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	Test each instruction individually, checks expected result from the register file. R-type instruction: Test involving access to all 8 registers, for each instruction Test the access to R0 special register. I-type Instruction: Test LD/ST instruction Test for branch instructions, +/- displacement, rollover from last accessible address. Test for +/-/0 immediate values
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.