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 idex_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.