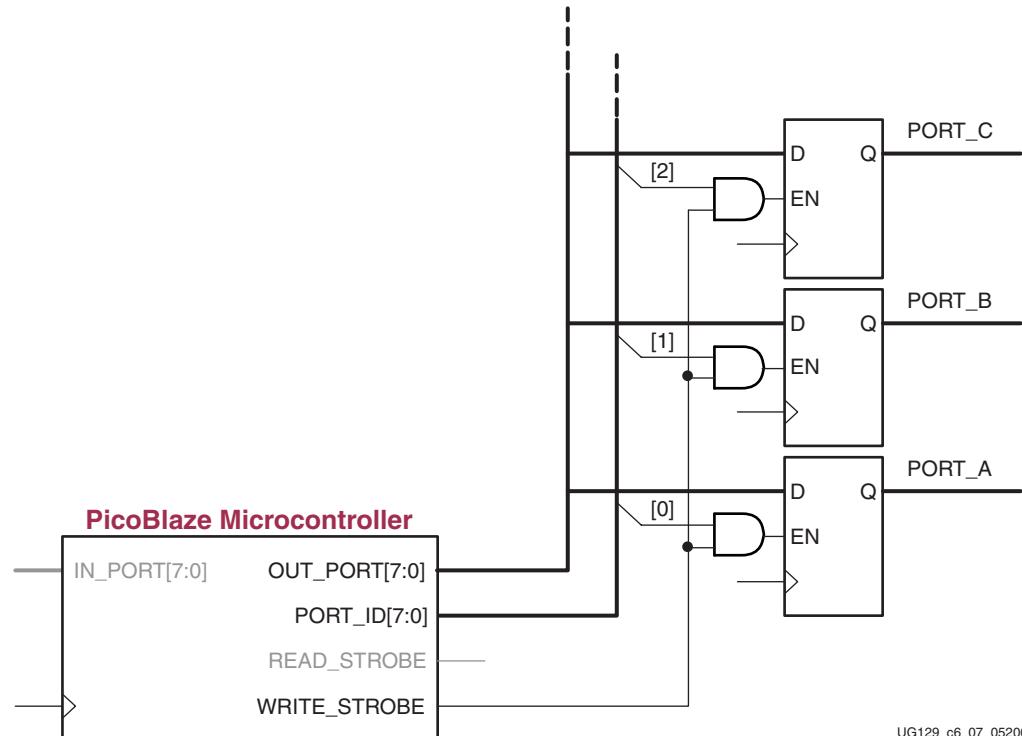


Figure 6-7: Simple Address Decoding for Designs with Few Output Destinations

As shown in Figure 6-8, use CONSTANT directives in the program make the code readable and help ensure that the correct ports are decoded. Because the PORT_ID addresses use “one-hot” encoding, it is also possible to create a single address that incorporates all the individual addresses. This way, the PicoBlaze microcontroller can send a broadcast message to all of the output destinations—in this case, a single instruction clears all destinations.

```

; Use CONSTANT declarations to define output port addresses
CONSTANT Port_A, 01
CONSTANT Port_B, 02
CONSTANT Port_C, 04
CONSTANT Port_D, 08
CONSTANT Broadcast, FF
;
; Use assigned port names for better readability
OUTPUT s0, Port_A
OUTPUT s1, Port_B
OUTPUT s2, Port_C
OUTPUT s4, Port_D
;
; Send broadcast message to all addresses to clear all output register
LOAD s0, 00
OUTPUT s0, Broadcast

```

Figure 6-8: Use CONSTANT Directives to Declare Output Port Addresses

Pipelining for Maximum Performance

In most applications, the PicoBlaze microcontroller has more than sufficient performance to meet application requirements. However, PicoBlaze designs attached to multiple memory blocks or that have many simple ports may end up using most, if not all, of the 256 available port addresses. Decoding and routing all 256 locations complicates the overall design, especially for designs requiring maximum performance.

Pipelining the PORT_ID decoding function improves overall system performance. During an OUTPUT operation, both the PORT_ID and OUT_PORT ports are valid for two clock cycles while the WRITE_STROBE output is only active during the second of the two cycles, as shown [Figure 6-6](#).

One approach to improving interface performance is to pipeline the PORT_ID decoding logic, as illustrated in [Figure 6-9](#). In designs with many ports, the fanout and loading on the PORT_ID bus limits maximum performance. Fortunately, because the PORT_ID port is active for two clock cycles, the PORT_ID logic can be pipelined. Each decoded PORT_ID value is then captured in a flip-flop. Each pipelined decode value is qualified using the WRITE_STROBE signal during the next clock cycle to actually capture the OUT_PORT data.

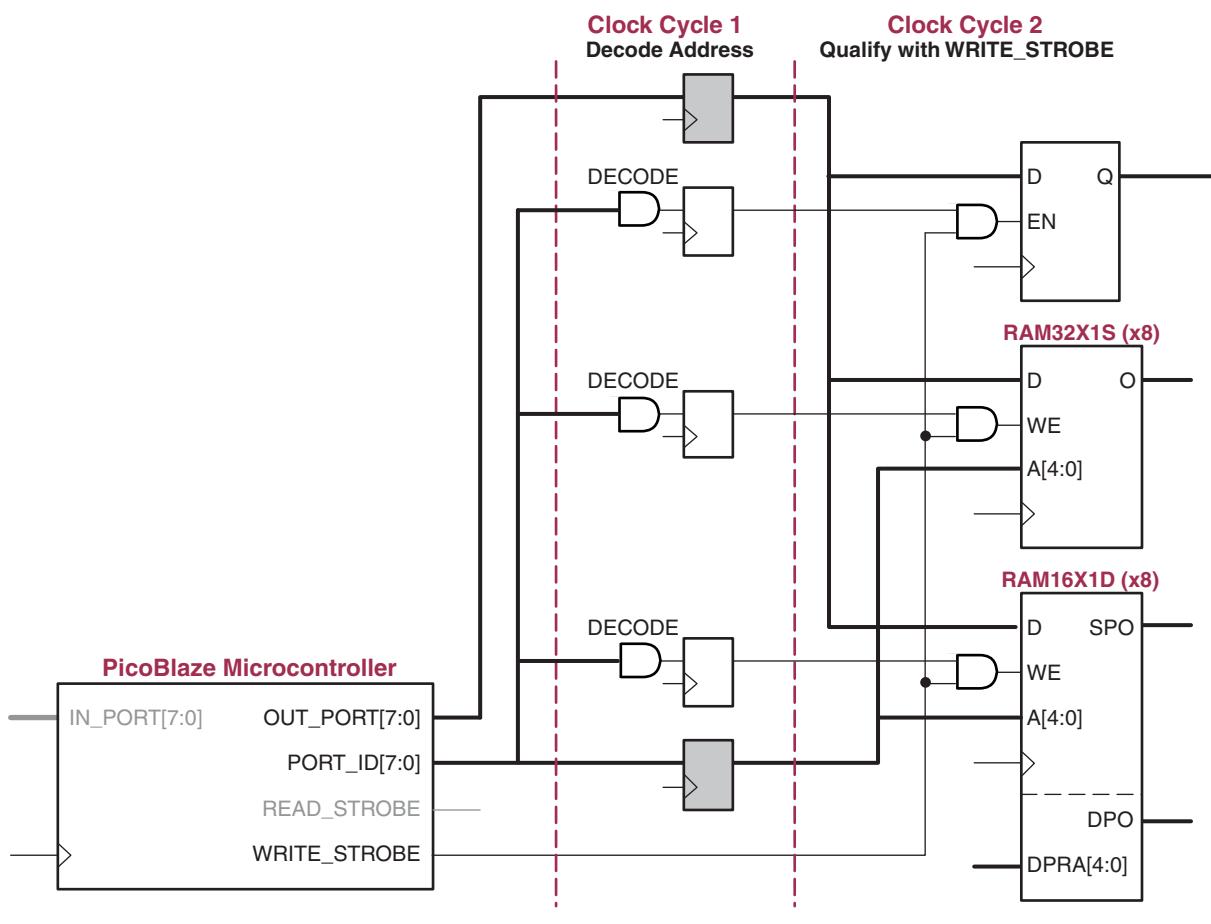


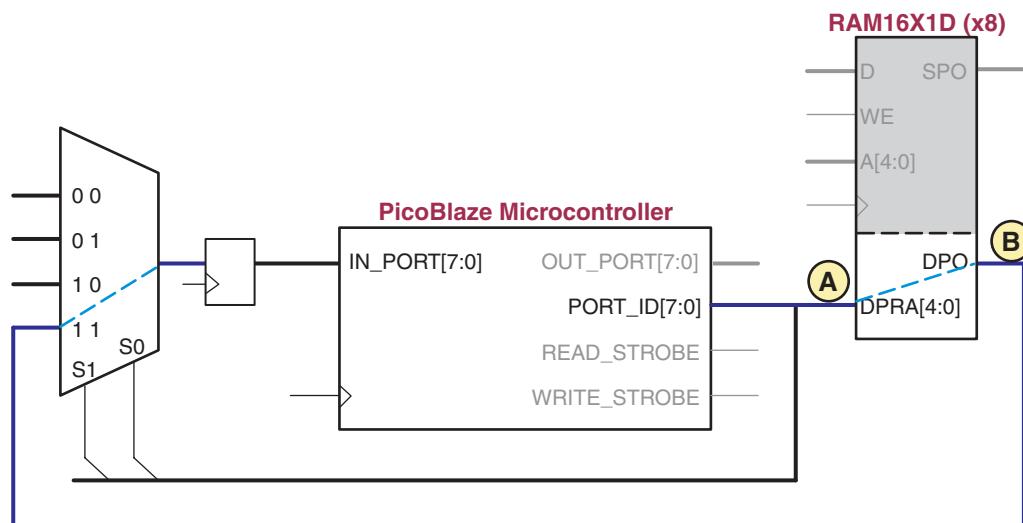
Figure 6-9: Pipelining the PORT_ID Decoding Improves Performance

UG129_c6_08_052004

The pipelining registers on the OUT_PORT and PORT_ID signals, shaded in [Figure 6-9](#), are optional. Both OUT_PORT and PORT_ID are valid for two clock cycles. However, pipelining them decreases the initial fanout and reduces the routing distance, both of which improve performance.

During OUTPUT operations, the PicoBlaze microcontroller has no data dependencies and consequently no dependencies on the FPGA interface logic. If data takes longer than the two-clock instruction cycle to be captured by the FPGA logic, so be it. The PicoBlaze microcontroller initiates the OUTPUT operation but does not need to wait while the FPGA logic captures the data in its ultimate location as long as data is not lost. However, pipelining INPUT operations can be more complicated. During an INPUT operation, the PicoBlaze microcontroller requests data from the FPGA logic and must receive the data to successfully complete the instruction.

[Figure 6-10](#) illustrates the dependency, where the critical timing path is blue. In this example, the PicoBlaze microcontroller is reading data from a dual-port RAM. This example assumes that some other function within the FPGA writes data into the dual-port RAM. When the PicoBlaze microcontroller reads data from the dual-port RAM, the read address appears on the PORT_ID port. The critical path is the delay from the PORT_ID port, through the dual-port RAM read path, through the input select multiplexer, to the setup on the pipelining register. If this path limits performance, add a pipelining register to improve performance. However, where is the best position for the pipeline register, Point A or Point B?



UG129_c6_09_052004

Figure 6-10: Without Pipelining, the Full Read Path Delay Can Reduce Performance

From [Figure 6-2](#), the read data for INPUT operations must be presented and valid on the IN_PORT port by the end of the second clock cycle. There is already one layer of pipelining immediately following the input select multiplexer feeding the IN_PORT port. Adding a pipelining register at Point A or Point B delays data by an additional clock cycle, too late to meet the PicoBlaze microcontroller's requirements.

The best place to position the pipeline register is at Point B, which splits the read path roughly in half. However, the input select multiplexer structure must be modified to accommodate the extra register layer, as shown in [Figure 6-11](#).

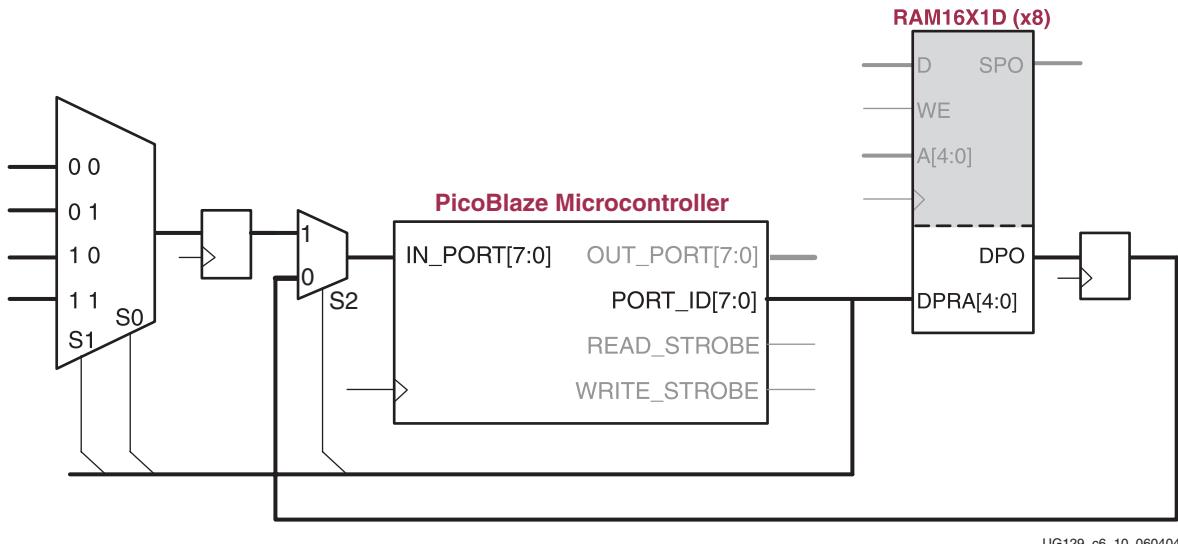


Figure 6-11: Effective Pipelining Improves Read Performance

Repartitioning the Design for Maximum Performance

Another approach to maximizing performance is to re-evaluate the system requirements. If the number of I/O ports is the bottleneck in the system, ask if all the ports are actually required as part of a single application or whether a single PicoBlaze microcontroller is performing multiple tasks. If multiple tasks share a single PicoBlaze core, consider partitioning the design into multiple PicoBlaze applications, each with a reduced number of I/O ports. Partitioning the design may have additional benefits such as simplifying the application code and reducing resource requirements.

Instruction Storage Configurations

The PicoBlaze microcontroller executes code from memory resources embedded within the FPGA. Figure 7-1 shows that the PicoBlaze microcontroller actually consists of two subfunctions. The KCPSM3 module contains the PicoBlaze ALU, register file, scratchpad RAM, etc. Some form of internal memory, typically a block RAM, provides the PicoBlaze instruction store. To effectively create an on-chip ROM, the block RAM's write enable pin, WE, is held Low, disabling any potential write operations.

However, the PicoBlaze microcontroller supports other implementations that have advantages for specific applications as described below. Many of these alternate implementations leverage the extra port provided by the dual-port block RAM on Spartan-3, Virtex-II, and Virtex-II Pro FPGAs.

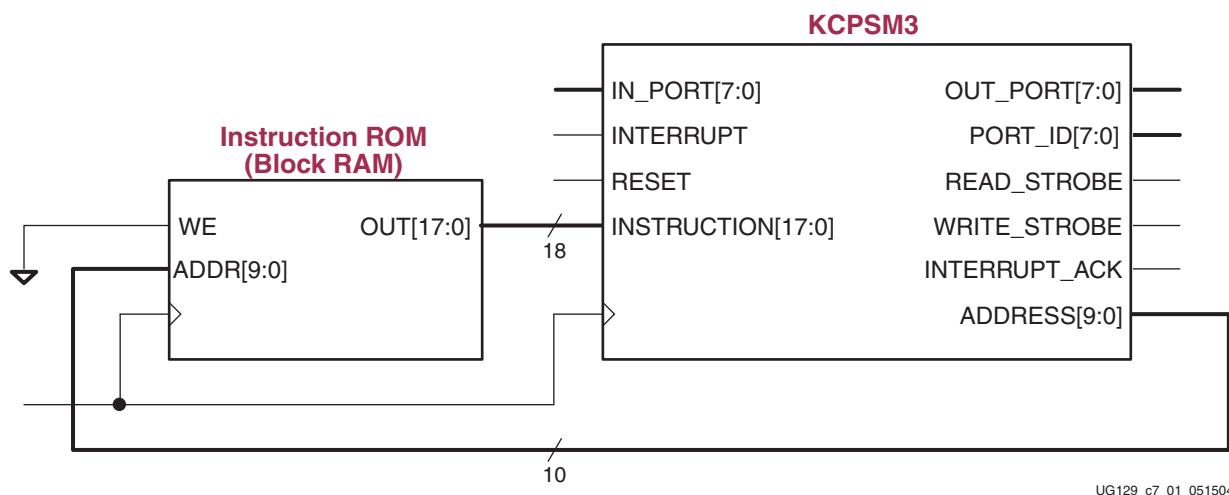


Figure 7-1: Standard Implementation using a Single 1Kx18 Block RAM as the Instruction Store

Standard Configuration – Single 1Kx18 Block RAM

In most applications, PicoBlaze instructions are stored in a single FPGA block RAM, configured as a 1Kx18 ROM shown in Figure 7-2. The application code is assembled and ultimately compiled as part of the FPGA design. The instruction store is automatically loaded into the attached block RAM during the FPGA configuration process.

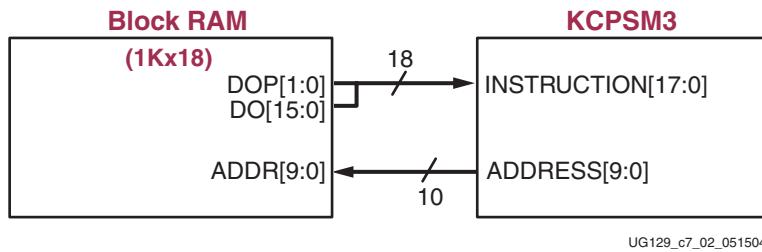


Figure 7-2: Standard Configuration using a Single 1Kx18 Block RAM

Standard Configuration with UART or JTAG Programming Interface

The second read/write port on the block RAM provides a convenient means to update the PicoBlaze instruction store without recompiling the entire FPGA design. While the processor is halted, application code can be updated via a simple UART or via the FPGA's JTAG port, as shown in Figure 7-3.

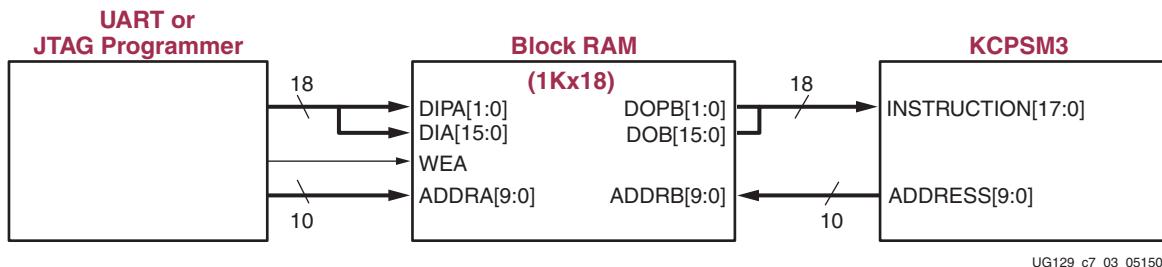


Figure 7-3: Standard Configuration with UART or JTAG Program Loader

Refer to “Reconfiguring Block RAMs via JTAG” (see Reference 6) for additional details on implementing this technique.

Two PicoBlaze Microcontrollers Share a 1Kx18 Code Image

As shown in Figure 7-4, two PicoBlaze microcontrollers can share a single dual-port block RAM to store a common or mostly common code image. The two microcontrollers operate entirely independently of one another although they each independently execute the same or mostly the same code. The clock input, I/O ports, and interrupt input are unique to each microcontroller.

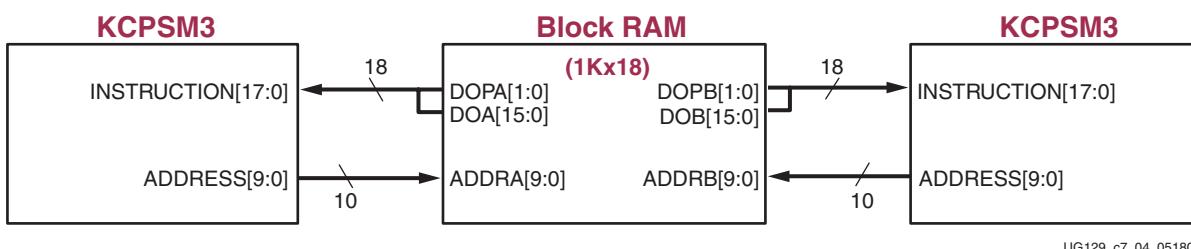


Figure 7-4: Two PicoBlaze Microcontrollers Sharing a Common Code Image

Two PicoBlaze Microcontrollers with Separate 512x18 Code Images in a Block RAM

Two PicoBlaze microcontrollers can also share a single dual-port RAM but each with a separate 512-instruction area, as shown in Figure 7-5. The most-significant address bit of one block RAM port is tied Low while the other same bit on the other port is tied High. This limits each port to half of the 1Kx18 memory, or 512x18. The two microcontrollers operate independently.

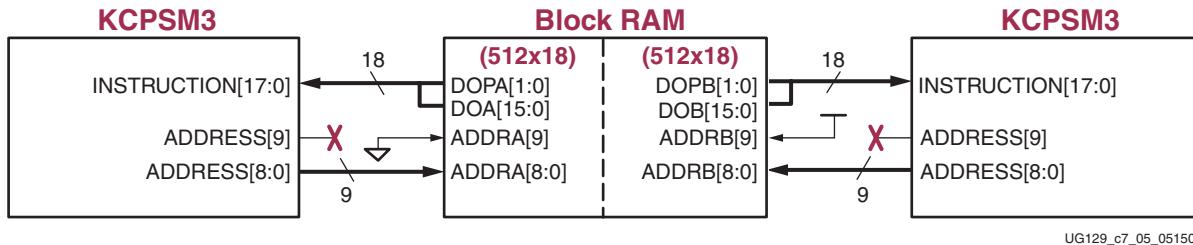


Figure 7-5: Two PicoBlaze Microcontrollers with Separate 512-Instruction Memory in one Block RAM

Despite that both PicoBlaze microcontrollers use half the normal instruction store, the interrupt vectors for both remain the same. When an interrupt occurs, the associated KCPSM3 block presents all ones on the ADDRESS bus, which is truncated to the last memory location in its half of the block RAM memory (address 1FF hexadecimal).

Figure 7-5 shows the block RAM split into two equal halves. If one microcontroller requires more than the other, then tie the upper address lines as appropriate. Practically any partition is allowed as long as the combined code size is 1,024x18 or less.

Distributed ROM Instead of Block RAM

Block RAM is the most efficient method to store PicoBlaze application code. However, if all the block RAM within the FPGA is already committed to other functions then the PicoBlaze code can be stored within the FPGAs Configurable Logic Blocks (CLBs), as shown in Figure 7-6.

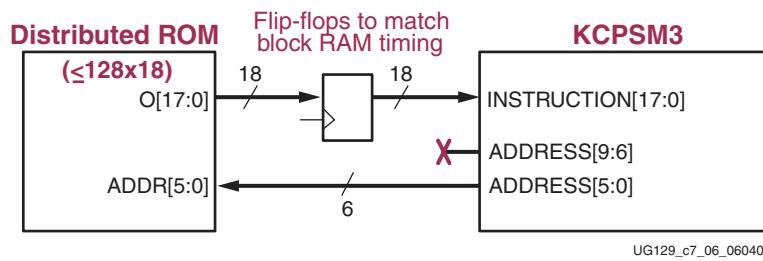


Figure 7-6: Using Distributed ROM for Instruction Memory

This technique is only roughly efficient if the program size is 128 instructions or less. Although larger instruction stores are possible, they quickly consume CLB logic. Table 7-1 shows the number of FPGA slices required for various instruction stores. Distributed ROM essentially uses the Look-Up Tables (LUTs) within its FPGA logic block as a small ROM instead of for logic.

To maintain compatibility with block RAM, the distributed ROM must have a registered output using CLB flip-flops.

Table 7-1: Slices Required when Using CLBs for ROM

Instructions	ROM Slices
≤ 16	9
≤ 32	18
≤ 64	36
≤ 128	72
≤ 256	144
≤ 512	297
$\leq 1,024$	594

The CORE Generator software can create all of the above distributed ROM functions using the coefficients file generated by the PicoBlaze assembler.

Performance

Input Clock Frequency

Table 8-1 shows the maximum available performance for the PicoBlaze microcontroller using various FPGA families and speed grades. The Virtex-II and Virtex-II Pro FPGA families are optimized for maximum performance. The Spartan-3 FPGA family is optimized for lowest cost.

Table 8-1: PicoBlaze Performance Using Slowest Speed Grade

FPGA Family (Speed Grade)	Maximum Clock Frequency	Maximum Execution Performance
Spartan-3 (-4) FPGA	88 MHz	44 MIPS
Virtex-II (-6) FPGA	152 MHz	76 MIPS
Virtex-II Pro (-7) FPGA	200 MHz	100 MIPS

Unless the end application requires absolute performance, there is no need to operate the PicoBlaze microcontroller at its maximum clock frequency. In fact, operating slower is advantageous. Often, the PicoBlaze microcontroller is managing slower peripheral operations like serial communications or monitoring keyboard buttons, neither of which stresses the FPGA's performance. A lower clock frequency reduces the number of idle instruction cycles and reduces total system power consumption.

The PicoBlaze microcontroller is a fully static design and operates down to DC (0 MHz).

Predicting Executing Performance

All instructions always execute in two clock cycles, resulting in predictable execution performance. In real-time applications, a constant execution rate simplifies calculating program execution times.

Using the PicoBlaze Microcontroller in an FPGA Design

The PicoBlaze microcontroller is primarily designed for use in a VHDL design flow. However, both Verilog and black box instantiation are also supported, as described below. Similarly, Xilinx XST/ISE 6.2i and later versions support mixed language support where both VHDL and Verilog can be mixed in a single project. There is also support for the Xilinx System Generator design environment.

VHDL Design Flow

The PicoBlaze microcontroller is supplied as a VHDL source file, called KCPSM3.vhd, which is optimized for efficient and predictable implementation in a Spartan-3, Virtex-II, or Virtex-II Pro FPGA. The code is suitable for both synthesis and simulation and was developed and tested using the Xilinx Synthesis Tool (XST) for logic synthesis and ModelSim for simulation. Designers have also successfully used other logic synthesis and simulation tools. The VHDL source code must not be modified in any way.

KCPSM3 Module

The KCPSM3 module contains the PicoBlaze ALU, register file, scratchpad, RAM, etc. The only function not included is the instruction store. The component declaration for the KCPSM3 module appears in Figure 9-1. Figure 9-2 lists the KCPSM3 component instantiation.

```
component KCPSM3
port (
    address      : out std_logic_vector( 9 downto 0);
    instruction   : in  std_logic_vector(17 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;
    interrupt_ack : out std_logic;
    reset         : in  std_logic;
    clk           : in  std_logic
);
end component;
```

Figure 9-1: VHDL Component Declaration of KCPSM3

```

processor: kcpsm3
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,
    interrupt_ack => interrupt_ack_signal,
    reset => reset_signal,
    clk => clk_signal
);

```

Figure 9-2: VHDL Component Instantiation of the KCPSM3

Connecting the Program ROM

The PicoBlaze program ROM is used within a VHDL design flow. The PicoBlaze assembler generates a VHDL file in which a block RAM and its initial contents are defined. This VHDL file can be used for both logic synthesis and simulation of the processor.

[Figure 9-3](#) shows the component declaration for the program ROM, and [Figure 9-4](#) shows the component instantiation. The name of the program ROM, shown as "prog_rom" in the following figures, is derived from the name of the PicoBlaze assembler source file. For example, if the assembler source file is named phone.psm, then the assembler generates a program ROM definition file called phone.vhd.

```

component prog_rom
port (
    address : in std_logic_vector( 9 downto 0 );
    instruction : out std_logic_vector(17 downto 0 );
    clk : in std_logic
);
end component;

```

Figure 9-3: VHDL Component Declaration of Program ROM

```

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

```

Figure 9-4: VHDL Component Instantiation of Program ROM

To speed development, a VHDL file called `embedded_KCPSM3.vhd` is provided. In this file, 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 design files.

Black Box Instantiation of KCPSM3 using KCPSM3.ngc

The Xilinx NGC file included with the reference design was generated by synthesizing the KCPSM3.vhd file using the Xilinx Synthesis Tool (XST), without inserting I/O buffers.

When used as a “black box” in a Spartan-3, Virtex-II or Virtex-II Pro FPGA design, the PicoBlaze microcontroller is merged with the remainder of the FPGA design during the translate phase (ngdbuild).

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

Generating the Program ROM using prog_rom.coe

The KCPSM assembler generates a memory coefficients file (*.coe). Using the Xilinx CORE Generator™ system, create a block ROM using the *.coe file.

The file defines the initial contents of a block ROM. The output files created by the CORE Generator system can then be used in the normal design flow and connected to the PicoBlaze “black box” instantiation of the KCPSM3 module.

Generating an ESC Schematic Symbol

To generate an ESC schematic symbol, use the embedded_KCPSM3.vhd file.

Verilog Design Flow

Beginning with XST/ISE 6.2i, the Xilinx development software allows mixed-language design projects using both VHDL and Verilog. Consequently, the KCPSM3 VHDL source can be included within a Verilog project. The KCPSM3 assembler generates a Verilog file named <filename>.v that defines the initial contents (see assembler notes for more detail). This Verilog file is used to implement and simulate the PicoBlaze instruction store.

The details of mixed VHDL and Verilog language support in the Xilinx ISE software is described in detail in Chapter 8, “Mixed Language Support”, in the *XST User Guide*.

- **XST User Guide**
<http://toolbox.xilinx.com/docsan/xilinx6/books/docs/xst/xst.pdf>

Black-box instantiation is an alternative Verilog design approach. Instantiate the kcspm3.ngc black box file within the Verilog design to define the remainder of the processor.

PicoBlaze Development Tools

There are three primary development environments for creating PicoBlaze application code, as summarized in [Table 10-1](#). Xilinx offers two PicoBlaze environments. The PicoBlaze reference design includes the KCPSM3 command-line assembler that executes in a Windows DOS box or command window. The Xilinx System Generator for DSP development environment includes both a PicoBlaze assembler and a simulation model for the Math Works MATLAB/Simulink environment. The Mediatronix pBlazIDE software is a graphical development environment including an assembler and full-featured instruction-set simulator (ISS).

Table 10-1: PicoBlaze Development Environments

	Xilinx KCPSM3	Mediatronix pBlazIDE	Xilinx System Generator
Platform Support	Windows	Windows 98, Windows 2000, Windows NT, Windows ME, Windows XP	Windows 2000, Windows XP
Assembler	Command-line in DOS window	Graphical	Command-line within System Generator
Instruction Syntax	KCPSM3	PBlazIDE	KCPSM3
Instruction Set Simulator	Facilities provided for VHDL simulation	Graphical/Interactive	Graphical/Interactive
Simulator Breakpoints	N/A	Yes	Yes
Register Viewer	N/A	Yes	Yes
Memory Viewer	N/A	Yes	Yes

KCPSM3

Assembler

The KCPSM3 Assembler is provided as a simple DOS executable file together with three template files. Copy all the files KCPSM3 .EXE, ROM_form.vhd, ROM_form.v, and ROM_form.coe into your working directory.

Programs are best written with either the standard Notepad or Wordpad tools available on most Windows computers. However, any PC-format text editor is sufficient. Save the PicoBlaze assembly program with a PSM file extension (eight-character name limit).

Open a DOS box and navigate to the working directory. To assemble the PicoBlaze program, type:

```
kcpasm3 <filename>[.psm]
```

Assembly Errors

The assembler halts as soon as an error is detected. A short message indicates the reason for any error. The assembler also displays the line that it was analyzing when it detected the problem. Fix each reported problem in turn and re-execute the assembler.

Since the execution of the assembler is very fast, it is unlikely that you will be able to ‘see’ it making progress, and the display will appear to be immediate. To review everything that the assembler has written to the screen, the DOS output can be redirected to a text file using:

```
kcpasm3 <filename>[.psm] > screen_dump.txt
```

Input and Output Files

The KCPASM3 assembler reads four input files and creates 15 output files as shown in [Figure 10-1](#). The KCPASM3 assembler reads the PicoBlaze source program, <filename>.psm, and three template files that instantiate and initialize a block RAM in various design flows, ROM_form.*.

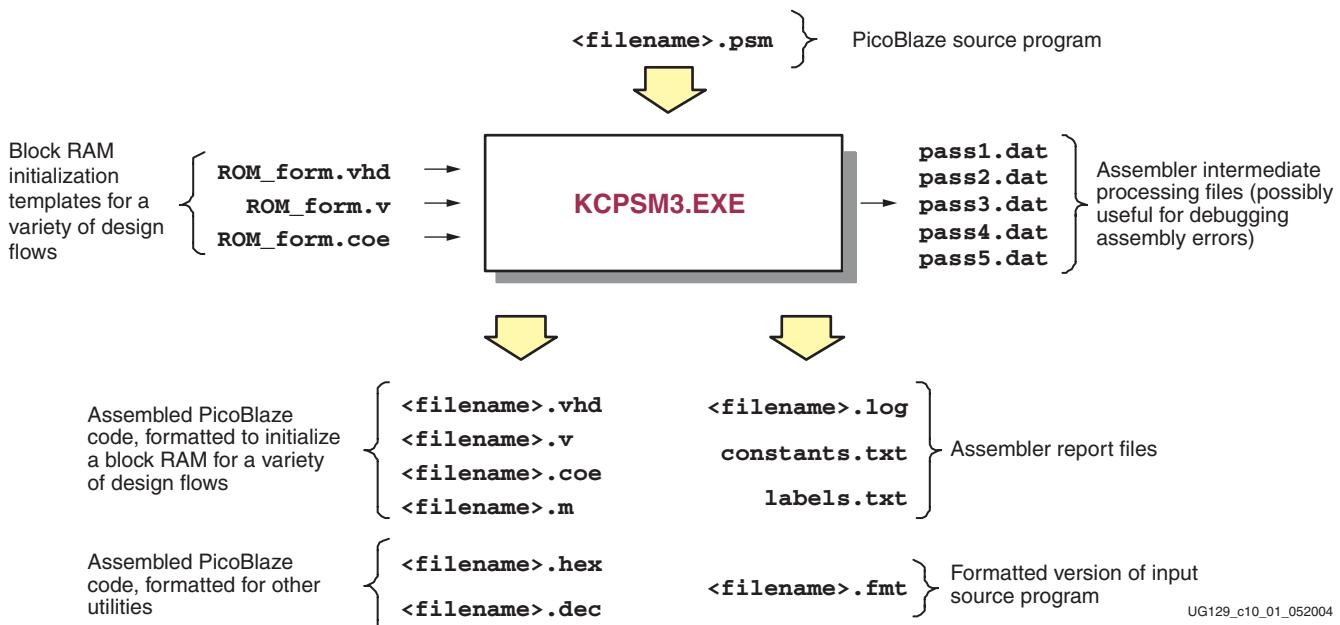


Figure 10-1: KCPASM3 Assembler Files

All five assembler passes are recorded in the pass*.dat output files. Should an error occur during assembly, these files may contain additional details on the error.

If the source program is free of errors, the KCPASM3 assembler generates an object code output, formatted for a variety of design flows, based on the initial template files. These output files generate the code ROM, instantiated as a block RAM, and properly initialized with the PicoBlaze object code. The assembler also generates equivalent raw decimal and hexadecimal output files for other utilities.

The assembler also produces a log file plus files that show the assignments for various labels and constants found in the source code. The log file shows the instruction address, the opcode for each instruction, and the source code instruction and comments for the associated instruction address. The assigned values for register names, labels, and constants appear immediately following the associated symbolic name.

Finally, the KCPSM3 assembler generates a formatted version of the source program (“pretty print” output). The formatted output file formats all labels and comments, converts all commands and hexadecimal constants to upper case, and consistently spaces operands.

Mediatronix pBlazIDE

The Mediatronix pBlazIDE software, shown in [Figure 12-1, page 85](#), is a free, graphical, integrated development environment for Windows-based computers. Its features are as follows:

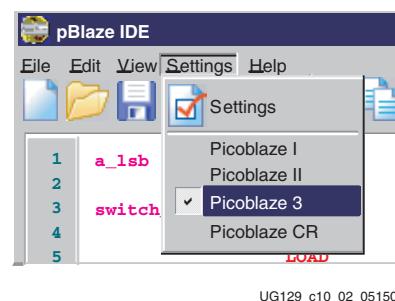
- Syntax color highlighting
- Instruction set simulator (ISS)
 - ◆ Breakpoints
 - ◆ Register display
 - ◆ Memory display
- Source code formatter (“pretty print”)
- KCPSM3-to-pBlazIDE import function/syntax conversion
- HTML output, including color highlighting

Download the pBlazIDE software directly from the Mediatronix website:

<http://www.meditronix.com/pBlazIDE.htm>

Configuring pBlazIDE for the PicoBlaze Microcontroller

The pBlazIDE development software supports all four variants of the PicoBlaze architecture. To use the PicoBlaze microcontroller for Spartan-3, Virtex-II, or Virtex-II Pro FPGAs, choose **Settings → Picoblaze 3** from the pBlazIDE menu, as shown in [Figure 10-2](#).



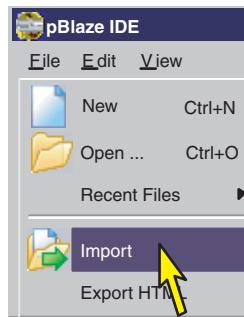
UG129_c10_02_051504

Figure 10-2: Configuring pBlazIDE to Support the PicoBlaze Microcontroller

Importing KCPSM3 Code into pBlazIDE

The pBlazIDE syntax and instruction mnemonics are different than the Xilinx KCPSM3 syntax. The pBlazIDE software provides an import function to convert KCPSM3 code to the pBlazIDE syntax.

From the pBlazIDE menu, choose **File → Import**, then select the KCPSM3-format *.psm file, as shown in [Figure 10-3](#). The pBlazIDE software automatically translates and formats the source code, as shown in [Figure 10-4](#).



UG129_c10_03_051504

Figure 10-3: Converting Xilinx Syntax PicoBlaze Source Code to pBlazIDE Syntax

KCMPSM Source Code	Code Imported/Converted into pBlaze IDE
<pre> CONSTANT myconstant, A5 NAMEREG s0, count16_lsb NAMEREG s1, count16_msb ADDRESS 000 main: ; initialize 16-bit counter, enable interrupts LOAD count16_lsb, myconstant ENABLE_INTERRUPT loop: ; continuously increment 16-bit counter CALL increment_count JUMP loop end_main: increment_count: ; add 1 to LSB of 16-bit counter ADD count16_lsb, 01 ; only add one to MSB if carry generated by LSB ADDCY count16_msb, 00 RETURN isr: ; decrement 16-bit counter by one on interrupt ; subtract 1 from LSB of 16-bit counter SUB count16_lsb, 01 ; only subtract one from MSB if borrow ; generated by LSB SUBCY count16_msb, 00 RETNEABLE ; interrupt vector is always in last memory location ADDRESS 3FF ; jump to interrupt service routing (ISR) JUMP isr </pre>	<pre> myconstant EQU \$A5 count16_lsb EQU s0 count16_msb EQU s1 ORG 0 main: ; initialize 16-bit counter, enable interrupts LOAD count16_lsb, myconstant EINT loop: ; continuously increment 16-bit counter CALL increment_count JUMP loop end_main: increment_count: ; add 1 to LSB of 16-bit counter ADD count16_lsb, 1 ; only add one to MSB if carry generated by LSB ADDC count16_msb, 0 RET isr: ; decrement 16-bit counter by one on interrupt ; subtract 1 from LSB of 16-bit counter SUB count16_lsb, 1 ; only subtract one from MSB if borrow ; generated by LSB SUBC count16_msb, 0 RETI ENABLE ; interrupt vector is always in last memory location ORG \$3FF ; jump to interrupt service routing (ISR) JUMP isr </pre>

UG129_c10_04_052004

Figure 10-4: Example of How KCPSM Source Code Converts to pBlazIDE Code

Differences Between the KCPSM3 Assembler and pBlazIDE

Table 10-2 details the differences between the KCPSM3 and pBlazIDE instruction mnemonics.

Table 10-2: Instruction Mnemonic Differences between KCPSM3 and pBlazIDE

KCPSM3 Instruction	pBlazIDE Instruction
RETURN	RET
RETURN C	RET C
RETURN NC	RET NC
RETURN Z	RET Z
RETURN NZ	RET NZ
RETURNI ENABLE	RETI ENABLE
RETURNI DISABLE	RETI DISABLE
ADDCY	ADDC
SUBCY	SUBC
INPUT sX, (sY)	IN sX, sY (no parentheses)
INPUT sX, kk	IN sX, kk
OUTPUT sX, (sY)	OUT sX, sY (no parentheses)
OUTPUT sX, kk	OUT sX, kk
ENABLE INTERRUPT	EINT
DISABLE INTERRUPT	DINT
COMPARE	COMP
STORE sX, (sY)	STORE sX, sY (no parentheses)
FETCH sX, (sY)	FETCH sX, sY (no parentheses)

Directives

Table 10-3 lists the KCPSM3 and PBlazIDE directives for various functions.

Table 10-3: Directives

Function	KCPSM3 Directive	PBlazIDE Directive
Locating Code	ADDRESS 3FF	ORG \$3FF
Aliasing Register Names	NAMEREG s5, myregname	myregname EQU s5
Declaring Constants	CONSTANT myconstant, 80	myconstant EQU \$80
Naming the program ROM file	Named using the same base filename as the assembler source file	VHDL "template.vhd", "target.vhd", "entity_name"

Assembler Directives

Both the KCPSM3 and pBlazIDE assemblers include directives that provide advanced control.

Locating Code at a Specific Address

In some cases, application code must be assigned to a specific instruction address. Examples include the code located at the reset vector, 0, and the interrupt vector, 3FF.

As an example, [Table 11-1](#) shows the PicoBlaze assembler directive to locate code at the interrupt vector for both the KCPSM3 and pBlazIDE formats.

Table 11-1: Assembler Directives to Locate Code

KCPSM3	pBlazIDE
ADDRESS 3FF	ORG \$3FF

Naming or Aliasing Registers

The PicoBlaze microcontroller has 16 general-purpose registers named s0 through sF. To improve code clarity and to enable easy code re-use, re-name or alias the PicoBlaze register with a variable name. Naming registers also prevents unintended use of a register and the associated data corruption, both improving code quality and reducing debugging efforts.

[Table 11-2](#) shows how to alias a register, s5 in this case, to a variable name, myregname. Both KCPSM3 and pBlazIDE formats are shown.

Table 11-2: Assembler Directives to Name or Alias Registers

KCPSM3	pBlazIDE
NAMEREG s5, myregname	myregname EQU s5

In the KCPSM3 assembler, the NAMEREG directive is applied in-line with the code. Before the NAMEREG directive, the register is named using the 'sX' style. Following the directive, *only* the new name applies. It is also possible to rename a register again (i.e., NAMEREG old_regname, new_regname) and only the new name applies in the subsequent program lines.

Defining Constants

Similar to renaming registers, assign names to constant values. By defining names for constants, it is easier to understand and document the PicoBlaze code rather than using the constant values in the code. Similarly, assigning names to registers and constants simplifies code maintenance. Updating the value assigned to a constant is easier if the constant is declared just once rather than searching for each occurrence in the application code.

[Table 11-3](#) shows how to define a constant called myconstant and assign the value 80 hexadecimal. Both KCPSM3 and pBlazIDE formats are shown.

Table 11-3: Assembler Directives to Name or Alias Registers

KCPSM3	pBlazIDE
<code>CONSTANT myconstant, 80</code>	<code>myconstant EQU \$80</code>

Naming the Program ROM Output File

The PicoBlaze assembler generates object code and formats the results for some form of internal memory within the FPGA. In general, the internal memory is block RAM, as described in [Chapter 7, “Instruction Storage Configurations.”](#)

KCPSM3

The output files from the KCPSM3 assembler are always named according to the source program name. For example, an assembly program named `myprog.psm` produces output files called `myprog.vhd`, `myprog.v`, etc. for the various output formats.

pBlazIDE

The pBlazIDE assembler provides a directive, using the keyword `VHDL`, to explicitly name the target output file and the VHDL entity name, as shown in [Figure 11-1](#). The `template.vhd` file contains the VHDL template for the program ROM. The `target.vhd` file is the output VHDL file, derived from the `template.vhd` file, which contains the initialization values created by assembling the PicoBlaze code. Finally, `entity_name` is the VHDL entity name used to name program ROM.

<code>VHDL "template.vhd", "target.vhd", "entity_name"</code>

[Figure 11-1: pBlazIDE Directive to Name the VHDL Output File](#)

Defining I/O Ports (pBlazIDE)

To aid modeling and debugging of the interaction between the PicoBlaze microcontroller and the remainder of the FPGA, the pBlazIDE assembler supports some additional directives to describe and define I/O ports. These directives are particularly useful during instruction set simulation, as shown in [Figure 12-1](#).

Input Ports

The DSIN directive defines the name and the port address (or port identification number) for a read-only input port. The DSIN directive models an input port that only connects to the PicoBlaze microcontroller's IN_PORT port. An optional field specifies a text file containing input values used during instruction set simulation. [Figure 11-2](#) provides an example.

```
; pBlazIDE syntax to define an input port
; input_port_name DSIN <port_id#>[, "<input_file_name>"]
;
    switches      DSIN    $00, "switch_inputs.txt"
    readport      DSIN    $1F
```

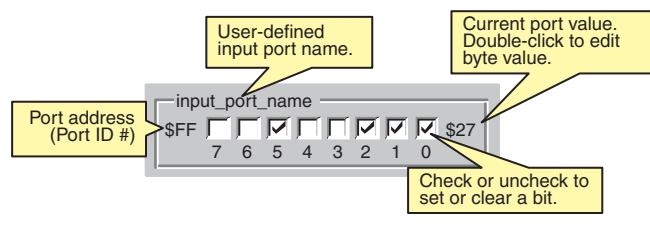
Figure 11-2: Example of pBlazIDE DSIN Directive

As shown in [Figure 11-3](#), the stimulus contained in the specified switch_inputs.txt is rather simple. Each line defines the data for an input (read) operation from the specified port address. Each line represents a single port input operation. Data is specified as decimal, unless prepended with a dollar sign (\$), which indicates the data is hexadecimal. If the simulation reads through all the values provided, the last value is persistent until the end of simulation.

```
$FF
01
02
03
$A5
$5A
```

Figure 11-3: Example File (switch_inputs.txt) to Describe Input Values for Simulation

During instruction set simulation, pBlazIDE displays the input port as shown in [Figure 11-4](#). Edit the input port values during simulation by checking the individual bit values or, to modify the entire byte value, double-click the current port value.



UG129_c11_01_051404

Figure 11-4: The pBlazIDE DSIN Directive Defines an Input Port

Output Ports

The DSOUT directive defines the name and the port address (or port identification number) for a write-only output port. The DSOUT directive models an output port that only connects to the PicoBlaze microcontroller's OUT_PORT port. An optional field

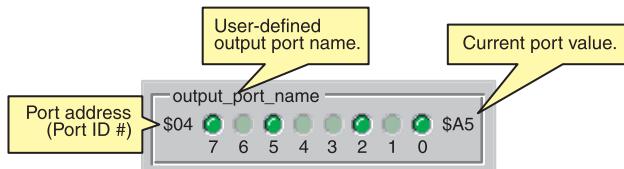
specifies a text file that records the result of any output operations to this during instruction set simulation. [Figure 11-5](#) provides an example.

```
; pBlazIDE syntax to define a write-only output port
; output_port_name DSOUT <port_id#>[, "<output_file_name>"]
;
LEDs          DSOUT $01, "output_values.txt"
writeport     DSOUT $1E
```

[Figure 11-5: Example of pBlazIDE DSOUT Directive](#)

The values recorded in the optional output file are always written as hexadecimal values.

During instruction set simulation, pBlazIDE displays the write-only output port as shown in [Figure 11-6](#). Output ports are write-only and cannot be modified from the graphical interface.



UG129_c11_02_052004

[Figure 11-6: The pBlazIDE DSOUT Directive Defines an Output Port](#)

The DSOUT directive is also useful to create stimulus files later read using DSIN directives in another pBlazIDE application program. For example, to create a stimulus input file containing incrementing data values, create a quick pBlazIDE program that increments a value and writes the value to an output port declared using a DSOUT directive, complete with file specification. Once the program completes, this resulting file can be read by another pBlazIDE program during instruction set simulation using the DSIN directive.

Input/Output Ports

The DSIO directive defines the name and the port address (or port identification number) for an output port. However, the DSIO directive differs from the DSOUT directive in that the PicoBlaze microcontroller can also read DSIO output values. The DSIO directive models an output port that connects to both the PicoBlaze microcontroller's IN_PORT and OUT_PORT ports and has the same port address for input and output operations. An optional field specifies a text file that records the result of any output operations to this during instruction set simulation. [Figure 11-7](#) provides an example.

```
; pBlazIDE syntax to define a readable output port
; input_output_port_name DSIO <port_id#>["<output_file_name>"]

mailbox          DSIO $02, "mailbox_out.txt"
readwrite        DSIO $1D
```

[Figure 11-7: Example of pBlazIDE DSIO Directive](#)

The values recorded in the optional output file are always written as hexadecimal values.

During instruction set simulation, pBlazIDE displays the readable output port as shown in [Figure 11-8](#). The port value can be modified from the graphical interface.

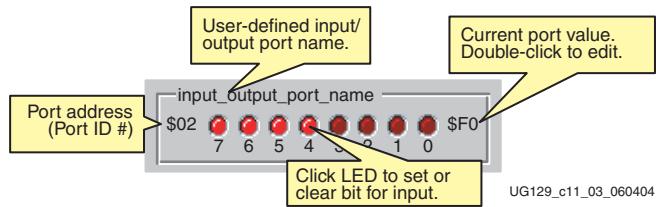


Figure 11-8: The pBlazIDE DSIO Directive Defines an Output Port That Can Be Read Back

Custom Instruction Op-Codes

The pBlazIDE environment supports the INST directive to force the assembler to generate a custom in-line instruction op-code. The INST directive is used when adding custom capabilities to the PicoBlaze reference design. However, adding custom capabilities requires modifications to PicoBlaze hardware.

Simulating PicoBlaze Code

Various tools support PicoBlaze code simulation, each with distinct strengths and weaknesses as described in [Table 12-1](#). For example, the pBlazIDE Instruction Set Simulator (ISS) is best for simulating PicoBlaze operation during code development. As shown in [Figure 12-1](#), the pBlazIDE ISS provides a seamless development environment where assembly code can be quickly tested with full observability. Registers, flags, and memory values appear on the graphical display. Likewise, each of these resources can be modified to enhance testing.

Table 12-1: PicoBlaze Code Simulation Options

Verification Tool	Strengths	Weaknesses
pBlazIDE	<ul style="list-style-type: none">• Ideal for rapid code development• Cycle-accurate Instruction Set Simulation (ISS)• Single-step• Breakpoints• Intimate interaction with PicoBlaze registers, flags, and memory values• Code coverage indicator• Software timing• Basic system-level simulation via file I/O functions• Free!	<ul style="list-style-type: none">• No modeling of custom logic attached to the PicoBlaze microcontroller
ModelSim	<ul style="list-style-type: none">• Holistic simulation of PicoBlaze microcontroller with associated FPGA logic• VHDL and Verilog logic and timing simulation• Cycle and timing accurate with FPGA hardware	<ul style="list-style-type: none">• Requires simulator setup and stimulus files• Digital simulator, not an ideal Instruction Set Simulation environment

Table 12-1: PicoBlaze Code Simulation Options

Verification Tool	Strengths	Weaknesses
Xilinx System Generator	<ul style="list-style-type: none"> • Full PicoBlaze system-level simulation with the System Generator environment • Cycle-accurate Instruction Set Simulation (ISS) • Single-step • Breakpoints • Register and memory viewer 	<ul style="list-style-type: none"> • Primarily only useful if already using Xilinx System Generator
In-system on FPGA	<ul style="list-style-type: none"> • Fast, real-time performance • Ideal for complex interactions • Integrated with peripherals, displays, UARTs, etc. 	<ul style="list-style-type: none"> • Poor visibility of register contents

Furthermore, the pBlazIDE ISS offers full single-step and breakpoint support while viewing the PicoBlaze assembly source code. Evaluate the software timing for end application. Observe code coverage. Simulate basic FPGA interaction using the pBlazIDE DSIN, DSOUT, and DSIO directives (see “Defining I/O Ports (pBlazIDE),” page 78 for more information).

Best of all, the pBlazIDE graphical development environment is free! Download the latest version directly from the Mediatronix website:

<http://www.meditronix.com/pBlazeIDE.htm>

The pBlazIDE ISS does not support full simulation of the PicoBlaze microcontroller embedded with all the other FPGA logic. Fortunately, the PicoBlaze core source files support both VHDL and Verilog simulation using the ModelSim simulator. ModelSim allows the entire design to be simulated, including accurate timing information and textual disassembly features.

If using the Xilinx System Generator software, there is full development and system-level simulation support for the PicoBlaze microcontroller. Refer to “Designing PicoBlaze Microcontroller Application” in the *Xilinx System Generator User Guide* (see Reference 3) for more information.

Instruction Set Simulation with pBlazIDE

The Mediatronix pBlazIDE instruction set simulator (ISS), shown in Figure 12-1, provides complete internal observability during code development. Begin simulation by clicking **Assemble**, as shown in Table 12-2. The pBlazIDE ISS also provides instruction breakpoints, single-stepping, register and RAM observation, code coverage, and software timing.

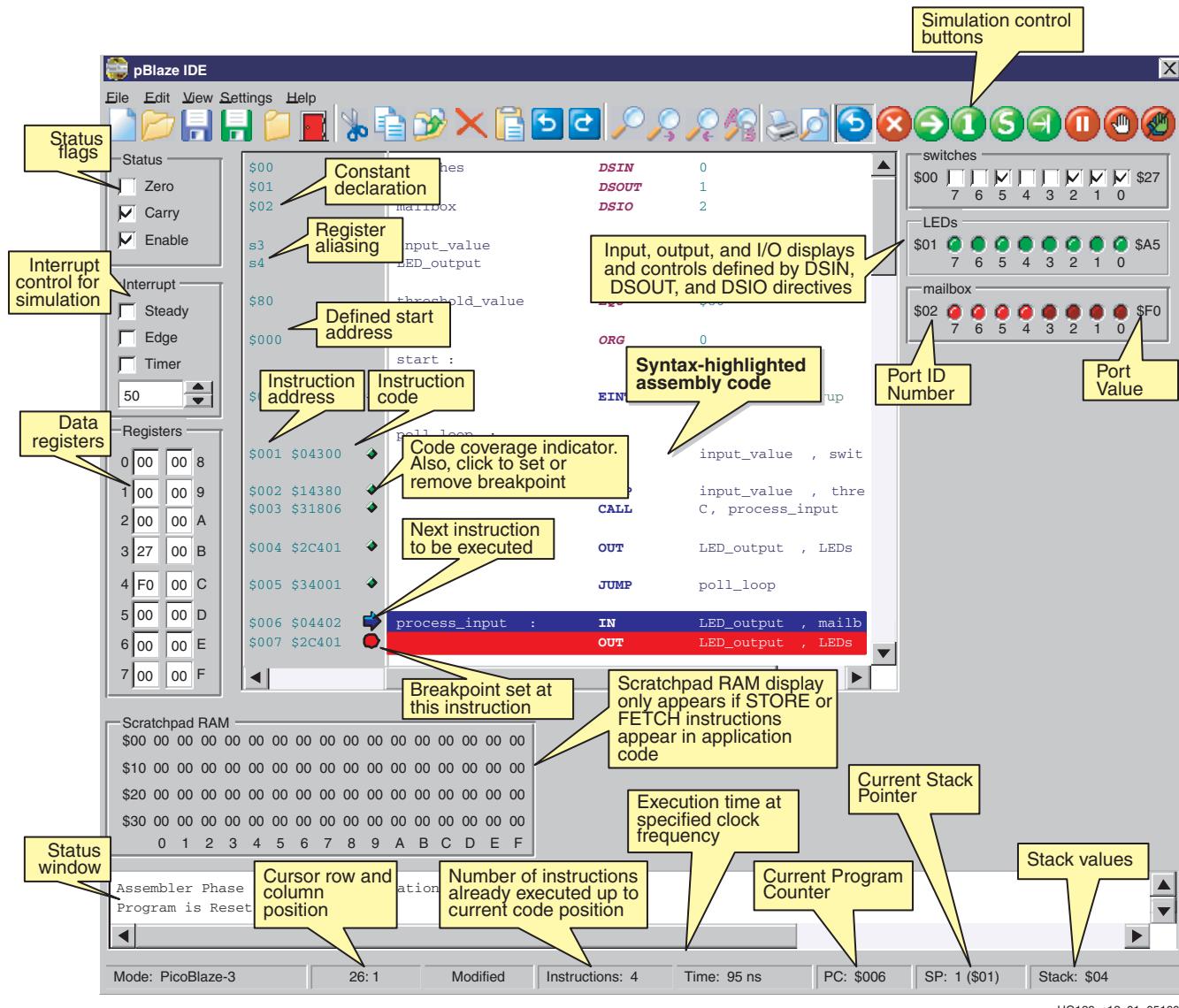


Figure 12-1: The pBlazIDE Instruction Set Simulator (ISS)

Simulator Control Buttons

Table 12-2 shows the various pBlazIDE control buttons and describes their functions.

Table 12-2: pBlazIDE Simulator Control Buttons

Button	Function
	Assemble the open document. If no errors are encountered, the simulator is invoked and the other simulation control buttons are enabled.
	Leave simulator and return to editor. All the simulation control buttons are disabled.

Table 12-2: pBlazIDE Simulator Control Buttons (*Continued*)

Button	Function
Reset 	Reset the simulator. Reset the Program Counter (PC) and Stack Pointer (SP). Register and RAM values are not reset.
Run 	Run the program. The program continues running until an active breakpoint is encountered or if the Reset or Pause button is pressed.
Single Step 	Execute a single instruction. The active register, memory, and I/O displays are updated and the cursor arrow advances to the next instruction. The next instruction to be executed is highlighted in blue and a blue arrow appears to the left of the instruction line. After executing an instruction, the code coverage indicator changes from blue to green. If executing a valid CALL instruction, the program steps into the subroutine function.
Step Over 	Behaves like the Single Step button except that the simulator does not step into CALL instructions. If executing a valid CALL instruction, the program executes the entire subroutine and the display advances to the instruction following the CALL instruction.
Run to Cursor 	Run the program until the program advances to the current cursor location.
Pause 	Pause a running simulation and display the current state of the PicoBlaze microcontroller. Continue the simulation using any of the green action buttons.
Toggle Breakpoint 	Set or remove a breakpoint on the instruction at the current cursor location. The instruction line is highlighted in red and a breakpoint indicator appears to the left of the line. Multiple breakpoints may be simultaneously active. If the simulation reaches an instruction with an active breakpoint, the simulation automatically pauses.
Remove All Breakpoints 	Clear all active breakpoints.

Using the pBlazIDE Instruction Set Simulator with KCPSM3 Programs

The pBlazIDE software primarily supports only a VHDL design flow, which is sufficient for many applications. The KCPSM3 assembler supports Verilog, black box, and Xilinx System Generator design flows in addition to the VHDL flow.

Fortunately, pBlazIDE has an import function that translates a KCPSM3-syntax program into the pBlazIDE syntax. Consequently, it is possible to use the pBlazIDE instruction set simulator to simulate KCPSM3 programs.

Simulating FPGA Interaction with the pBlazIDE Instruction Set Simulator

Although pBlazIDE does not simulate FPGA logic, it does provide basic capabilities to test the PicoBlaze INPUT and OUTPUT operations that connect to FPGA logic. The DSIN, DSOUT, and DSIO directives declare input and output ports, and also provide simple file I/O functions. A stimulus file associated with a DSIN statement simulates data reads during a PicoBlaze INPUT operation. Similarly, the DSOUT statement specifies an output file where PicoBlaze OUTPUT operations are recorded.

Turbocharging Simulation using FPGAs!

Hardware simulators track results with picosecond or nanosecond resolution. In contrast, the PicoBlaze microcontroller is often employed in applications that are less time critical or deliberately slow. For example, a real-time clock is impractical to simulate using a hardware or software simulator. Even UART-based communication is desperately slow relative to a 50 MHz system clock. Likewise, VHDL and Verilog simulation requires accurate stimulus to drive the application.

In test cases, the solution is to simply use the FPGA hardware directly as the testing and debugging medium. While in-system “simulation” is not a replacement for worst-case timing analysis or code-coverage testing, it is possible to recompile a small design in less than a minute and make iterative changes to code and hardware. Using the download interface described in [“Standard Configuration with UART or JTAG Programming Interface” in Chapter 7](#), rapid code changes can be quickly downloaded into the FPGA without recompiling the FPGA hardware design. Likewise, testing happens in the actual environment, along with the other support circuitry, which reduces the burden to create detailed and accurate stimulus models. For example, UART interaction can be tested using the HyperTerminal program on the PC, not just simulated using a VHDL model.

Furthermore, the PicoBlaze microcontroller itself is ideal for an internal test pattern generator, either to test another PicoBlaze core or to test integrated FPGA logic. The PicoBlaze microcontroller can output a test vector, calculate the result expected back from the FPGA, read the actual value from the FPGA, and verify that the actual and expected values match.

Related Materials and References

This appendix provides links to additional information relevant to a PicoBlaze design.

1. **PicoBlaze 8-bit Embedded Microcontroller**
Download PicoBlaze reference designs and additional files.
http://www.xilinx.com/ipcenter/processor_central/picoblaze
2. Mediatronix pBlazIDE Integrated Development Environment for PicoBlaze
<http://www.mediatrix.com/pBlazeIDE.htm>
3. *Xilinx System Generator User Guide*: “Designing PicoBlaze Microcontroller Applications”
http://www.support.xilinx.com/products/software/sysgen/app_docs/user_guide Chapter_7_Section_6.htm
4. MicroBlaze 32-bit Soft Processor Core
<http://www.xilinx.com/microblaze>
5. XAPP466: Using Dedicated Multiplexers in Spartan-3 FPGAs
<http://www.xilinx.com/bvdocs/appnotes/xapp466.pdf>
6. Reconfiguring Block RAMs via JTAG
http://support.xilinx.com/xlnx/xweb/xil_tx_display.jsp?sTechX_ID=krs_blockRAM
7. *XST User Guide*: Chapter 8, “Mixed Language Support”
<http://toolbox.xilinx.com/docsan/xilinx6/books/docs/xst/xst.pdf>

Example Program Templates

The following code templates provide the basic recommended structure for PicoBlaze application programs. Both KCPSM3 and pBlazIDE templates are provided.



If reading this document in Adobe Acrobat,
use the Select Text tool to select code snippets,
then copy and paste the text into your text editor.

KCPSM3 Syntax

Figure B-1 provides a code template for creating PicoBlaze applications using the KCPSM3 assembler.

```

NAMEREG sX, <name> ; Rename register sX with <name>
CONSTANT <name>, 00 ; Define constant <name>, assign value

; ROM output file is always called
; <filename>.vhd

ADDRESS 000           ; Programs always start at reset vector 0

ENABLE_INTERRUPT      ; If using interrupts, be sure to enable
; the INTERRUPT input

BEGIN:
; <<< your code here >>>

JUMP BEGIN          ; Embedded applications never end

ISR: ; An Interrupt Service Routine (ISR) is
; required if using interrupts
; Interrupts are automatically disabled
; when an interrupt is recognized
; Never re-enable interrupts during the ISR
RETURNI_ENABLE        ; Return from interrupt service routine
; Use RETURNI DISABLE to leave interrupts
; disabled

ADDRESS 3FF           ; Interrupt vector is located at highest
; instruction address
JUMP ISR              ; Jump to interrupt service routine, ISR

```

Figure B-1: PicoBlaze Application Program Template for KCPSM3 Assembler

pBlazIDE Syntax

Figure B-2 provides a code template for creating PicoBlaze applications using the pBlazIDE assembler.

```
<name> EQU sX           ; Rename register sX with <name>
<name> EQU $00          ; Define constant <name>, assign value

; name ROM output file generated by pBlazIDE assembler
VHDL "template.vhd", "target.vhd", "entity_name"

<name> DSIN <port_id> ; Create input port, assign port address
<name> DSOUT <port_id>; Create output port, assign port address
<name> DSIO <port_id> ; Create readable output port,
                        ; assign port address

ORG 0; Programs always start at reset vector 0

EINT                         ; If using interrupts, be sure to enable
                                ; the INTERRUPT input

BEGIN:
; <<< your code here >>>

JUMP BEGIN                   ; Embedded applications never end

ISR:                          ; An Interrupt Service Routine (ISR) is
                                ; required if using interrupts
                                ; Interrupts are automatically disabled
                                ; when an interrupt is recognized
                                ; Never re-enable interrupts during the ISR
                                ; Return from interrupt service routine
                                ; Use RETURNI DISABLE to leave interrupts
                                ; disabled

RETI ENABLE

ORG $3FF                     ; Interrupt vector is located at highest
                                ; instruction address
                                ; Jump to interrupt service routine, ISR
```

Figure B-2: PicoBlaze Application Program Template for KCPSM3 Assembler

PicoBlaze Instruction Set and Event Reference

This appendix provides a detailed operational description of each PicoBlaze instruction and the Interrupt and Reset events, including pseudocode for each instruction. The pseudocode assumes that all variable to the right of an assignment symbol (\leftarrow) have the original value before the instruction is executed. The values for variables to the left of an assignment symbol are assigned at the end of the instruction and all assignments occur in parallel, similar to VHDL.

ADD sX, Operand —Add Operand to Register sX

The ADD instruction performs an 8-bit addition of two operands, as shown in [Figure C-1](#). The first operand is any register, which also receives the result of the operation. A second operand is also any register or an 8-bit constant value. The ADD instruction does not use the CARRY as an input, and hence, there is no need to condition the flags before use. Flags are affected by this operation.

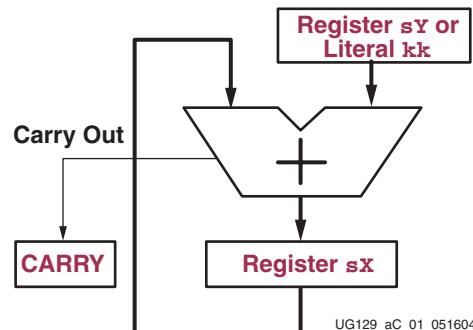


Figure C-1: ADD Operation

Example

Operand is a register location, sY , or an immediate byte-wide constant, kk .

```
ADD sX, sY; Add register.    sX = sX + sY.  
ADD sX, kk; Add immediate.   sX = sX + kk.
```

Description

Operand is added to register sX . The ZERO and CARRY flags are set appropriately.

Pseudocode

```

sX ← (sX + Operand) mod 256; always an 8-bit result

if ( (sX + Operand) > 255 ) then
    CARRY ← 1
else
    CARRY ← 0
endif

if ( ((sX + Operand) = 0) or ((sX + Operand) = 256) ) then
    ZERO ← 1
else
    ZERO ← 0
endif

PC ← PC + 1

```

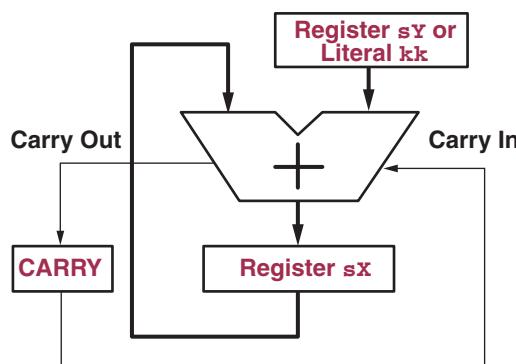
Registers/Flags Altered

Registers: sX, PC

Flags: CARRY, ZERO

ADDCY sX, Operand —Add Operand to Register sX with Carry

The ADDCY instruction performs an addition of two 8-bit operands and adds an additional '1' if the CARRY flag was set by a previous instruction, as shown in [Figure C-2](#). The first operand is any register, which also receives the result of the operation. A second operand is also any register or an 8-bit constant value. Flags are affected by this operation.



UG129_aC_02_051604

Figure C-2: ADDCY Instruction

Example

Operand is a register location, sY, or an immediate byte-wide constant, kk.

```

ADDCY sX, sY; Add register.    sX = sX + sY + CARRY
ADDCY sX, kk; Add immediate.   sX = sX + kk + CARRY

```

Description

Operand and CARRY flag are added to register sX. The ZERO and CARRY flags are set appropriately.

Pseudocode

```

if (CARRY = 1) then
    sX ← (sX + Operand + 1) mod 256; always an 8-bit result
else
    sX ← (sX + Operand) mod 256      ; always an 8-bit result
end if

if ( (sX + Operand + CARRY) > 255 ) then
    CARRY ← 1
else
    CARRY ← 0
endif

if ( ((sX + Operand + CARRY) = 0) or ((sX + Operand + CARRY) = 256) )
then
    ZERO ← 1
else
    ZERO ← 0
endif

PC ← PC + 1

```

Registers/Flags Altered

Registers: sX, PC

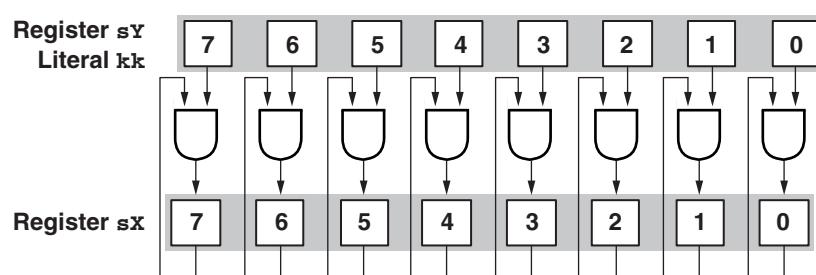
Flags: CARRY, ZERO

Notes

pBlazIDE Equivalent: ADDC

AND sX, Operand — Logical Bitwise AND Register sX with Operand

The AND instruction performs a bitwise logical AND operation between two operands, as shown in [Figure C-3](#). The first operand is any register, which also receives the result of the operation. A second operand is also any register or an 8-bit immediate constant. The ZERO flag is set if the resulting value is zero. The CARRY flag is always cleared by an AND instruction.

**Figure C-3: AND 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.

Examples

```

AND sX, sY ; Logically AND the individual bits of register sX with
              ; the corresponding bits in register sY
AND sX, kk ; Logically AND the individual bits of register sX with
              ; the corresponding bits in the immediate constant kk

```

Pseudocode

```

; logically AND the corresponding bits in sX and the Operand
for (i=0; i<= 7; i=i+1)
{
    sX(i) ← sX(i) AND Operand(i)
}

CARRY ← 0

if (sX = 0) then
    ZERO ← 1
else
    ZERO ← 0
end if

PC ← PC + 1

```

Registers/Flags Altered

Registers: sX, PC

Flags: ZERO, CARRY is always 0

CALL [Condition,] Address — Call Subroutine at Specified Address, Possibly with Conditions

The CALL instruction modifies the normal program execution sequence by jumping to a specified program address. Each CALL instruction must specify the 10-bit address as a three-digit hexadecimal value or a label that the assembler resolves to a three-digit hexadecimal value.

The CALL instruction has both conditional and unconditional variants. A conditional CALL is only performed if a test performed against either the ZERO flag or CARRY flag is true. If unconditional or if the condition is true, the CALL instruction pushes the current value the PC to the top of the CALL/RETURN stack. Simultaneously, the specified CALL location is loaded into the PC.

A subroutine function must exist at the specified address. The subroutine function requires a RETURN instruction to return from the subroutine.

The CALL instruction does not affect the ZERO or CARRY flags. However, if a CALL is performed, the resulting subroutine instructions may modify the flags.

Examples

```

CALL      MYSUB; Unconditionally call MYSUB subroutine
CALL C,   MYSUB; If CARRY flag set, call MYSUB subroutine
CALL NC,  MYSUB; If CARRY flag not set, call MYSUB subroutine
CALL Z,   MYSUB; If ZERO flag set, call MYSUB subroutine
CALL NZ,  MYSUB; If ZERO flag not set, call MYSUB subroutine

```

Condition

Depending on the specified Condition, the program calls the subroutine beginning at the specified Address. If the specified Condition is not met, the program continues to the next instruction.

Table C-1: CALL Instruction Conditions

Condition	Description
<none>	Always true. Call subroutine unconditionally.
C	CARRY = 1. Call subroutine if CARRY flag is set.
NC	CARRY = 0. Call subroutine if CARRY flag is cleared.
Z	ZERO = 1. Call subroutine if ZERO flag is set.
NZ	ZERO = 0. Call subroutine if ZERO flag is cleared.

Pseudocode

```

if (Condition = TRUE) then
    ; push current PC onto top of the CALL/RETURN stack
    ; TOS = Top of Stack
    TOS ← PC
    ; load PC with specified Address
    PC ← Address
else
    PC ← PC + 1
endif

```

Registers/Flags Altered

Registers: PC, CALL/RETURN stack

Flags: Not affected

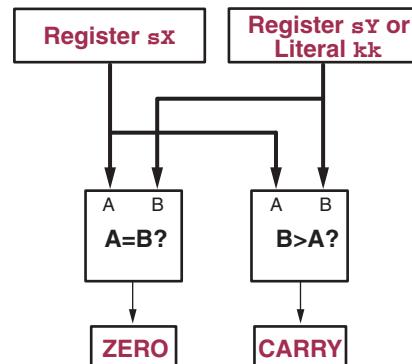
Notes

The maximum number of nested subroutine calls is 31 levels, due to the depth of the CALL/RETURN stack.

COMPARE sX, Operand — Compare Operand with Register sX

The COMPARE instruction performs an 8-bit comparison of two operands, as shown in Figure C-4. The first operand, sX, is any register and this register is NOT affected by the

COMPARE operation. The second operand is also any register or an 8-bit immediate constant value. Only the flags are affected by this operation.



UG129_aC_05_051604

Figure C-4: COMPARE Operation

Register *sX* is compared against Operand. The *ZERO* flag is set when Register *sX* and Operand are identical. The *CARRY* flag is set when Operand is larger than Register *sX*, where both Operand and Register *sX* are evaluated as unsigned integers.

Example

Operand is a register location, *sY*, or an immediate byte-wide constant, *kk*.

```

COMPARE sx, sY; Compare sx against sY.
COMPARE sx, kk; Compare sx against immediate constant, kk.
  
```

Pseudocode

```

if ( Operand > sx ) then
    CARRY ← 1
else
    CARRY ← 0
endif

if ( sx = Operand ) then
    ZERO ← 1
else
    ZERO ← 0
endif

PC ← PC + 1
  
```

Registers/Flags Altered

Registers: PC only. No data registers affected.

Flags: CARRY, ZERO

Notes

pBlazIDE Equivalent: COMP

The COMPARE instruction is only supported on PicoBlaze microcontrollers for Spartan-3, Virtex-II, and Virtex-II Pro FPGAs.

DISABLE INTERRUPT — Disable External Interrupt Input

The DISABLE_INTERRUPT instruction clears the interrupt enable (IE) flag. Consequently, the PicoBlaze microcontroller ignores the INTERRUPT input. Use this instruction to temporarily disable interrupts during timing-critical code segments. Use the ENABLE_INTERRUPT instruction to re-enable interrupts.

Example

```
DISABLE_INTERRUPT; Disable interrupts
```

Pseudocode

```
INTERRUPT_ENABLE ← 0
PC ← PC + 1
```

Registers/Flags Altered

Registers: PC

Flags: INTERRUPT_ENABLE

Notes

PBlazIDE Equivalent: DINT

ENABLE INTERRUPT — Enable External Interrupt Input

The ENABLE_INTERRUPT instruction sets the interrupt enable (IE) flag. Consequently, the PicoBlaze microcontroller recognizes the INTERRUPT input. Before using this instruction, a suitable interrupt service routine (ISR) must be associated with the interrupt vector address, 3FF.

Never issue the ENABLE_INTERRUPT instruction from within an ISR.

Example

```
ENABLE_INTERRUPT; Enable interrupts
```

Pseudocode

```
INTERRUPT_ENABLE ← 1
PC ← PC + 1
```

Registers/Flags Altered

Registers: PC

Flags: INTERRUPT_ENABLE

Notes

PBlazIDE Equivalent: EINT

FETCH sX, Operand — Read Scratchpad RAM Location to Register sX

The FETCH instruction reads scratchpad RAM location specified by Operand into register sX, as shown in [Figure C-5](#). There are 64 scratchpad RAM locations. The two most-significant bits of Operand, bits 7 and 6, are discarded and the RAM address is truncated to

the least-significant six bits of Operand, bits 5 to bit 0. Consequently, a FETCH operation from address FF is equivalent to a FETCH operation from address 3F.

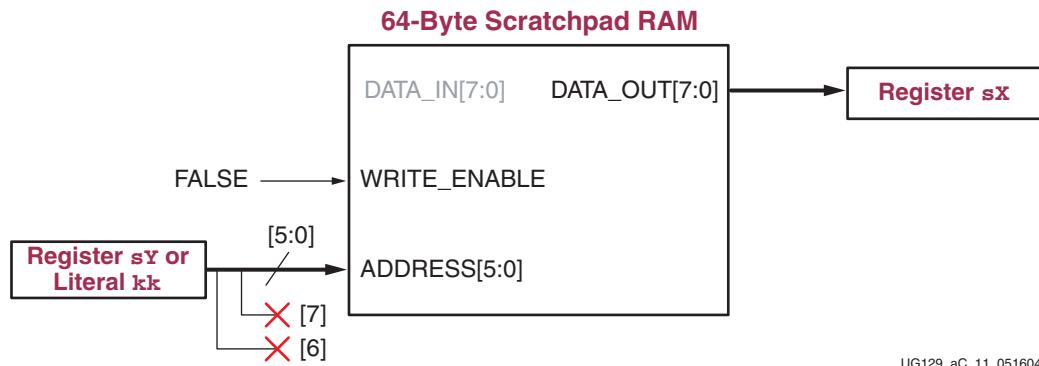


Figure C-5: FETCH Operation

Examples

```

FETCH sX, (sY) ; Read scratchpad RAM location specified by the
; contents of register sY into register sX

FETCH sX, kk     ; Read scratchpad RAM location specified by the
; immediate constant kk into register sX

```

Pseudocode

```

sX ← Scratchpad_RAM [Operand[5:0]]
PC ← PC + 1

```

Registers/Flags Altered

Registers: PC

Flags: None

Notes

pBlazIDE Equivalent: The instruction mnemonic, FETCH, is the same for both KCPSM3 and pBlazIDE. However, the instruction syntax for indirect addressing is slightly different. The KCPSM3 syntax places parentheses around the indirect address while the pBlazIDE syntax uses no parentheses.

KCPSM3 Instruction	PBlazIDE Instruction
FETCH sX, (sY)	FETCH sX, sY

The FETCH instruction is only supported on PicoBlaze microcontrollers for Spartan-3, Virtex-II, and Virtex-II Pro FPGAs.

INPUT sX, Operand — Set PORT_ID to Operand, Read value on IN_PORT into Register sX

The INPUT instruction sets the PORT_ID output port to either the value specified by register sY or by the immediate constant kk. The instruction then reads the value on the IN_PORT input port into register sX. Flags are not affected by this operation.

Interface logic decodes the PORT_ID address to provide the correct value on IN_PORT.

Examples

```
INPUT sx, sy ; Read the value on IN_PORT into register sx, set PORT_ID
; to the contents of sy
INPUT sx, kk ; Read the value on IN_PORT into register sx, set PORT_ID
; to the immediate constant kk
```

Pseudocode

```
PORT_ID ← Operand
sx ← IN_PORT
PC ← PC + 1
```

Registers/Flags Altered

Registers: sx, PC

Flags: None

Notes

pBlazIDE Equivalent: IN

The READ_STROBE output is asserted during the second CLK cycle of the two-cycle INPUT operation.

INTERRUPT Event, When Enabled

The interrupt event is not an instruction but the response of the PicoBlaze microcontroller to an external interrupt input. If the INTERRUPT_ENABLE flag is set, then a recognized logic level on the INTERRUPT input generates an Interrupt Event. The action essentially generates a subroutine CALL to the most-significant instruction address, location 1023 (\$3FF). The currently executing instruction is allowed to complete. Once the PicoBlaze microcontroller recognizes the interrupt input, the INTERRUPT_ENABLE flag is automatically reset. The next instruction in the normal program flow is preempted by the Interrupt Event and the current PC is pushed onto the CALL/RETURN stack. The PC is loaded with all ones and the program calls the instruction at the most-significant address.

The instruction at the most-significant address must jump to the interrupt service routine (ISR).

Pseudocode

```
; only respond to INTERRUPT input if INTERRUPT_ENABLE flag is set
if ( (INTERRUPT_ENABLE = 1)and (INTERRUPT input = High) ) then
    ; clear the INTERRUPT_ENABLE flag
    INTERRUPT_ENABLE ← 0

    ; push the current program counter (PC) to the stack
    ; TOS = Top of Stack
    TOS ← PC

    ; preserve the current CARRY and ZERO flags
    PRESERVED_CARRY ← CARRY
    PRESERVED_ZERO ← ZERO

    ; load program counter (PC) with interrupt vector ($3FF)
    PC ← $3FF
```

```
endif
```

Registers/Flags Altered

Registers: PC, CALL/RETURN stack

Flags: CARRY, ZERO, INTERRUPT_ENABLE

Notes

The PicoBlaze microcontroller asserts the INTERRUPT_ACK output on the second CLK cycle of the two-cycle Interrupt Event, as shown in [Figure 4-3, page 45](#). This signal is optionally used to clear any hardware interrupt flags.

The programmer must ensure that a RETURNI instruction is only performed in response to a previous interrupt. Otherwise, the PC stack may not contain a valid return address.

Do not use the RETURNI instruction to return from a subroutine CALL. Instead, use the RETURN instruction.

Because an interrupt event may happen at any time, the values of the CARRY and ZERO flags cannot be predetermined. Consequently, the corresponding Interrupt Service Routine (ISR) must not depend on specific values for the CARRY and ZERO flags.

JUMP [Condition,] Address — Jump to Specified Address, Possibly with Conditions

The JUMP instruction modifies the normal program execution sequence by jumping to a specified program address. Each JUMP instruction must specify the 10-bit address as a three-digit hexadecimal value or a label that the assembler resolves to a three-digit hexadecimal value.

The JUMP instruction has both conditional and unconditional variants. A conditional JUMP is only performed if a test performed against either the ZERO flag or CARRY flag is true. If unconditional or if the condition is true, the JUMP instruction loads the specified jump address into the Program Counter (PC).

The JUMP instruction does not affect the CALL/RETURN stack.

The JUMP instruction does not affect the ZERO or CARRY flags.

Example

```

JUMP      NEW_LOCATION; Unconditionally jump to NEW_LOCATION
JUMP C,   NEW_LOCATION; If CARRY flag set, jump to NEW_LOCATION
JUMP NC,  NEW_LOCATION; If CARRY flag not set, jump to NEW_LOCATION
JUMP Z,   NEW_LOCATION; If ZERO flag set, jump to NEW_LOCATION
JUMP NZ,  NEW_LOCATION; If ZERO flag not set, jump to NEW_LOCATION

```

Condition

Depending on the specified condition, the program jumps to the instruction at the specified address. If the specified condition is not met, the program continues on to the next instruction.

Table C-2: JUMP Instruction Conditions

Condition	Description
<none>	Always true. Jump unconditionally.
C	CARRY = 1. Jump if CARRY flag is set.
NC	CARRY = 0. Jump if CARRY flag is cleared.
Z	ZERO = 1. Jump if ZERO flag is set.
NZ	ZERO = 0. Jump if ZERO flag is cleared.

Pseudocode

```

if (Condition = TRUE) then
    PC ← Address
else
    PC ← PC + 1
endif

```

Registers/Flags Altered

Registers: PC

Flags: Not affected

LOAD sX, Operand — Load Register sX with Operand

The LOAD instruction loads the contents of any register. The new value is either the contents of any other register or an immediate constant. The LOAD instruction has no effect on the status flags.

Because the LOAD instruction does not affect the flags, use it to reorder and assign register contents at any stage of the program execution. Loading a constant into a register via the LOAD instruction has no impact on program size or performance and is the easiest method to assign a value or to clear a register.

Examples

```
LOAD sX, sY; Move the contents of register sY to register sX
LOAD sX, kk; Load register sX with the immediate constant kk
```

Pseudocode

```
sX ← Operand
```

```
PC ← PC + 1
```

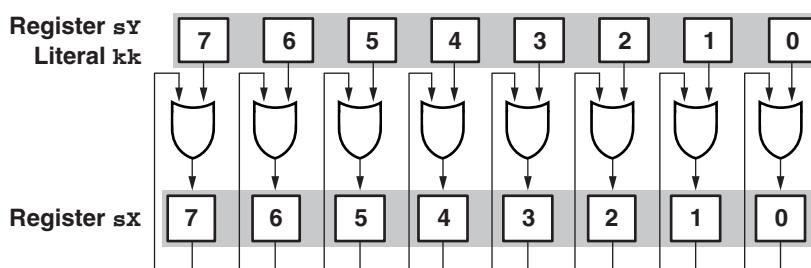
Registers/Flags Altered

Registers: sX, PC

Flags: Not affected

OR sX, Operand — Logical Bitwise OR Register sX with Operand

The OR instruction performs a bitwise logical OR operation between two operands, as shown in Figure C-6. The first operand is any register, which also receives the result of the operation. A second operand is also any register or an 8-bit immediate constant. The ZERO flag is set if the resulting value is zero. The CARRY flag is always cleared by an OR instruction.



UG129_aC_07_051604

Figure C-6: OR Operation

The OR instruction provides a way to force the setting any bit of the specified register, which can be used to form control signals.

Examples

```
OR sX, sY ; Logically OR the individual bits of register sX with the
             ; corresponding bits in register sY
OR sX, kk ; Logically OR the individual bits of register sX with the
             ; corresponding bits in the immediate constant kk
```

Pseudocode

```

; logically OR the corresponding bits in sX and the Operand
for (i=0; i<= 7; i=i+1)
{
    sX(i) ← sX(i) OR Operand(i)
}

CARRY ← 0

if (sX = 0) then
    ZERO ← 1
else
    ZERO ← 0
end if

PC ← PC + 1

```

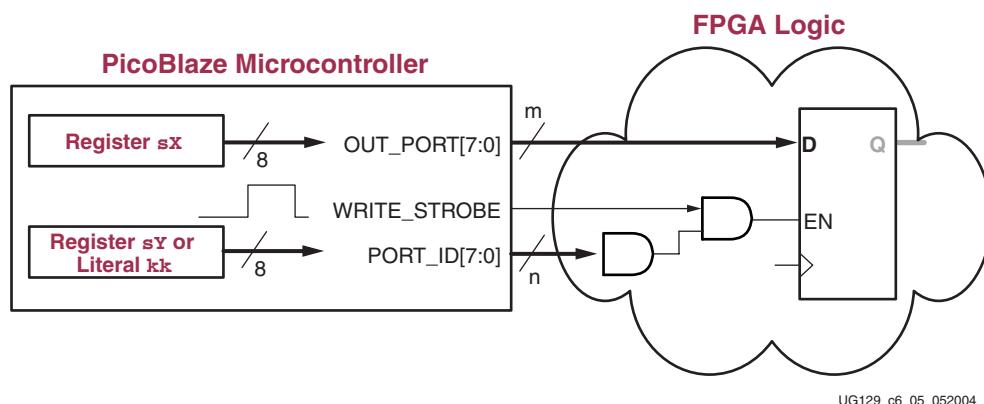
Registers/Flags Altered

Registers: sX, PC

Flags: ZERO, CARRY is always 0

OUTPUT sX, Operand — Write Register sX Value to OUT_PORT, Set PORT_ID to Operand

The OUTPUT instruction sets the PORT_ID port address to the value specified by either the register sY or the immediate constant kk. The instruction writes the contents of register sX to the OUT_PORT output port. FPGA logic captures the output value by decoding the PORT_ID value and WRITE_STROBE output, as shown in Figure C-7.

**Figure C-7: OUTPUT Operation and FPGA Interface Logic****Examples**

```
OUTPUT sX, sY ; Write register sX to OUT_PORT, set PORT_ID to the
; contents of sY
```

```
OUTPUT sX, kk ; Write register sX to OUT_PORT, set PORT_ID to the
; immediate constant kk
```

Pseudocode

```

PORT_ID ← Operand
OUT_PORT ← SX
PC ← PC + 1

```

Registers/Flags Altered

Registers: PC

Flags: None

Notes

pBlazIDE Equivalent: OUT

The WRITE_STROBE output is asserted during the second CLK cycle of the two-cycle OUTPUT operation.

RESET Event

The reset event is not an instruction but the response of the PicoBlaze microcontroller when the RESET input is High. A RESET Event restarts the PicoBlaze microcontroller and clears various hardware elements, as shown in [Table C-3](#).

A RESET Event is automatically generated immediately following FPGA configuration, initiated by the FPGA's internal Global Set/Reset (GSR) signal. After configuration, the FPGA application generates RESET Event by asserting the RESET input before a rising CLK clock edge.

Table C-3: PicoBlaze Reset Values

Resource	RESET Event Effect
General-purpose Registers	Unaffected.
Program Counter	0
ZERO Flag	0
CARRY Flag	0
INTERRUPT_ENABLE Flag	0
Scratchpad RAM	Unaffected.
Program Store	Unaffected.
CALL/RETURN Stack	Stack Pointer reset.

The general-purpose registers, the scratchpad RAM, and the program store are not affected by a RESET Event. The CALL/RETURN stack is a circular buffer, although a RESET Event essentially resets the CALL/RETURN stack pointer.

Pseudocode

```

if (RESET input = High) then
; clear Program Counter
PC ← 0

; disable the INTERRUPT input by clearing the INTERRUPT_ENABLE flag
INTERRUPT_INPUT ← 0

```

```

; clear the ZERO and CARRY flags
ZERO ← 0
CARRY ← 0
endif

```

Registers/Flags Altered

Registers: PC, CALL/RETURN stack

Flags: CARRY, ZERO, INTERRUPT_ENABLE

RETURN [Condition] — Return from Subroutine Call, Possibly with Conditions

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 that the program executes the instruction following the CALL instruction that called the subroutine. The RETURN instruction has no effect on the status of the flags.

The RETURN instruction has both conditional and unconditional variants. A conditional RETURN is only performed if a test performed against either the ZERO flag or CARRY flag is true. If unconditional or if the condition is true, the RETURN instruction pops the return address from the top of the CALL/RETURN stack into the PC. The popped value forces the program to return to the instruction immediately following the original subroutine CALL.

Ensure that a RETURN is only performed in response to a previous CALL instruction so that the CALL/RETURN stack contains a valid address.

The RETURN instruction does not affect the ZERO or CARRY flags. The flag values set prior to the RETURN instruction are maintained and available after the return from the subroutine call.

Condition

Depending on the specified condition, the program returns from a subroutine call. If the specified condition is not met, the program continues on to the next instruction.

Table C-4: RETURN Instruction Conditions

Condition	Description
<none>	Always true. Return from called subroutine unconditionally.
C	CARRY = 1. Return from called subroutine if CARRY flag is set.
NC	CARRY = 0. Return from called subroutine if CARRY flag is cleared.
Z	ZERO = 1. Return from called subroutine if ZERO flag is set.
NZ	ZERO = 0. Return from called subroutine if ZERO flag is cleared.

Pseudocode

```

if (Condition = TRUE) then
    ; pop the top of the CALL/RETURN stack into PC
    ; TOS = Top of Stack
    PC ← TOS + 1; incremented value from Top of Stack

```

```

    else
        PC ← PC + 1
    endif

```

Registers/Flags Altered

Registers: PC, CALL/RETURN stack

Flags: Not affected

Notes

Do not use the RETURN instruction to return from an interrupt. Instead, use the RETURNI instruction.

PBlazIDE Equivalent: RET, RET C, RET NC, RET Z, RET NZ

RETURNI [ENABLE/DISABLE] — Return from Interrupt Service Routine and Enable or Disable Interrupts

The RETURNI instruction is a special variation of the RETURN instruction. It concludes an interrupt service routine. The RETURNI instruction is unconditional and pops the return address from the top of the CALL/RETURN stack into the PC. The return address points to the instruction preempted by an Interrupt Event. The RETURNI instruction restores the CARRY and ZERO flags to the values preserved by the Interrupt Event.

The ENABLE or DISABLE operand defines whether the INTERRUPT input is re-enabled or remains disabled when returning from the interrupt service routine (ISR).

Example

```

RETURNI ENABLE; Return from interrupt, re-enable interrupts
RETURNI DISABLE; Return from interrupt, leave interrupts disabled

```

Pseudocode

```

; pop the top of the CALL/RETURN stack into PC
PC ← TOS

; restore the flags to their pre-interrupt values
CARRY ← PRESERVED_CARRY
ZERO ← PRESERVED_ZERO

; if "ENABLE" specified, re-enable interrupts
if (ENABLE = TRUE) then
    INTERRUPT_ENABLE ← 1
else
    INTERRUPT_ENABLE ← 0
endif

```

Registers/Flags Altered

Registers: PC, CALL/RETURN stack

Flags: CARRY, ZERO, INTERRUPT_ENABLE

Notes

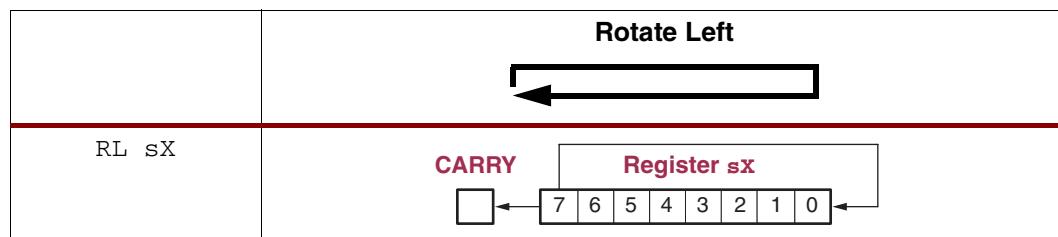
Do not use the RETURNI instruction to return from a subroutine CALL. Instead, use the RETURN instruction.

PBlazIDE Equivalent: RETI ENABLE, RETI DISABLE

RL sX — Rotate Left Register sX

The rotate left instruction operates on any single data register. Each bit in the specified register is shifted left by one bit position, as shown in [Table C-5](#). The most-significant bit, bit 7, shifts both into the CARRY bit and into the least-significant bit, bit 0.

Table C-5: Rotate Left (RL) Operation



Example

RL sX; Rotate left. Bit sX[7] copied into CARRY.

Pseudocode

```
CARRY ← sX[7]

sX ← { sX[6:0], sX[7] }

if ( sX = 0 ) then
    ZERO ← 1
else
    ZERO ← 0
endif

PC ← PC + 1
```

Registers/Flags Altered

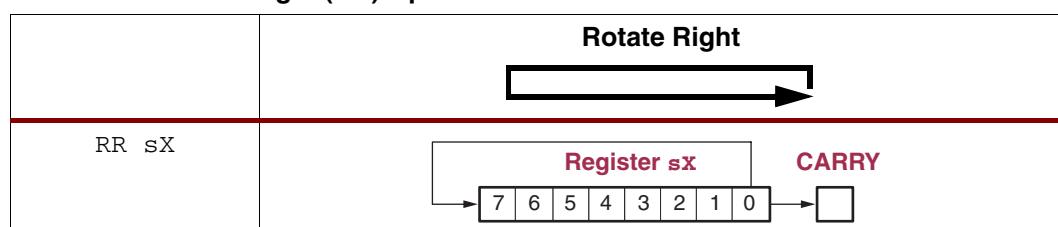
Registers: sX, PC

Flags: CARRY, ZERO

RR sX — Rotate Right Register sX

The rotate right instruction operates on any single data register. Each bit in the specified register is shifted right by one bit position, as shown in [Table C-6](#). The least-significant bit, bit 0, shifts both into the CARRY bit and into the most-significant bit, bit 7.

Table C-6: Rotate Right (RR) Operation



Example

RR sX; Rotate right. Bit sX[0] copied into CARRY

Pseudocode

```

CARRY ← sX[0]

sX ← {sX[0], sX[7:1]}

if ( sX = 0 ) then
    ZERO ← 1
else
    ZERO ← 0
endif

PC ← PC + 1

```

Registers/Flags Altered

Registers: sX, PC

Flags: CARRY, ZERO

SL[0 | 1 | X | A] sX — Shift Left Register sX

There are four variants of the shift left instruction, as shown in [Table C-7](#), that operate on any single data register. Each bit in the specified register is shifted left by one bit position. The most-significant bit, bit 7, shifts into the CARRY bit. The last character of the instruction mnemonic—i.e., ‘0’, ‘1’, ‘X’, or ‘A’—indicates the value shifted into the least-significant bit, bit 7.

Table C-7: Shift Left Operations

Shift Left	
SL0 sX	<p style="text-align: center;">Shift Left with ‘0’ fill.</p> <p style="text-align: center;">CARRY Register sX</p>
SL1 sX	<p style="text-align: center;">Shift Left with ‘1’ fill.</p> <p style="text-align: center;">CARRY Register sX</p>
SLX sX	<p style="text-align: center;">Shift Left, eXtend bit 0.</p> <p style="text-align: center;">CARRY Register sX</p>
SLA sX	<p style="text-align: center;">Shift Left through All bits, including CARRY.</p> <p style="text-align: center;">CARRY Register sX</p>

The ZERO flag is always 0 after executing the SL1 instruction because register sX is never zero.

Examples

SL0 sX; Shift left. 0 shifts into LSB, MSB shifts into CARRY.

SL1 sX; Shift left. 1 shifts into LSB, MSB shifts into CARRY.
 SLX sX; Shift left. LSB shifts into LSB, MSB shifts into CARRY.
 SLA sX; Shift left. CARRY shifts into LSB, MSB shifts into CARRY.

Pseudocode

```

case (INSTRUCTION)
when "SL0"
  LSB ← 0
when "SL1"
  LSB ← 1
when "SLX"
  LSB ← sX(7)
when "SLA"
  LSB ← CARRY
end case

CARRY ← sX[7]

sX ← {sX[6:0], LSB}

if ( sX = 0 ) then
  ZERO ← 1
else
  ZERO ← 0
endif

PC ← PC + 1

```

Registers/Flags Altered

Registers: sX, PC

Flags: CARRY, ZERO

SR[0 | 1 | X | A] sX — Shift Right Register sX

There are four variants of the shift right instruction, as shown in [Table C-8](#), that operate on any single data register. Each bit in the specified register is shifted right by one bit position. The least-significant bit, bit 0, shifts into the CARRY bit. The last character of the instruction mnemonic—i.e., ‘0’, ‘1’, ‘X’, or ‘A’—indicates the value shifted into the most-significant bit, bit 7.

Table C-8: Shift Right Operations

Shift Right					
SR0 sX	<p>Shift Right with ‘0’ fill.</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">Register sX</td> <td style="text-align: center;">CARRY</td> </tr> <tr> <td style="text-align: center;">‘0’ →</td> <td style="text-align: center;">7 6 5 4 3 2 1 0 → □</td> </tr> </table>	Register sX	CARRY	‘0’ →	7 6 5 4 3 2 1 0 → □
Register sX	CARRY				
‘0’ →	7 6 5 4 3 2 1 0 → □				
SR1 sX	<p>Shift Right with ‘1’ fill.</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">Register sX</td> <td style="text-align: center;">CARRY</td> </tr> <tr> <td style="text-align: center;">‘1’ →</td> <td style="text-align: center;">7 6 5 4 3 2 1 0 → □</td> </tr> </table>	Register sX	CARRY	‘1’ →	7 6 5 4 3 2 1 0 → □
Register sX	CARRY				
‘1’ →	7 6 5 4 3 2 1 0 → □				

Table C-8: Shift Right Operations (Continued)

Shift Right	
SRX sx	<p style="text-align: center;">Shift Right, sign eXtend.</p>
SRA sx	<p style="text-align: center;">Shift Right through All bits, including CARRY.</p>

The ZERO flag is always 0 after executing the SR1 instruction because register sx is never zero.

Example

```

SR0 sx; Shift right. 0 shifts into MSB, LSB shifts into CARRY.
SR1 sx; Shift right. 1 shifts into MSB, LSB shifts into CARRY.
SRX sx; Shift right MSB shifts into MSB, LSB shifts into CARRY.
SRA sx; Shift right CARRY shifts into MSB, LSB shifts into CARRY.

```

Pseudocode

```

case (INSTRUCTION)
when "SR0"
    MSB ← 0
when "SR1"
    MSB ← 1
when "SRX"
    MSB ← sx(7)
when "SRA"
    MSB ← CARRY
end case

CARRY ← sx[0]

sx ← {MSB, sx[7:1]}

if ( sx = 0 ) then
    ZERO ← 1
else
    ZERO ← 0
endif

PC ← PC + 1

```

Registers/Flags Altered

Registers: sx, PC

Flags: CARRY, ZERO

STORE sX, Operand — Write Register sX Value to Scratchpad RAM Location

The STORE instruction writes register sX to the scratchpad RAM location specified by Operand, as shown in [Figure C-8](#). There are 64 scratchpad RAM locations. The two most-significant bits of Operand, bits 7 and 6, are discarded and the RAM address is truncated to the least-significant six bits of Operand, bits 5 to bit 0. Consequently, a STORE operation to address FF is equivalent to a STORE operation to address 3F.

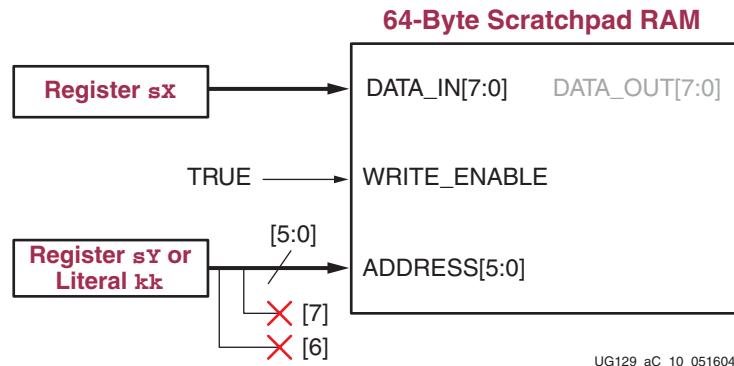


Figure C-8: STORE Operation

Examples

```
STORE sX, (sY) ; Write register sX to scratchpad RAM location
; specified by the contents of register sY

STORE sX, kk    ; Write register sX to scratchpad RAM location
; specified by the immediate constant kk
```

Pseudocode

```
Scratchpad_RAM[Operand[5:0]] ← sX  
PC ← PC + 1
```

Registers/Flags Altered

Registers: sX, PC

Flags: None

Notes

pBlazIDE Equivalent: The instruction mnemonic, STORE, is the same for both KCPSM3 and pBlazIDE. However, the instruction syntax for indirect addressing is slightly different. The KCPSM3 syntax places parentheses around the indirect address while the pBlazIDE syntax uses no parentheses.

KCPSM3 Instruction	PBlazIDE Instruction
STORE sX, (sY)	STORE sX, sY

The STORE instruction is only supported on PicoBlaze microcontrollers for Spartan-3, Virtex-II, and Virtex-II Pro FPGAs.

SUB sX, Operand —Subtract Operand from Register sX

The SUB instruction performs an 8-bit subtraction of two operands, as shown in Figure C-9. The first operand is any register, which also receives the result of the operation. The second operand is also any register or an 8-bit constant value. Flags are affected by this operation. The SUB instruction does not use the CARRY as an input, and therefore there is no need to condition the flags before use.

The CARRY flag, when set, indicates when an underflow (borrow) occurred.

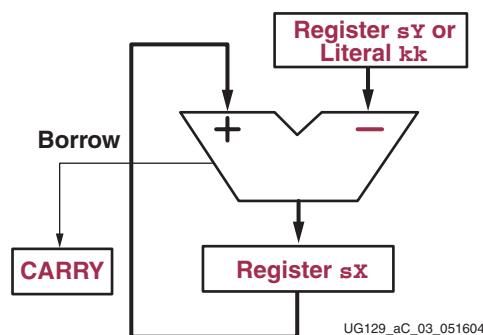


Figure C-9: SUB Instruction

Examples

Operand is a register location, sY, or an immediate byte-wide constant, kk.

```
SUB sX, sY; Subtract register.    sX = sX - sY.  
SUB sX, kk; Subtract immediate.  sX = sX - kk.
```

Description

Operand is subtracted from register sX. The ZERO and CARRY flags are set appropriately.

Pseudocode

```
sX ← (sX - Operand) mod 256; always an 8-bit result  
  
if ( (sX - Operand) < 0 ) then  
    CARRY ← 1  
else  
    CARRY ← 0  
endif  
  
if ( (sX - Operand) = 0 ) then  
    ZERO ← 1  
else  
    ZERO ← 0  
endif  
  
PC ← PC + 1
```

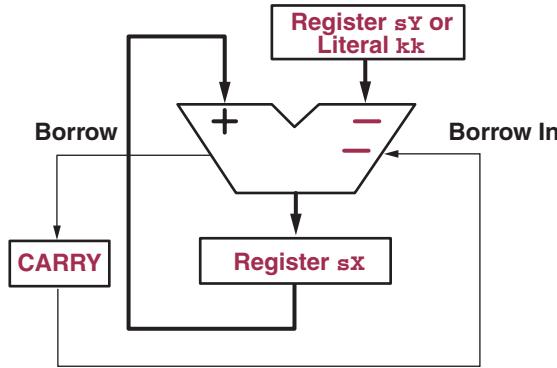
Registers/Flags Altered

Registers: sX, PC

Flags: CARRY, ZERO

SUBCY sX, Operand —Subtract Operand from Register sX with Borrow

The SUBCY instruction performs an 8-bit subtraction of two operands and subtracts an additional ‘1’ if the CARRY (borrow) flag was set by a previous instruction, as shown in [Figure C-10](#). The first operand is any register, which also receives the result of the operation. The second operand is also any register or an 8-bit constant value. Flags are affected by this operation.



UG129_aC_04_051604

Figure C-10: SUBCY Instruction

Examples

Operand is a register location, sY, or an immediate byte-wide constant, kk.

```
SUBCY sX, sY; Subtract register.    sX = sX - sY - CARRY
SUBCY sX, kk; Subtract immediate.   sX = sX - kk - CARRY
```

Description

Operand and CARRY flag are subtracted from register sX. The ZERO and CARRY flags are set appropriately.

Pseudocode

```

if (CARRY = 1) then
    sX ← (sX - Operand - 1) mod 256; always an 8-bit result
else
    sX ← (sX - Operand) mod 256      ; always an 8-bit result
endif

if ( (sX - Operand - CARRY) < 0 ) then
    CARRY ← 1
else
    CARRY ← 0
endif

if ( ((sX - Operand - CARRY) = 0) or ((sX - Operand - CARRY) = -256) )
then
    ZERO ← 1
else
    ZERO ← 0
endif

PC ← PC + 1

```

Registers/Flags Altered

Registers: sX

Flags: CARRY, ZERO

Notes

pBlazIDE Equivalent: SUBC

TEST sX, Operand — Test Bit Location in Register sX, Generate Odd Parity

The TEST instruction performs two related but separate operations. The ZERO flag indicates the result of a bitwise logical AND operation between register sX and the specified Operand. The ZERO flag is set if the resulting bitwise AND is zero, as shown in [Figure C-11](#). The CARRY flag indicates the XOR of the result, as shown in [Figure C-12](#), which behaves like an odd parity generator.

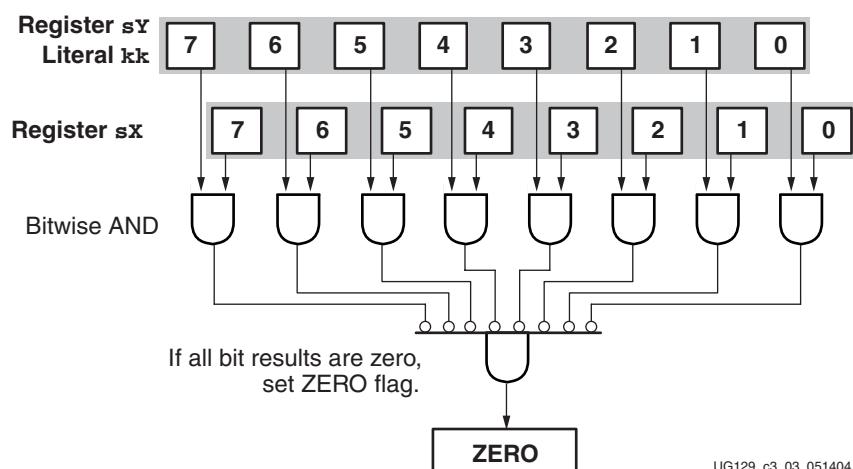


Figure C-11: ZERO Flag Logic for TEST Instruction

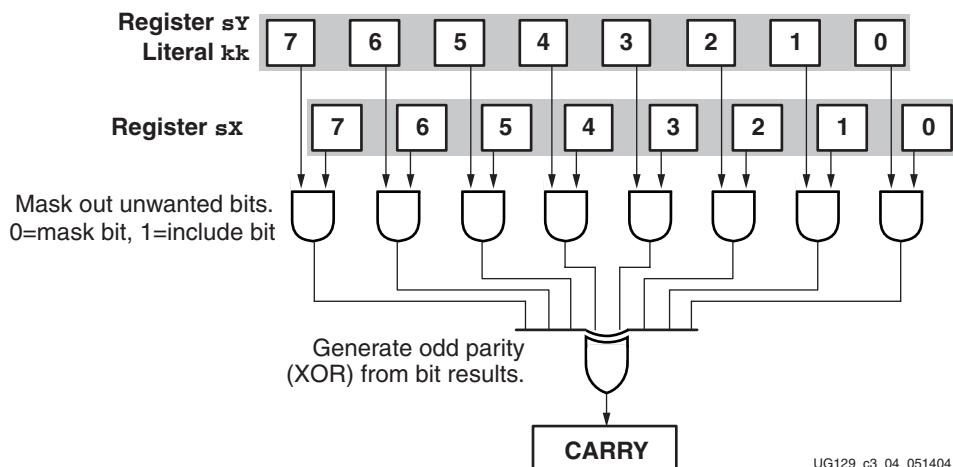


Figure C-12: CARRY Flag Logic for TEST Instruction

Examples

```
TEST sX, sY ; Test register sX using register sY as the test mask
TEST sX, kk ; Test register sX using the immediate constant kk as the
; test mask
```

Pseudocode

```
; logically AND the corresponding bits in sX and the Operand
for (i=0; i<= 7; i=i+1)
{
    AND_TEST(i) ← sX(i) AND Operand(i)
}

if (AND_TEST = 0) then
    ZERO ← 1
else
    ZERO ← 0
end if

; logically XOR the corresponding bits in sX and the Operand
XOR_TEST = 0
for (i=0; i<= 7; i=i+1)
{
    XOR_TEST ← AND_TEST(i) XOR XOR_TEST
}

if (XOR_TEST = 1) then; generate odd parity
    CARRY ← 1           ; odd number of one's, CARRY=1 for odd parity
else
    CARRY ← 0           ; even number of one's, CARRY=0 for odd parity
end if

PC ← PC + 1
```

Registers/Flags Altered

Registers: PC

Flags: ZERO, CARRY

The TEST instruction is only supported on PicoBlaze microcontrollers for Spartan-3, Virtex-II, and Virtex-II Pro FPGAs.

XOR sX, Operand — Logical Bitwise XOR Register sX with Operand

The XOR instruction performs a bitwise logical XOR operation between two operands, as shown in Figure C-13. The first operand is any register, which also receives the result of the operation. A second operand is also any register or an 8-bit immediate constant. The ZERO flag is set if the resulting value is zero. The CARRY flag is always cleared by an XOR instruction.

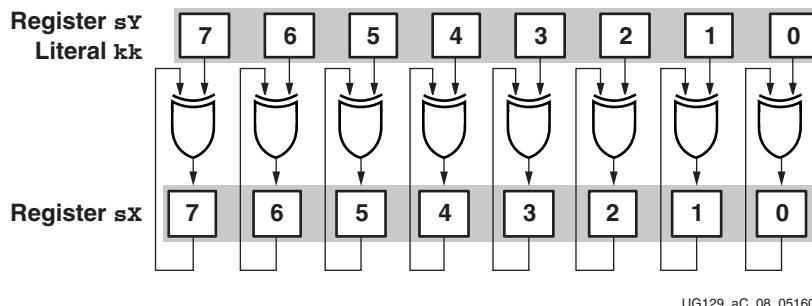


Figure C-13: XOR Operation

The XOR operation inverts bits contained in a register, which is used in forming control signals.

Examples

```

XOR sX, sY ; Logically XOR the individual bits of register sX with
              ; the corresponding bits in register sY
XOR sX, kk ; Logically XOR the individual bits of register sX with
              ; the corresponding bits in the immediate constant kk

```

Pseudocode

```

; logically XOR the corresponding bits in sX and the Operand
for (i=0; i<= 7; i=i+1)
{
    sX(i) ← sX(i) XOR Operand(i)
}

CARRY ← 0

if (sX = 0) then
    ZERO ← 1
else
    ZERO ← 0
end if

PC ← PC + 1

```

Registers/Flags Altered

Registers: sX, PC

Flags: ZERO, CARRY is always 0

Instruction Codes

[Table D-1](#) provides the 18-bit instruction code for every PicoBlaze instruction.

Table D-1: PicoBlaze Instruction Codes

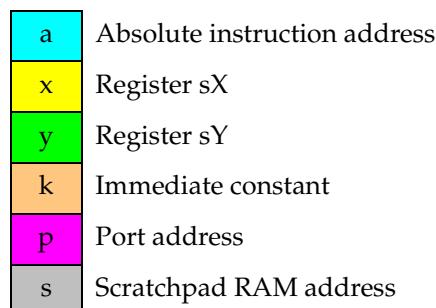
Instruction	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD sX,kk	0	1	1	0	0	0	x	x	x	x	k	k	k	k	k	k	k	k
ADD sX,sY	0	1	1	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0
ADDCY sX,kk	0	1	1	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k
ADDCY sX,sY	0	1	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
AND sX,kk	0	0	1	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k
AND sX,sY	0	0	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
CALL	1	1	0	0	0	0	0	0	a	a	a	a	a	a	a	a	a	a
CALL C	1	1	0	0	0	1	1	0	a	a	a	a	a	a	a	a	a	a
CALL NC	1	1	0	0	0	1	1	1	a	a	a	a	a	a	a	a	a	a
CALL NZ	1	1	0	0	0	0	1	0	1	a	a	a	a	a	a	a	a	a
CALL Z	1	1	0	0	0	0	1	0	0	a	a	a	a	a	a	a	a	a
COMPARE sX,kk	0	1	0	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k
COMPARE sX,sY	0	1	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
DISABLE INTERRUPT	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ENABLE INTERRUPT	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
FETCH sX, ss	0	0	0	1	1	0	x	x	x	x	0	0	s	s	s	s	s	s
FETCH sX,(sY)	0	0	0	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
INPUT sX,(sY)	0	0	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
INPUT sX,pp	0	0	0	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p
JUMP	1	1	0	1	0	0	0	0	a	a	a	a	a	a	a	a	a	a
JUMP C	1	1	0	1	0	1	1	0	a	a	a	a	a	a	a	a	a	a
JUMP NC	1	1	0	1	0	1	1	1	a	a	a	a	a	a	a	a	a	a
JUMP NZ	1	1	0	1	0	1	0	1	a	a	a	a	a	a	a	a	a	a

Table D-1: PicoBlaze Instruction Codes (Continued)

Instruction	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JUMP Z	1	1	0	1	0	1	0	0	a	a	a	a	a	a	a	a	a	a
LOAD sX,kk	0	0	0	0	0	0	x	x	x	x	k	k	k	k	k	k	k	k
LOAD sX,sY	0	0	0	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0
OR sX,kk	0	0	1	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k
OR sX,sY	0	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
OUTPUT sX,(sY)	1	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
OUTPUT sX,pp	1	0	1	1	0	0	x	x	x	x	P	P	P	P	P	P	P	P
RETURN	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
RETURN C	1	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
RETURN NC	1	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
RETURN NZ	1	0	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0
RETURN Z	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
RETURNI DISABLE	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RETURNI ENABLE	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
RL sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	0	1
RR sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	1	0
SL0 sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	1	1
SL1 sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	1	1
SLA sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	0	0
SLX sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	1	0
SR0 sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	1	0
SR1 sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	1	1
SRA sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	0	0
SRX sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	0	0
STORE sX, ss	1	0	1	1	1	0	x	x	x	x	0	0	s	s	s	s	s	s
STORE sX,(sY)	1	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
SUB sX,kk	0	1	1	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k
SUB sX,sY	0	1	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
SUBCY sX,kk	0	1	1	1	1	0	x	x	x	x	k	k	k	k	k	k	k	k
SUBCY sX,sY	0	1	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
TEST sX,kk	0	1	0	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k
TEST sX,sY	0	1	0	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0

Table D-1: PicoBlaze Instruction Codes (*Continued*)

Instruction	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XOR sX,kk	0	0	1	1	1	0	x	x	x	x	k	k	k	k	k	k	k	k
XOR sX,sY	0	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0



Register and Scratchpad RAM Planning Worksheets

This appendix provides worksheets to plan register assignment and allocation for a PicoBlaze application. A similar worksheet is also provided to plan scratchpad RAM assignment and allocation.

Registers

Reg.	Description
s0	
s1	
s2	
s3	
s4	
s5	
s6	
s7	
s8	
s9	
sA	
sB	
sC	
sD	
sE	
sF	

Scratchpad RAM

Loc.	Description	Loc.	Description
00		20	
01		21	
02		22	
03		23	
04		24	
05		25	
06		26	
07		27	
08		28	
09		29	
0A		2A	
0B		2B	
0C		2C	
0D		2D	
0E		2E	
0F		2F	
10		30	
11		31	
12		32	
13		33	
14		34	
15		35	
16		36	
17		37	
18		38	
19		39	
1A		3A	
1B		3B	
1C		3C	
1D		3D	
1E		3E	
1F		3F	