8-Bit Pipelined Picoblaze Validation Plan

Jim Rowe, Victoria Van Gaasbeck, Danny Hale

ECE571-FALL 2021

12/07/2021

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
idex_operation	Asserts is reset to LOAD	Good

idex_reg_x_out	Asserts is reset to zero	Good
idex_reg_y_out	Asserts is reset to zero	Good
idex_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/OUTPUT
		Use randomization to test each picoblaze enumeration in opcode_t	No
AluAssert.sv	shift_test.hex	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.hex	Check CALL opcode using carry and zero flags	Yes
ExAssert.sv	test_call.hex	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.hex	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
#  idex_operation: LOAD should be LOAD
#  idex_reg_x_out: 00 should be 00
#  idex_reg_y_out: 00 should be 00
#  idex_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$time, idu_shift_constant wrong
  Time: 4825 ns Started: 4825 ns  Scope: alt_rojo_tb.dut File: N:/ECE571/ece571f21/vvan/FinalProject/picoblaze_project_bugs/kcpsm_rojo.svp Line: 277
* Error:                4825idu_implied_value: 01, instruction.instr_type.reg_const.constant: 00
  Time: 4825 ns Started: 4825 ns  Scope: alt_rojo_tb.dut File: N:/ECE571/ece571f21/vvan/FinalProject/picoblaze_project_bugs/kcpsm_rojo.svp Line: 251
             4825        LOAD s1,00  z = 0 c = 0 || shiftbit = 0 || shiftop = SA || reg cont = 0c
* Error:                4835$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:                4835idu_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:                4845idu_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.

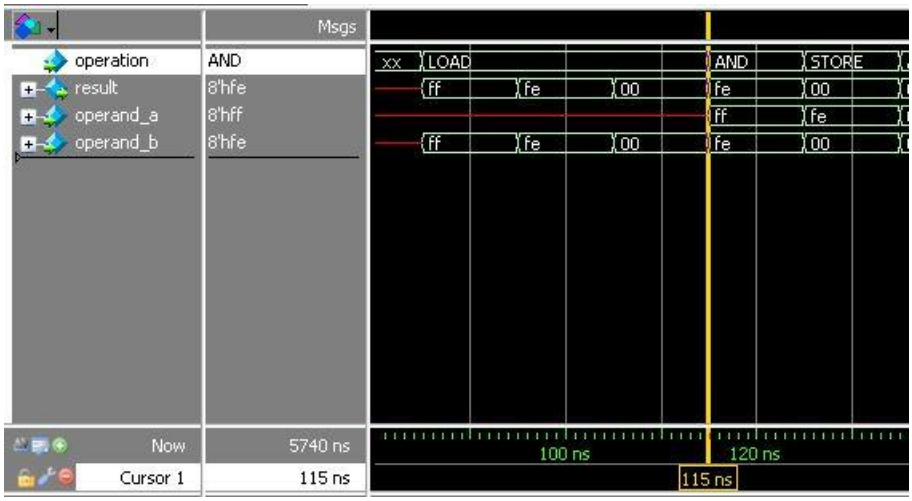

Figure 3: Unbroken model showing the result of and'ing 0xff and 0xfe

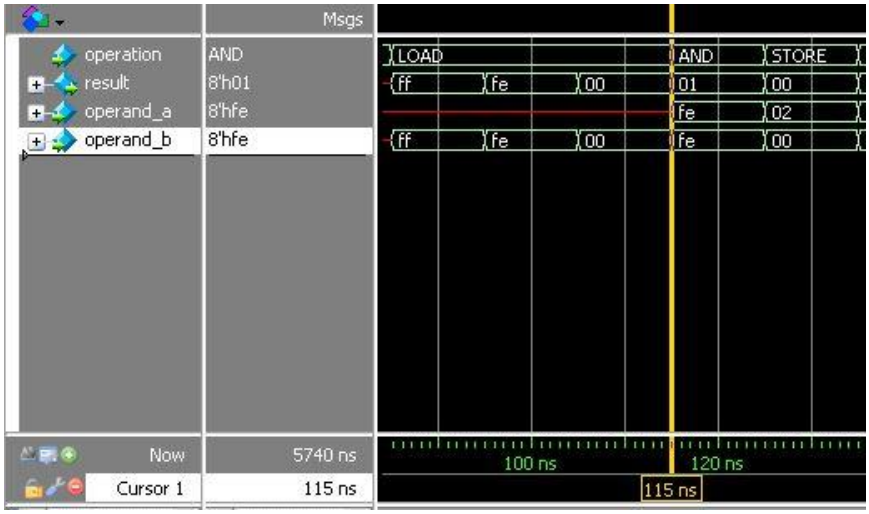

Figure 4: Broken model showing the result holding a negation of operand_b

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

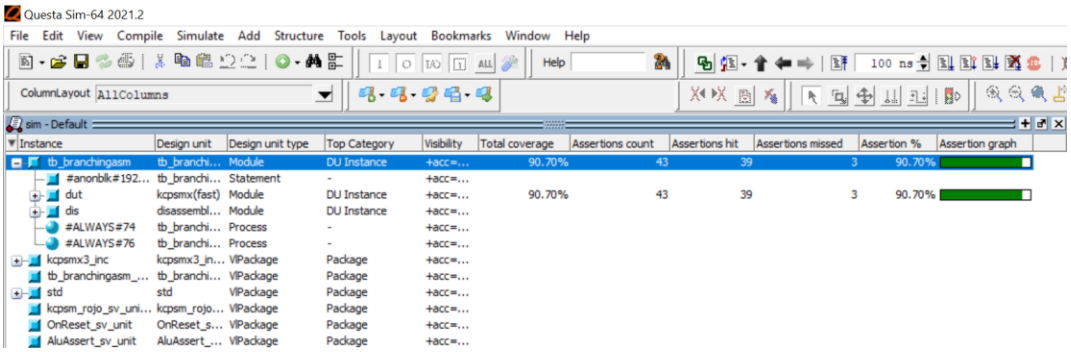

Figure 5: Questasim Assertion Coverage

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.