ECE 593: Fundamentals of PreSilicon Validation
Verification Plan 2.0: Pipelined Picoblaze

Group 3: Yashodhan Wagle, Ramaa Potnis, Supreet Gulavani

---

*This document is the second revision of the verification plan which was turned in earlier. While certain sections remain the same, there is additional information about other sections. Refer to the document Verification Plan - 1.0 to see the changes

The main idea of this project is to input an instruction and to check whether the instruction functions as intended,

1. **Description of Verification Levels:**The design being verified is the pipelined picoblaze-RojoBlaze microprocessor. The general architecture is as follows:

- 8-bit data path
- 16 addressable registers
- 18-bit instruction word
- Maximum of 1024 instructions supported in a single program
- The architecture features CISC like flags.

For more details regarding the design the Architectural documentation provided along with this submission is to be referenced.

The verification plan for the Pipelined Picoblaze fall into the following major headings:

2. **Description of Verification Levels:**

   The design that is being implemented is not such a large design. Therefore after a consensus among the team members, it was decided that the design would be verified along the levels of hierarchy.

   If time there is time and resources are present, verification at unit level would also be performed.

   Additionally, SystemVerilog assertions can also be implemented to verify the design.

3. **Required Tools:**

   The tools required for verifying this design consist of a single software simulation engine- with a valid license to run it, one workstation, and a test case language(which for this project is SystemVerilog). The language should talk with the simulation engine through the engine's API that can provide means to drive the inputs and check the outputs.

4 . Risks and Dependencies

For the given time bracket, the design is large, hence not all verification aspects cannot be covered.

Certain alterations to the pre-existing HDL have to be made.

5. Functions to be verified

Critical Functions: Reset, valid opcode, legal instruction

Secondary Functions: Branch taken/not taken, PC Rollover

Non-Verified Functions: Disassmebler

6. Specific tests and methods:

Method for testing the *Instruction Fetch* stage: Load the memory, simulate the output behavior ID stage and check for valid output.

Method for testing the *Instruction Decode* stage: Send a valid instruction, valid data for the register read access request to simulate the behavior of IF (instruction), and register file. Check for the expected decoded output

Method for testing the *Execute* stage: 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 results.

Method for testing the *Write Back* stage: We are going toThis unit will 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.

7. Functions to be verified:Test Matrix

Test	Description
reset	Test Reset Expected behaviour: PC will reset to 0, flags will be cleared, interrupts will be disabled
branch_taken	Implement a branching (taken) condition Expected behaviour: PC will jump to the specified location for the next instruction.
branch_not_taken	Implement a branching (not taken) condition Expected behaviour: PC will continue to increment.
pc_rollover	Implement a program past the 3FF boundary Expected behaviour: PC will rollback to 000

Test	Description
no_op_id	Test the No-operation instr decode Expected behaviour: No operations will take place for further stages
jump_id	Test the JUMP instr decode Expected behaviour: Opcode for JUMP will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
jump_z_id	Test the JUMP Z instr decode Expected behaviour: Opcode for JUMP Z will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
jump_nz_id	Test the JUMP NZ instr decode Expected behaviour: Opcode for JUMP NZ will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
jump_c_id	Test the JUMP C instr decode Expected behaviour: Opcode for JUMP C will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
jump_nc_id	Test the JUMP NC instr decode Expected behaviour: Opcode for JUMP NC will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
call_id	Test the CALL instr decode Expected behaviour: Opcode for CALL will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
call_z_id	Test the CALL Z instr decode Expected behaviour: Opcode for CALL Z will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
call_nz_id	Test the CALL NZ instr decode Expected behaviour: Opcode for CALL NZ will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
call_c_id	Test the CALL C instr decode Expected behaviour: Opcode for CALL C will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
call_nc_id	Test the CALL NC instr decode Expected behaviour: Opcode for CALL NC will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
return_id	Test the RETURN instr decode

	Expected behaviour: Opcode for RETURN will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
return_z_id	Test the RETURN Z instr decode Expected behaviour: Opcode for RETURN Z will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
return_nz_id	Test the RETURN NZ instr decode Expected behaviour: Opcode for RETURN NZ will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
return_c_id	Test the RETURN C instr decode Expected behaviour: Opcode for RETURN C will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
return_nc_id	Test the RETURN NC instr decode Expected behaviour: Opcode for RETURN NC will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr)
add_id	Test the ADD instr decode Expected behaviour: Opcode for ADD will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr). Opcode will be the same regardless of the operands (be literals or from registers)
addcy_id	Test the ADDCY instr decode Expected behaviour: Opcode for ADDCY will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr). Opcode will be the same regardless of the operands (be literals or from registers)
sub_id	Test the SUB instr decode Expected behaviour: Opcode for SUB will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr). Opcode will be the same regardless of the operands (be literals or from registers)
subcy_id	Test the SUBCY instr decode Expected behaviour: Opcode for SUBCY will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr). Opcode will be the same regardless of the operands (be literals or from registers)
compare_id	Test the COMPARE instr decode Expected behaviour: Opcode for COMPARE will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr). Opcode will be the same regardless of the operands (be literals or from registers)
returni_en_id	Test the RETURNI ENABLE instr decode Expected behaviour: Opcode for RETURNI ENABLE will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).

returni_d_id	Test the RETURN DISABLE instr decode Expected behaviour: Opcode for RETURN DISABLE will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
intp_en_id	Test the ENABLE INTERRUPT instr decode Expected behaviour: Opcode for ENABLE INTERRUPT will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
intp_d_id	Test the DISABLE INTERRUPT instr decode Expected behaviour: Opcode for DISABLE INTERRUPT will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
load_id	Test the LOAD instr decode Expected behaviour: Opcode for LOAD will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
and_id	Test the AND instr decode Expected behaviour: Opcode for AND will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
or_id	Test the OR instr decode Expected behaviour: Opcode for OR will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
xor_id	Test the XOR instr decode Expected behaviour: Opcode for XOR will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
test_id	Test the TEST instr decode Expected behaviour: Opcode for TEST will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
store_id	Test the STORE instr decode Expected behaviour: Opcode for STORE will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
fetch_id	Test the FETCH instr decode Expected behaviour: Opcode for FETCH will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
sr0_id	Test the SR0 instr decode Expected behaviour: Opcode for SR0 will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
sr1_id	Test the SR1 instr decode Expected behaviour: Opcode for SR1 will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).

srx_id	Test the SRX instr decode Expected behaviour: Opcode for SRX will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
sra_id	Test the SRA instr decode Expected behaviour: Opcode for SRA will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
rr_id	Test the RR instr decode Expected behaviour: Opcode for RR will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
sl0_id	Test the SL0 instr decode Expected behaviour: Opcode for SL0 will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
sl1_id	Test the SL1 instr decode Expected behaviour: Opcode for SL1 will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
slx_id	Test the SLX instr decode Expected behaviour: Opcode for SLX will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
sla_id	Test the SLA instr decode Expected behaviour: Opcode for SLA will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
rl_id	Test the RL instr decode Expected behaviour: Opcode for RL will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
input_id	Test the INPUT instr decode Expected behaviour: Opcode for INPUT will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).
output_id	Test the OUTPUT instr decode Expected behaviour: Opcode for OUTPUT will be loaded in IDEX register along with operands X and Y (if Y exists for the given instr).

Test	Description

add	Test addition with literal/sY using various operands that cover all bits. Expected behaviour: Correct output will be saved in the sX register and C and Z flags are set accordingly
addcy	Test addition with literal/sY along with carry using various operands that cover all bits Expected behaviour: Correct output will be saved in the sX register and C and Z flags are set accordingly
sub	Test subtracting literal/sY from sX with various operands that cover all bits Expected behaviour: Correct output will be saved in the sX register and C (set if result is negative) and Z flags are set accordingly
subcy	Test subtracting literal/sY and carry from sX with various operands that cover all bits Expected behaviour: Correct output will be saved in the sX register and C (set if result is negative) and Z flags are set accordingly
and	Test bit-wise logical AND between sX and literal/sY with various operands that cover all bits Expected behaviour: Correct output will be saved in the sX register. C = 0, Z is set if result is 0
or	Test bit-wise logical OR between sX and literal/sY with various operands that cover all bits Expected behaviour: Correct output will be saved in the sX register. C = 0, Z is set if result is 0
xor	Test bit-wise logical XOR between sX and literal/sY with various operands that cover all bits Expected behaviour: Correct output will be saved in the sX register. C = 0, Z is set if result is 0
sr0	Test 0 padding to MSB Expected behaviour: C = LSB bit and Z is set if all bits 0
sr1	Test 1 padding to MSB Expected behaviour: C = LSB bit and Z is set if all bits 0
srx	Test MSB padding to MSB Expected behaviour: C = LSB bit and Z is set if all bits 0
sra	Test Carry padding to MSB Expected behaviour: C = LSB bit and Z is set if all bits 0
rr	Test rotate to right Expected behaviour: C = LSB bit and Z is set if all bits 0

sl0	Test 0 padding to LSB Expected behaviour: C = MSB bit and Z is set if all bits 0
sl1	Test 1 padding to LSB Expected behaviour: C = MSB bit and Z is set if all bits 0
slx	Test LSB padding to LSB Expected behaviour: C = MSB bit and Z is set if all bits 0
sla	Test Carry padding to LSB Expected behaviour: C = MSB bit and Z is set if all bits 0
rl	Test rotate to left Expected behaviour: C = MSB bit and Z is set if all bits 0

Tests	Description
r0_load_store	Load a literal in register s0, store it to the scratchpad memory and load it back again
r1_load_store	Load a literal in register s1, store it to the scratchpad memory and load it back again
r2_load_store	Load a literal in register s2, store it to the scratchpad memory and load it back again
r3_load_store	Load a literal in register s3, store it to the scratchpad memory and load it back again
r4_load_store	Load a literal in register s4, store it to the scratchpad memory and load it back again
r5_load_store	Load a literal in register s5, store it to the scratchpad memory and load it back again
r6_load_store	Load a literal in register s6, store it to the scratchpad memory and load it back again
r7_load_store	Load a literal in register s7, store it to the scratchpad memory and load it back again
r8_load_store	Load a literal in register s8, store it to the scratchpad memory and load it back again
r9_load_store	Load a literal in register s9, store it to the scratchpad memory and load it back again

rA_load_store	Load a literal in register sA, store it to the scratchpad memory and load it back again
rB_load_store	Load a literal in register sB, store it to the scratchpad memory and load it back again
rC_load_store	Load a literal in register sC, store it to the scratchpad memory and load it back again
rD_load_store	Load a literal in register sD, store it to the scratchpad memory and load it back again
rE_load_store	Load a literal in register sE, store it to the scratchpad memory and load it back again
rF_load_store	Load a literal in register sF, store it to the scratchpad memory and load it back again

Tests	Description
data_haz_stall	Test for a hazard solved by stalling
data_haz_fwd	Test for a hazard solved by forwarding

8. Type of verification:

We intend to drive inputs and validate the data with checkers at the output. We're also thinking about putting a few checkers on some of the design's internal states. We believe that a gray box approach is the best solution due to our expertise with the RTL's internal workings and checker placement. Gray box verification allows us to combine the benefits of both black box and white box testing.

9. Verification strategy:

Deterministic, random, and formal verification are all viable technolo-
gies for this design. A complete random environment can be very tedious and an overkill for this design, and also a complete deterministic environment can be not completely viable. Thus a mixture of both random and deterministic testing is chosen for this design

10. Abstraction levels:

The entire infrastructure will be based on System Verilog.

There will be a testbench for every stage of the pipeline (i.e. IF, ID, EX and WB stage). We will be developing a common Makefile for environment setup and setting parameters of the testbench. The output is saved in a transcript containing the simulations.
For this design, we intend to do instruction-level abstraction. An instruction which is 18-bit long will be fed to the memory

11. Coverage requirements

Coverage goals for this design are determined by the types of test cases defined in the Functions to be verified section. The goals require that verification tests create all the test mentioned in that section.
It will be ensured that valid data and address values are covered.. Since the verification approach is gray box testing, coverpoints can also be set for any internal signal, one that is crucial to the design, as needed.

12. Resource requirements

The resource requirement for this design will need 3 verification engineers. A single workstation will be required to run the simulation engine.

13. Output Checking

Output would be checked at System level using a reference module which will be created by the team.
Inside the verification environment, a scoreboard is also created.

14, Schedule details

Approximately, 4 weeks would be needed to complete the verification as mentioned in all the above sections.
Following would be the timeline we intend to stick to.

Time	Task
Week 6	Verification plan turn in
Week 7	Waveform generation
Week 8	Code drop - 1
Week 9	Code drop - 2
Week 10	Final submission- Final code, second revision of verification plan, architecture description document, cross reference document and the final report