RISC-V Simulator Test Plan Record

Version: 0.03 | 2/12/2022

Editors: Hector Soto
Created By: Hector Soto (sotohec@pdx.edu)
December 2/12/2022

Preface

This document describes tests and their results. The purpose of this document is to track tests and their results until tested code is functional with no/minimal errors. Errors/bugs encountered during tests will be recorded until fixed. If a bug is fixed, it will be removed from this document.

Section 1 – SimMemory

This section goes over the testing of simulated memory in our simulation. It is defined as the class SimMemory with its definition + implementation in files “SimMemory.h”, “SimMemory.cpp”. SimMemory uses 4-byte aligned addressing (2 LSBs = 0). So simulated load/store instructions may have to make 2 calls instead of 1 when requesting 4-byte words or 2-byte half-words.

1.1 – Basic Load/Store Test (test_mm.cpp)

“test_mm.cpp” is a test designed for SimMemory’s ability to read/write to memory consistently if given a large amount of random memory requests with random amounts of contiguous memory being requested.

Runs by default have 65,536 memory request tests. A memory request test looks like the following.

```
Address = rand() % 0xFFFFFFFF; //Addresses from 0 -> 0xFFFFFFFE.
Amount = rand() % 512 + 1; //Will write 1 to 512 words.
RanData = rand() % 0xFFFFFFFF;
ExtraData += Amount - 1;
for(uint32_t j = 0; j < Amount; ++j)
{
        Data = Memory.GetDataPointer(Address + j*4); //j*4 to ge
        *Data = RanData;
        Data = Memory.GetDataPointer(Address + j*4);
        if(*Data == RanData)
        {
                ++Success;
        }
        else
        {
                printf("Error: Mismatch at address 0x%08x | 0x%0
        }
}
```

By the end of a run, around 16,800,000 attempts at reading and writing to a memory location are performed. It is possible for some attempts to be on the same address though as addresses are generated semi-randomly.

Biggest Flaw: Since this test immediately checks a memory location after writing it. There’s no guarantee the data is stored in that location upon the request of data at a different address. The code for SimMemory indicates that is not the case, but this test doesn’t check that.

There was only one bug encountered during testing, and it was from GetDataPointer() not correctly calculating where in the memory hierarchy a page was due to simply not shifting masked addresses. **Test now has a 100% success rate.**

```
hector@hector-VirtualBox:~/Desktop/RISC-V_Simulator/test_mm$ ./executable
Success/Failure: 16839061/16839061 | Success Rate: 100.00%
```

1.2 – Consistent Memory Test (test_cm.cpp)

"test_cm.cpp" is a test designed for SimMemory's advanced load/store functions which can load/store a variable amount of data. More specifically, unlike GetDataPointer(), Store() and Load() can access 1-byte, 2-bytes, or 4-bytes of contiguous memory at any address. GetDataPointer() was limited to accessing 4 bytes on 4-byte aligned addresses.

This test generates one random address and one random 4-byte word. The first byte is stored at the address. The first 2 bytes are stored at the address + 1 (account for the byte written), and the entire 4-byte word is stored at the address + 3 (account for the 3-bytes written). This process then repeats on subsequent addresses from the original. So all memory locations written to should be contiguous. Then all memory locations written to are tested to see if they hold the correct data.

Runs by default have 65,536 memory request tests. A memory request test looks like the following.

```
Address = rand() % 0xFFFFFFFF; //Addresses from 0 -> 0xFFFFFFFE.
RanData = rand() % 0xFFFFFFFF;
for(uint32_t i = 0; i < TEST_COUNT; ++i)
{
        Memory.Store(Address + 7*i, RanData, LS_BYTE); //Store LSByte.
        Memory.Store(Address + 1 + 7*i, RanData, LS_HALF); //Store first
        Memory.Store(Address + 3 + 7*i, RanData, LS_WORD);
        ExtraData += 2;
}
for(uint32_t i = 0; i < TEST_COUNT; ++i)
{
        if(Memory.Load(Address + 7*i, LS_BYTE) == (RanData & 0xFF))
        {
                ++Success;
        }
        else
        {
                printf("Address: 0x%08X Data: 0x%08X | BYTE: 0x%08lX\n", A
        }
        if(Memory.Load(Address + 1 + 7*i, LS_HALF) == (RanData & 0xFFFF))
        {
                ++Success;
        }
        else
        {
                printf("Address: 0x%08X Data: 0x%08X | HALF: 0x%08lX\n", A

        }
        if(Memory.Load(Address + 3 + 7*i, LS_WORD) == RanData)
        {
                ++Success;
        }
        else
        {
                printf("Address: 0x%08X Data: 0x%08X | WORD: 0x%08lX\n", A

        }
```

Biggest Flaw: Since only one random 4-byte word is generated and checked for, it is possible this test could fail to catch a consistently occurring bug in the Load/Store functions.

There was a minimal amount of bugs during testing.
Test now has a 100% success rate.

```
hector@hector-VirtualBox:~/Desktop/RISC-V_Simulator/test_cm$ ./executable
Success/Failure: 196605/196605 | Success Rate: 100.00%
```

Section 2 – RVSimulator

This section goes over the testing of the overall simulator. Only major tests that checked the functionality of instructions were recorded. Minor tests for things such as reading files in correctly, terminal output, command line processing, cross-platform compilation, terminal/file output, etc, were done incrementally at the time of coding.

There was no major direct instruction tests for load/store instructions as they are just wrappers for calls to SimMemory's Load()/Store() functions which were already tested.

2.1 – Immediate Instructions (Immediate.s/mem)

The following test was written in RISC-V assembly (.s) and converted into a memory file using the RISC-V assembler and manually fixing any inconsistencies from assembler optimizations. This test is performed by loading the "Immediate.mem" file into the simulator and running it. This file tests all immediate functions (ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI). Whenever a test for an instruction is passed, a value of 1 is stored in memory, when a test fails, a value of 0 is stored in memory.

Here is the first portion of the test file.

```
# This test assembly will not follow normal conventions.
# This will test ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI
# If a test passes, 1 will be stored at the current memory location.
# Otherwise a 0 will be stored.
# ADDI's result will be stored at 0x0. SLTI's at 0x4, and so on.
li      sp,0            # Start at memory location 0x0.
li      t0,5            # ---| ADDI TEST |---
li      t1,8            # Expected ADDI result.
li      t3,0            # Set test result to 0 (false).
addi    t0,t0,3         # ADDI 5 + 3. Result should be 8.
bne     t0,t1,8         # If ADDI 5 + 3 != 8, ADDI failed. Jump past success.
li      t3,1            # Set test result to 1 (true).
sw      t3,0(sp)        # Store test result at current memory location.
addi    sp,sp,4         # Increment to next memory location to store next test result.
li      t0,-83          # ---| SLTI TEST |---
slti    t0,t0,-47       # t0 = 1 if -83 < -47.
sw      t0,0(sp)        # SLTI gives test result. No branch code needed.
addi    sp,sp,4         # Increment memory location.
li      t0,457          # ---| SLTIU TEST |---
sltiu   t0,t0,-1        # (unsigned)-1 should be > 457.
sw      t0,0(sp)
addi    sp,sp,4         # Increment memory location.
li      t1,0            # ---| XORI TEST |---
li      t3,0            # Set test result to 0.
li      t0,-1           # Load 0xFFFFFFFF.
xori    t0,t0,-1        # Result should be 0.
bne     t0,t1,8
li      t3,1            # Set test result to 1.
sw      t3,0(sp)        # Store test result
addi    sp,sp,4         # Increment memory location.
li      t0,0            # ---| ORI TEST |---
```

Biggest Flaw: There's no randomization for this test, so it may not catch potential edge cases that would cause bugs.

This test now has a 100% success rate.

2.2 – Register Instructions (Register.s/mem)

This test was created and designed nearly identically to the previous test. It tests the register instructions (ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND).

Here is the first portion of the test file.

```
li      sp,0            # Start a memory location 0x0.
li      t0,531          # ---| ADD TEST |---
li      t1,364
li      t2,895
li      t3,0            # Set test result to 0 (false).
add     t0,t0,t1        # t0 = 531 + 364 = 895
bne     t0,t2,8         # Branch if result is incorrect.
li      t3,1            # Set test result to 1 (true).
sw      t3,0(sp)        # Store test result at current memory location.
addi    sp,sp,4         # Increment memory location.
li      t0,279          # ---| SUB TEST |---
li      t1,128
li      t2,151
li      t3,0
sub     t0,t0,t1        # t0 = 279 - 128 = 151
bne     t0,t2,8
li      t3,1
sw      t3,0(sp)
addi    sp,sp,4         # Increment memory location.
li      t0,7            # ---| SLL TEST |---
li      t1,5
li      t2,224          # t2 = t1 << 5
li      t3,0
sll     t0,t0,t1        # t0 = 7 << 5
bne     t0,t2,8
li      t3,1
sw      t3,0(sp)
addi    sp,sp,4         # Increment memory location.
li      t0,-47          # ---| SLT TEST |---
```

Biggest Flaw: There's no randomization for this test, so it may not catch potential edge cases that would cause bugs.

This test now has a 100% success rate.

2.3 – JALR Instruction (JALR.c/s/mem)

This test was created in C, converted to RISC-V, and then converted into a memory file. The test consists of three function calls buried within each other. E.g. main() would call First(), First() called Second(), and Second() called Third(). A success was considered to be the simulation successfully jumping in the expected order and ending correctly.

Here is the test file.

```
void Third()
{
    return;
}

void Second()
{
    Third();
    return;
}


void First()
{
    Second();
    return;
}


int main()
{
    First();
    return 0;
}
```

Biggest Flaw: There's no randomization for this test, so it may not catch potential edge cases that would cause bugs.

This test now has a 100% success rate.

2.4 – The GUI, Time-Stepping, & Time Reversal

The GUI was made for extra credit and was tested incrementally during its creation. There were no specific major tests for the GUI itself, it was considered working if all UI elements showed the same information that was accessible via terminal simulation or trace files.

There was no specific major test for time-stepping. Time-stepping was implemented by moving the code that processes instructions outside of the simulation loop into its own function that would be called by the GUI or terminal simulation. Thus both the GUI and terminal are guaranteed to have the same simulation results.

Time reversal was implemented for easier testing and potential extra credit. There was no specific major tests for time reversal. If the same results were achieved as a normal test when time-reversal was used, then it was successful. Time reversal as of now appears to be completely accurate code-wise and in actual usage.