Contents

Getting started

I hope you've had a chance to look at the specification and resources for Stage 1 of the RV64SIM Assignment and are thinking about starting it soon. As I mentioned in the specification document, I have provided skeleton files in the rv64sim.tgz and rv64sim.tar archives accessible from the RV64SIM Assignment Specification and Resources page. You should download one of these, unpack it, and use the files as the initial versions in your SVN repository for the assignment.

I'd suggest that you start with the memory class, since much of the simulator relies on it. You can then get the m and l commands working. Next you could implement the processor registers, allowing you to get the x and pc commands working. After that, you'll be ready to dive into execution of instructions. That will involve fetching instructions from memory, decoding them, and performing the required operations for each type of instruction.

To help you check your work as you incrementally develop your simulator, I have provided a suite of tests. They're also accessible via the RVSIM Assignment Specification and Resources page. You should create a directory call *tests* in the same parent directory as your *rv64sim* directory. Then unpack the archives in this directory. You should end up with the following directory hierarchy:

- parent directory
  - rv64sim
  - tests
    - command_tests
    - harness_tests
    - instruction_tests
    - compiled_tests

The test suite includes scripts for running tests that assume this directory hierarchy.

You can start with the test cases in the command_tests directory. Each test case exercises one of the commands. You can run each test case by invoking your simulator with the .cmd file as redirected standard input on the command line, for example:

```
../../rv64sim/rv64sim < command_test_m.cmd
```

I've provided log files in the *expected* directory containing output from my simulator for each test case. You can use them as a guide for the expected results from your simulator. If you want to compare your output with the expected output, you can pipe you simulator output to diff, for example:

```
../../rv64sim/rv64sim < command_test_m.cmd | diff - expected/command_test_m.log
```

The log files with "-v" in the name are from runs with the -v option on the command line. This causes my simulator to output additional "verbose" information useful for debugging. I'd strongly recommend you do something similar in your simulator. (You don't have to produce exactly the same debug output as my simulator; you can output more or less information, depending on what you need to debug.)

The next group of test cases is in the *harness_tests* directory. These are basic tests for execution of six instructions: addi, addiw, bne, jalr, lui, and slli. If you look at the .cmd files, you'll see that they work by setting a memory location to contain the binary-coded word for an instruction, then executing the instruction and displaying the resulting register and/or pc value. Again, I've provided log files from my simulator, including from runs with the -v option.

The reason for choosing these six instructions for basic tests is that I use them for a more comprehensive test harness for a complete test suite for all instructions. That test suite is in the *instruction_tests* directory. As you will see, the test cases are in assembly language, translated to hex files for loading into the simulator. I'll describe that test suite and how to use it in more detail later.

So now to the obligatory harangue: Get started early, pace yourself, and don't leave it all to the last minute. There's a reasonable amount of work involved, but once you get into it you should find that it's not a very complex program to develop. Much of the effort will be in the testing and debugging process. Please allow time for that.

I hope you find this assignment interesting and a good way to learn about real-world instruction set architectures in more detail.

The memory class

The memory class in rv64sim models the behaviour of the main memory attached to the RISC-V processor. In order to impement the memory class, you'll need to decide on a data structure to represent the memory storage. Conceptually, memory is just an array of bytes, indexed using byte addresses. The methods in the class interface read and write doublewords, rather than bytes, since that more directly models how real memory hardware works. Since most processor accesses to memory are for a doubleword at a time, a hardware system achieves better performance by optimizing doubleword access. So the refined conceptual model of memory is an array of doublewords, indexed using byte addresses that are rounded down to the nearest multiple of 8.

The memory class methods are listed and described as follows:

- `memory(bool verbose);`

This is the constructor, called by the rv64sim main program to create the simulated memory. The verbose argument is set if the -v option is included on the command line. Your memory methods can display useful debugging information if verbose is true. If verbose is false, memory methods must only display output expected by the test cases.

- `uint64_t read_doubleword (uint64_t address);`

The read_doubleword method reads a doubleword of data (8 bytes) starting at the location given by the address argument. The address should be doubleword aligned, meaning that it should be a multiple of 8. For robustness, your implementation should round the address down to the nearest multiple of 8, just in case address is not properly aligned.

- `void write_doubleword (uint64_t address, uint64_t data, uint64_t mask);`

The write_doubleword method updates the doubleword of data in memory starting at the location given by the address argument. The new data is provided in the data argument. The mask argument is used to specify which bytes in the doubleword of memory are to be updated. The mask will contain 1 bits corresponding to those bytes that are to be written, and 0 bits elsewhere. For example, if only the least significant byte is to be written, the mask will contain 1s in bits 0 to 7, and 0s in bits 8 to 63. As for the read_doubleword method, the address argument should be doubleword aligned, but you should allow for the possibility that it's not by rounding down.

- `void show_address (uint64_t address);`

The show_address method is used for the m command to display the doubleword of data starting at the location given by the address argument (which should be aligned). This is a separate method from read_address, as it needs to display the data rather than returning it, and it needs to not perform any debug output that read_address would do.

- `void set_address (uint64_t address, uint64_t data);`

The set_address method is used for the m command to update the doubleword of data in memory starting at the location given by the address argument. The new data is provided in the data argument. There is no mask argument, so the whole doubleword location is updated.

- `bool load_file(string file_name, uint64_t &start_address);`

The load_file method is used by the l command to read contents from a hex image file into memory. The file is identified by the file_name argument. A hex image file is produced by the compiler and linker and contains the binary-coded instruction words for a program. The memory address at which the program should start executing is included in the hex image file and is returned in the start_address argument of the method. The return value of the method is true if the load operation completed successfully, or false otherwise. The skeleton code for the method does all the work of interpreting the hex image file. The only part you need to complete is the part that updates memory with each loaded byte.

The assignment specification mentions that performance of the simulator is important, and advises against using a large array to implement storage. Since addresses are 64 bits long, the number of bytes in the conceptual memory array is 2^{64}, or 2^{61} doublewords. If you try to use an actual array of

bytes or doublewords, the host computer (your computer on which you run your rv64sim program) would need to allocate 2^{64} bytes of its memory for the array. Computers simply don't have that much memory.

The key to improving this aspect of performance is to recognize that a simulated RISC-V program would typically use only a small fraction of the memory address space. The memory class only needs to provide storage for those locations that are actually addressed by the simulated program. We could achieve this by using a data structure that allocates a doubleword of data on the first read_doubleword or write_doubleword access to a given address. We would need to be able to look up that doubleword using the address as a key for subsequent accesses to the address. Like array access, the lookup operation should be constant-time complexity order.

While this approach may be functionally correct, it too suffers performance problems. Allocation of data in a program using new() requires run-time library and operating system support, so is relatively time consuming. If the memory class calls new() for each doubleword that is accessed, that will impose a significant performance overhead.

The compromise suggested in the assignment specification is to use a data structure that allocates a block of doublewords on the first read_doubleword or write_doubleword access to any address within the block. A block is just a fixed-size contiguous group of addresses. It's usually convenient to make the block size a power of 2. As an example, suppose you chose the block size to be 2^{10} = 1024. Then addresses 0 to 1023 would be in block 0, addresses 1024 to 2047 would be in block 1, 2048 to 3071 in block 2, etc. If a write at address 1032 is the first access to block 1, the memory class would allocate storage for block 1 and write to the doubleword at offset 8 in that block. Subsequently, a read at address 1048 would find that storage was already allocated for the block and would return the value of the doubleword at offset 24 in the block.

The processor class

The processor class in rv64sim models the behaviour of the RISC-V processor. The processor class methods are listed and described as follows:

- `processor(memory* main_memory, bool verbose);`

This is the constructor, called by the rv64sim main program to create the simulated processor. The main_memory argument refers to the memory that is to be used for load and store instructions. The verbose argument is set if the -v option is included on the command line. Your processor methods can display useful debugging information if verbose is true. If verbose is false, processor methods must only display output expected by the test cases.

- `void show_pc();`

The show_pc method is used for the pc command to display the value of the program counter.

- `void set_pc(uint64_t new_pc);`

The set_pc command is ised for the pc command to update the value of the program counter. The new value is provided in the new_pc argument.

- `void show_reg(unsigned int reg_num);`

The show_reg method is used for the x command to display the value of a processor register. The register number is provided in the reg_num argument.

- `void set_reg(unsigned int reg_num, uint64_t new_value);`

The set_reg method is used for the x command to update the value of a processor register. The register number is provided in the reg_num argument, and the new value is provided in the new_value argument.

- `void execute(unsigned int num, bool breakpoint_check);`

The execute method is used by the . command to simulate execution of instructions in the processor. The number of instructions to execute is specified by the num argument. If the breakpoint_check argument is true, the method checks whether there is a breakpoint set at the address given by the program counter before executing that instruction. If so, execution stops without executing that instruction. If the breakpoint_check argument is false, execution continues for the specified number of instructions regardless of any breakpoints.

- `void clear_breakpoint();`

The clear_breakpoint method removes any previously set brekpoint.

- `void set_breakpoint(uint64_t address);`

The set_breakpoint method sets a breakpoint at the address specified by the argument. Any previously set brekpoint is removed.

- `uint64_t get_instruction_count();`

The get_instruction_count method returns the number of intructions executed since the start of simulation.

- `uint64_t get_cycle_count();`

The get_cycle_count method returns the number of clock cycles that would be required for execution of instructions since the start of simulation.

Operations on bit fields

In the assignment, we can represent instruction words using the C/C++ type uint32_t and data doublewords using the C/C++ type uint64_t. Value of these types are represented in the host computer (the computer on which you run your simulator) in binary as 32-bit or 64-bit unsigned integers. Since the host computer can perform operations on these types very efficiently, we get very efficient simulation of operations in the RISC-V processor and memory.

One place where we need to operate on uint64_t values is in the write_doubleword method of the memory class. That method has a mask argument that contains 1 bits for bytes to be updated in the memory doubleword. For example, to write just the least-significant byte of a doubleword, we would provide the mask value (shown in binary and hex):

```
0000000000000000000000000000000000000000000000000000000011111111    0x0000000000000FF
```

The write operation needs to access the memory doubleword at the specified address and just change the bits corresponding to the 1s in the mask. Those bits should be changed to the corresponding bits from the data argument, and the other bit should remain unchanged. The modified doubleword is then written back to the memory data structure. For example:

```
old_data        0010000000000001101000000000000011000011010001000001111101111010
    0x2001A000C3441F7A
data            0001000011111111000000000000000011001100000000111100001000100088
    0x10FF00003300F088
mask            0000000000000000000000000000000000000000000000000000000011111111
    0x00000000000000FF
                ----------------------------------------------------------------
    ------------------
updated word    0010000000000001101000000000000011000011010001000001111110001000
0    0x2001A000C3441F88
```

Note that in this case, the data doubleword contains values in the most-sigificant bytes that we need to ignore.

We can combine the values in a couple of steps using C/C++ bitwise logical operators. First, we can select the bits of the data doubleword corresponding to the 1 bits in the mask by using the & (bitwise AND) operator:

```
data          0001000011111111000000000000000011001100000000111100001000100088
   0x10FF00003300F088
mask          0000000000000000000000000000000000000000000000000000000011111111
   0x00000000000000FF
              ----------------------------------------------------------------
   ------------------
data & mask    0000000000000000000000000000000000000000000000000000000010001000
    0x0000000000000088
```

Second, we can select the bits of the orignal memory word that are to be retained, corresponsing to the 0 bits of the mask, by complementing the mask value with the ~ (bitwise NOT) operator, then applying the & operator:

```
old_data           0010000000000001101000000000000011000011010001000001111101111010
1010    0x2001A000C3441F7A
~mask              1111111111111111111111111111111111111111111111111111111100000000
0000    0xFFFFFFFFFFFFFF00
                  ----------------------------------------------------------------
```

```
---    ------------------
old_data & ~mask    0010000000000001101000000000000011000011010001000001111100000000    0x2001A000C3441F00
```

Finally, we combine the two results with the | (bitwise OR) operator:

```
data & mask          0000000000000000000000000000000000000000000000000000001000001000    0x0000000000000088
old_data & ~mask     0010000000000001101000000000000011000011010001000001111100000000    0x2001A000C3441F00
                     ----------------------------------------------------------------    ------------------
(data & mask)        0010000000000001101000000000000011000011010001000001111110001000    0x2001A000C3441F88
| (old_data & ~mask)
```

This is the final value that we then write back to the memory data structure at the specified address.

We can use the same operations to update other parts of a word. Here are some examples of mask values that would be used with the write_douleword method:

- most-signficant byte:

```
1111111100000000000000000000000000000000000000000000000000000000    (0xFF00000000000000)
```

- byte at offset 2:

```
0000000000000000000000000000000000000000111111110000000000000000    (0x0000000000FF0000)
```

- least-significant halfword:

```
0000000000000000000000000000000000000000000000001111111111111111    (0x0000000000FFFF)
```

- whole word:

```
1111111111111111111111111111111111111111111111111111111111111111    (0xFFFFFFFFFFFFFFFF)
```

An important point to note is that the &, |, and ~ bitwise operators are different from the &&, ||, and ! logical operators in C/C++. As illustrated above, the bitwise operators work on corresponding bits of the operands to produce each bit of the result. The logical operators, in the other hand, treat the operands as Boolean values and produce a Boolean result. In C, any non-zero value is

considered to be a true Boolean value, and 0 is the false Boolean value. The logical operators produce either 1 or 0 as the result. Confusing the bitwise and logical operators is a common pitfall in C programming, one that can be hard to detect, since the compiler cannot identify the error. In C++, there is a distinct type, bool, for Boolean values, which helps to avoid such errors.

Another place where bitwise operations need to be used in the RV64SIM simulator is in decoding RISC-V instructions in the processor class. Each RISC-V instruction is encoded in a word of type uint32_t. The encodings for the RV64I subset are summarized in Chapter 19 of the RISC-V User-Level ISA V2.2 document. To decode a RISC-V instruction, we need to extract the opcode fields and use the values to decide what operations to simulate. Similarly, we need to extract the register number fields and use them to index an array of register values.

As an example, consider the RISC-V instruction ld x7, -12(x24). This I-format instruction has opcode 0000011 (0x03), funct3 field 011, register number rd is 7, register number rs1 is 24, and immediate field is -12. So the binary instruction word is

```
11111111010011000011001110000011    0xFF4C3383
```

Suppose this instruction word is assigned to a variable instr in our C++ program. To decode the instruction, we first need to extract the opcode field (the rightmost 7 bits). What we want to end up with is an unsigned integer value with the opcode value in the least significant 7 bits. We can use the & operator discussed previously to do this:

```
instr & 0x0000007F
```

The mask value in binary is

```
00000000000000000000000001111111
```

which has 1 bits in positions corresponding to the opcode. So the result of the expression is just 0x00000003, as required.

Next, we need to extract the funct3 field (bits 14:12 of the word in instr), ending up with an unsigned integer value with the funct3 value in the least significant 3 bits. We can pick out just the funct3 field using the & operator, as before:

```
instr                 11111111010011000011001110000011
0x00007000             00000000000000000111000000000000
                      --------------------------------
instr & 0x00007000     00000000000000000011000000000000
```

The problem is that the funct3 value is still in bits 14:12 of the result. We can correct that by shifting using the C/C++ >> operator, which does a right shift of bits by a given number of placed. In this case, we need to shift right by 12 places:

```
(instr & 0x00007000) >> 12    00000000000000000000000000000011
```

Now that we have extracted the opcode and funct3 values, we check them and determine that the instruction is a ld, so we need to extract the register number and immediate value. We extract the register numbers in the same way as the funct3 field:

```
rd = (instr & 0x00000F80) >> 7

rs1 = (instr & 0x000F8000) >> 15
```

That just leaves the immediate field. We could try the mask and shift approach, as follows:

```
imm = (instr & 0xFFF00000) >> 20
```

However, the RISC-V ISA specification states that the immediate value is 2s-complement signed. If we just apply the shift operator as show above, we would get an incorrect value for negative immediates. For example, the value -12 in the lw instruction is represented as 111111110100 in bits 31:20 of instr. Applying the above operators would give the result

```
00000000000000000000111111110100
```

which is the value 4084 in decimal. We need to treat the 12-bit immediate value as signed, and produce a 32-bit representation of the same value. In other words, we need to sign extend the 12-bit signed value to a 32-bit signed value. Fortunately, the RISC-V ISA designers have helped by ensuring that the sign bit of the 12-bit immediate value is in the leftmost position of the 32-bit instruction word. That allows us to get the 32-bit immediate value by performing an arithmetic right shift by 20 places. That operation also conveniently removes the other fields from the value.

In order to do the arithmetic right shift, we need to use the C/C++ >> operator, but with a signed operand. We can cast the instr value to the signed type int32_t with the expression

```
(int32_t) instr
```

This simply tells the compiler to treat the value of instr as being of type int32_t without modifying the bits in the value. Putting this together with the arithmetic shift operation gives:

```
instr                    11111111010011000011001110000011    0xFF4C3383
(int32_t) instr           11111111010011000011001110000011    0xFF4C3383 (same b
its)
((int32_t) instr) >> 20    11111111111111111111111111110100    0xFFFFFFF4
```

which is the 32-bit signed binary representation of -12 decimal.

Extracting the 12-bit immediate value for an I-format instruction is the simplest case, since all of the imm field bit are adjacent and in order in the instruction word. For other instruction formats with immediate operands, the bits of the immediate field are distributed differently. That means we need to extract each of the parts of the immediate field and combine them to produce a 32-bit signed immediate value. We can do that using the various logical and shift operators discussed in this section. We leave that as an exercise for the reader.

Endianness in memory accesses

The RISC-V ISA specification states that RISC-V is little endian. If you haven't come across that term before, I'll explain it here.

The addresses used to access memory are byte addresses, each identifying a location for storing a byte of data. The RV64I ld and sd instructions, however, read and write whole doublewords of data, where a doubleword comprises eight bytes. The issue is how the eight bytes of a doubleword are arranged in eight locations in memory.

The little-endian approach is to put the least significant byte at the least address, the second most significant byte in the next highest address, and so on, up to the most significant byte in the highest address. The following ascii-art shows an example of how the doubleword 0x3F3E3D3C3B3A3938 would be stored starting at address 0x00400010, showing each byte location separately:

```
   Address    |       |
              +-------+
   0x00400010 |   38  |
              +-------+
   0x00400011 |   39  |
              +-------+
   0x00400012 |   3A  |
              +-------+
   0x00400013 |   3B  |
              +-------+
   0x00400014 |   3C  |
              +-------+
   0x00400015 |   3D  |
              +-------+
   0x00400016 |   3E  |
              +-------+
   0x00400017 |   3F  |
              +-------+
              |       |
```

Contrast this with the big-endian approach, where the bytes of the doubleword are in the reverse order:

```
   Address    |       |
              +-------+
   0x00400010 |   3F  |
              +-------+
   0x00400011 |   3E  |
              +-------+
   0x00400012 |   3D  |
              +-------+
   0x00400013 |   3C  |
              +-------+
```

```
    0x00400014 |   3B   |
               +--------+
    0x00400015 |   3A   |
               +--------+
    0x00400016 |   39   |
               +--------+
    0x00400017 |   38   |
               +--------+
               |        |
```

So the endian-ness indicates which end of the doubleword appears at the least address - the little end (least significant, or the big end (most significant).

Does endian-ness matter? Provided ld and sd both read and write bytes in the same order, no, it doesn't. However, if a doubleword is written using a sd instruction, then the bytes are read separately using lb instructions, a difference in endian-ness will be apparent. Given that RISC-V is little endian, if we did a sw to write 0x3F3E3D3C3B3A3938 to address 0x00400010 as above, then did a lb to read from 0x00400010, we should get 0x38 as the result. If we had stored in big-endian format, we would get 0x3F as the result of the lb, which would be incorrect.

One final point on endian-ness is that it applies not just to doublewords, but also to words (four bytes) and halfwords (2 bytes). In each of these cases, little-endian arrangement stores the least-significant byte of the quantity at the least address and the most-significant byte at the highest address; and bit-endian arrangement stores the most-significant byte of the quantity at the least address and the least-significant byte at the highest address.

Running the instruction tests

Once you've got the basic command and harness tests running, you can move onto the instruction tests. These tests are in the instruction_tests folder of the test suite provided in the instruction_tests.gz and instruction_tests.zip archives.

You'll see that there is a test case for each instruction type in the instruction set. There is an assembly language file called instruction_test_*.s, that contains code to exercise the instruction. You'll see calls to macros TEST_INIT, TEST_EXPECT, TEST_FAIL and TEST_END. These are all defined in test_defines.h. The macros use register a0 to record the outcome of the test (0 for fail, 1 for pass) and register a1 as a temporary. TEST_EXPECT compares a register with an expected value and terminates the test with failure if the register differs from expected. TEST_FAIL terminates with failure immediately. TEST_END terminates with success.

I've built all of the test cases for you and generated hex files (instruction_test_*.hex) that you can load in your simulator. The instruction_test_*.dump files are disassembled listings, which show the address offsets of instructions in the text segment. I found this helpful when stepping through execution during simulation.

The Makefile contains commands for building using the GNU RISC-V assembler and linker. I've provided the Makefile in case you're interested in installing the GNU RISC-V tools and adding further test cases.

The run_test shell script allows you to run the simulator with a test case and display the test result, eg:

```
./run_test instruction_test_add
```

The script assumes your rv64sim directory containing your simulator executable is two levels up from the test directory, so you may need to edit it if that's not the case in your directory structure.

The run_test script allows you to pass command-line options to the simulator by setting the environment variable RV64SIM_FLAGS to a string containing the options. You can use this feature to pass the -v option to your simulator to enable verbose debugging output, eg:

```
export RV64SIM_FLAGS='-v'
./run_test instruction_test_add
```

Similarly, if you're doing the postgraduate version of the assignment and need to pass options to the simulator to enable cycle counting, eg:

```
export RV64SIM_FLAGS='-c'
./run_test instruction_test_add
```

There is a second shell script, run_instruction_tests, that runs all of the instruction test cases in sequence, eg:

```
./run_instruction_tests
```

It uses run_test for each test case, and writes output for intruction_test_*x* to instruction_test_*x*.log. If the RV64SIM_FLAGS environment variable is set, the command-line option string is included in the log file name For example, the following

```
export RVSIM_FLAGS='-v -c'
./run_instruction_tests
```

writes output to instruction_test_*x*-v-c.log, for each test case.

I've run all of the tests with my simulator and captured output logs in the 'expected' directory for your reference. For each test case, there are *.log, *-v.log, and *-c.log files. Ultimately, in the websubmit assessment script, I'll run your simulator with each of the test cases and compare output (using diff -iw) with the non-verbose logs. So you should make sure your output matches the logs, with only whitespace and capitalization differences allowed.

Running the compiled code tests

The final set of test cases for Stage 1 is a suite of compiled C-code tests. These tests are in the compile_tests folder of the test suite provided in the compiled_tests.gz and compiled_tests.zip archives. There are six test cases:

- leaf: The simple leaf function from Section 2.8 of the textbook.

- fib: A recursive version of Fibonacci number generator. This is deliberately a very inefficient way to generate Fibonacci numbers, since it involves a huge number of recursive function calls. So it does lots of memory accesses for saving and restoring registers on the stack.
- random: A random number generator using a linear feedback shift register (LFSR) algorithms. This uses bit-wise logical and shift operations.
- sort: A program to sort an array of 1000 random numbers using the bubble sort algorithm from Section 2.13 of the textbook. The program verifies that the numbers in the resulting array are in ascending order.
- quicksort: A program to sort and array of 100000 random numbers using the quicksort algorithm and to verify that the numbers in the resulting array are in ascending order.
- thrash: A program to test performance of your memory implementation. The program uses the random number function to generate addresses for accesses to memory. If you've implemented memory efficiently, this program should run in just a few seconds.

Each test case comprises the C function in a file compiled_test_*x*.c, and an assembly program in a file compiled_test_call_*x*.s. The assembly program sets up the stack pointer, frame pointer, and (if required) global pointer, and then calls the C function. The Makefile contains the commands I used to build the hex file that you can load and run in your simulator, as well as the dump file containing a disassembled listing.

The run_test shell script runs the simulator with a breakpoint at address 0 (indicating termination of the program), and displays the function result in a0. The run_compiled_tests shell script uses run_test to run each of the test cases. Both scripts use the RV64SIM_FLAGS environment variable in the same way as the scripts for instruction tests.