**RISC-V Class Project Phase 3 – Verification**

Phase 3 of the Class Project involves creating a Verification program for the RISC-V instruction set. This is an assembly language program which will exercise a subset of the instruction set. This process is very similar to one of the critical functions of computer design, which is verifying the design using a test program. To check the quality of verification, each student will be given a unique hardware design (an IA model) which has a set of errors. The goal of the test program is to detect and report all of the possible errors in the hardware, several of which occur in the provided design.

1. **Import and Build the Sample Project**

The following steps must be followed in order to prepare the project.

1. Build Assembler (ia) and Disassembler (ia) for the standardname2 project if they are not already built. This will create a correct Assembler which must be used to build the test program.
2. Import the project standardname3 from the H: drive. Note that this file will not become available until Phase 2 has been successfully submitted.
3. Build Model Compilation (ia) and Simulator (ia) for the standardname3 project. This will give you a simulator to use in Run -> Debug Configurations called standardname3.ia.standardname3-ia-isimulator. Use this whenever you run the program.
4. Import the test program phase3\_test from the G:/Information/Phase 3 folder. Rename the project to standardname3\_test. Rename the phase3\_test/src/phase3\_test.s source file to standardname3\_test.s.
5. Change the SDK in the standardname3\_test software project to standardname2.ia. This SDK was created in Phase 2 and should still be available, but if it isn’t simply build the Assembler in your standardname2 hardware project.
6. Build the standardname3\_test project. At this point there should be a working environment for enhancing the test program.
7. **Modify the Test Program**

The test standardname3\_test.s will be modified to exercise a set of possible errors. Each possible error is associated with one or more bits in the result register, which must be x1. The test program should exercise each possible error, and set the corresponding bit(s) in x1 if an error is detected. At the end of the test execution, the value in x1 should match the expected value which is defined in line 17 of the isa.codal file.

* 1. **Result Register (x1) Bit Definitions**

The bits of x1 are defined as follows:

1. Bit 0 – 1 => error in the XOR instruction
2. Bit 1 – 1 => error in the ANDI instruction
3. Bit 2 – 1 => error in the ORI instruction
4. Bit 3 – 1 => error in the SLT instruction
5. Bit 4 – 1 => error in the SLTU instruction
6. Bit 5 – 1 => error in the AUIPC instruction
7. Bit 6 – 1 => error in the BLT instruction. The instruction will branch to the correct address but the branch decision is incorrect.
8. Bit 7 – 1 => error in the BLTU instruction. The instruction will branch to the correct address but the branch decision is incorrect.
9. Bits 13:8 – a 1 in bit 13 indicates an error in the ADD instruction, which will be a single output bit stuck at either 0 or 1. If bit 13 indicates an error, bits 12:8 hold the number of the bit which fails. For example, if bit 5 is stuck (at either 0 or 1) bits 12:8 will have the value 00101.
10. Bit 14 – 1 => error in the SRA instruction
11. Bit 15 – 1 => error in the SRL instruction
12. Bit 16 – 1 => error in the LW instruction
13. Bit 17 – 1 => error in the SW instruction
14. Bit 18 – 1 => error in the SUB instruction
    1. **Known Good Instructions**

The following instructions are guaranteed to operate correctly:

AND, OR, SLL, ADDI, XORI, SLLI, BEQ, BNE, JAL, JALR, SB, LB, LBU

Other instructions may have errors which do not need to be reported.

1. **Testing the Project**

After a few instructions have been written, the test program can be built with CTRL-B and run with Run -> Debug Configurations. Step through the instructions and observe the changes in the Register File registers. Once some errors are correctly detected, add more instructions and repeat until register x1 is correct.

It may be valuable to develop the x1 register bits in the order they are specified (bit 0 -> bit 18) although that is not required.

1. **Test Development Ideas**

The key to developing good Verification tests is to consider all of the possible errors which can occur, and to create tests for each case. This should result in several tests setting the same bit of x1. The sections below provide some ideas for various tests.

* 1. **Getting Started**

The basic concept of almost all verification tests is to exercise each instruction with enough different input values to detect all errors and log the errors (in x1 in our case). There are two basic approaches:

* + 1. **Check all Cases**

This case is shown for each instruction in the outline below. “Test” means set up some registers, execute the instruction under test to get a result register. “Check” means compare the result to the expected result – this is “good” if the values match. “Log Error” means set the appropriate bit in the result register x1. There may be several such cases for each instruction to exercise all the cases. The bit in x1 may be set multiple times but that doesn’t matter.

INSTR1:

Test

Check

If good – jump to INSTR\_OK1

Log error

INSTR1\_OK1:

* + 1. **Exit on Error**

This case is shown for each instruction in the outline below. This case branches on a failure, logs the error and then immediately goes to the next test, so that the bit in x1 is set at most once. Either approach is fine.

Test

Check

If bad – jump to INSTR\_FAIL

More tests

Jump to NEXT\_INSTR:

INSTR1\_FAIL:

Log error

NEXT\_INSTR:

* 1. **Useful Constants**

The provided phase3\_test program initializes a few useful values. Register x1 is set to 0, and the error bits should be ORed into this register as they are encountered. Registers x31 (-1) and x30 (1) are initialized to values which are very useful in testing. The test should terminate at the FINISH label so that it halts correctly with the registers available for viewing.

* 1. **Register Instructions**

For the register instructions, consider all of the cases which can occur, execute the instruction under test for each case and report the error when detected.

* 1. **Branch Instructions**

Branch instructions will branch to the correct address, but the branch condition may be interpreted incorrectly (i.e. a branch may be taken when it shouldn’t be or vice versa). Verification of incorrect branch addressing is quite complex and not suitable for this type of exercise.

* 1. **Memory Instructions**

Note that only the word memory operations (LW and SW) may be erroneous, and the byte instructions (SB, LB and LBU) are known to be correct. This should provide an idea of how the instructions should be tested. The error may either be that the address calculation is wrong or that the data read/write is corrupted, so the test should cover both cases.

* 1. **Subroutines**

Note that the JAL and JALR instructions are both known to be good. This allows the creation of subroutines which may be used to simplify the test program. At least one subroutine must be implemented to receive the maximum subjective score as defined in Section 5.

* 1. **Loops**

One necessary function is to find a stuck bit (the output of a function is always either 1 or 0 independent of the inputs). This function must be implemented with a loop and NOT with a large number of constant comparisons.

* 1. **Comments**

Each test should include a comment as to which function is being tested, and additional comments are very useful. This will also contribute to the subjective score.

1. **Scoring the Project**

Phase 3 will be scored in several ways, as described in Section 5.1.

There will be a 1% bonus for each day the project is submitted prior to the Target Date (up to a maximum of 7%), and a 5% deduction for each day the project is submitted after the Target Date.

* 1. **Detailed Scoring**

The score is determined in the following way:

1. Your test is run against your failed design – 10 points for each error correctly identified (max 80).
2. Your test is run against a known good Phase 2 design, and should get 0 errors (x1 = 0). 2 points are deducted for each error found.
3. Your test is run against three designs with many errors, including all the ones not in your failed design. If a test crashes, 4 points are deducted. If a test fails to find the correct errors in one of the designs, the number of bad errors is computed for each test and the maximum of the errors in the three tests is calculated. 2.5 points are deducted for each of the max errors (up to 6).
4. If the stuck bits in ADD are not correctly detected in one of the tests, 2 points are deducted.
5. If your code does not call a subroutine, as required in Section 4.6, 2 points are deducted.
6. If your code does not use a loop properly for the stuck bits, as required in Section 4.7, 4 points are deducted.
7. Small deductions may be made for poor coding technique, such as extraneous instructions or lack of commenting.
8. One point is added for each day before the Target Date the project was submitted.
9. 5 points are deducted for each day after the Target Date, up to 5 days.
10. **Exporting the Project**

Once the test program is running, the test project standardname3\_test should be Exported in the ZIP file standardname3.zip. Do NOT include the hardware project in the zip file, only the test project. There is no “work” directory to delete.