# OSU ECEN 4243 Computer Architecture, Spring 2024

Lab 1: Instruction-Level RISC-V Simulator

Group: Brendan Bovenschen A20323473 and Landon Fox A20319340

February 20th, 2024

## Section 1: Introduction

In this lab, we have written a C program that acts as an instruction-level simulator for the RISC-V 32-bit integer instruction set. To further understand the RISC-V instruction set and low-level programming, we will simulate these processor instructions and decode programs in machine language. While the instruction processes will be different from the function we have written in C due to the difference between these hardware and software implementations, understanding the logic behind these instructions is key to understanding the design of a computer architecture system as well as programming in assembly. This lab gives us insight into what an architecture's instructions look like and how they are shaped by the design of a processor.

## Section 2: Baseline Design

The processor that we are simulating utilizes the 32-bit instructions for the RISC-V instruction set to perform operations and branching logic. The components that make up this system are a series of 32-bit registers (32 total), a unit of system memory, and a processing unit to perform instructions that are read and supplied by our shell.

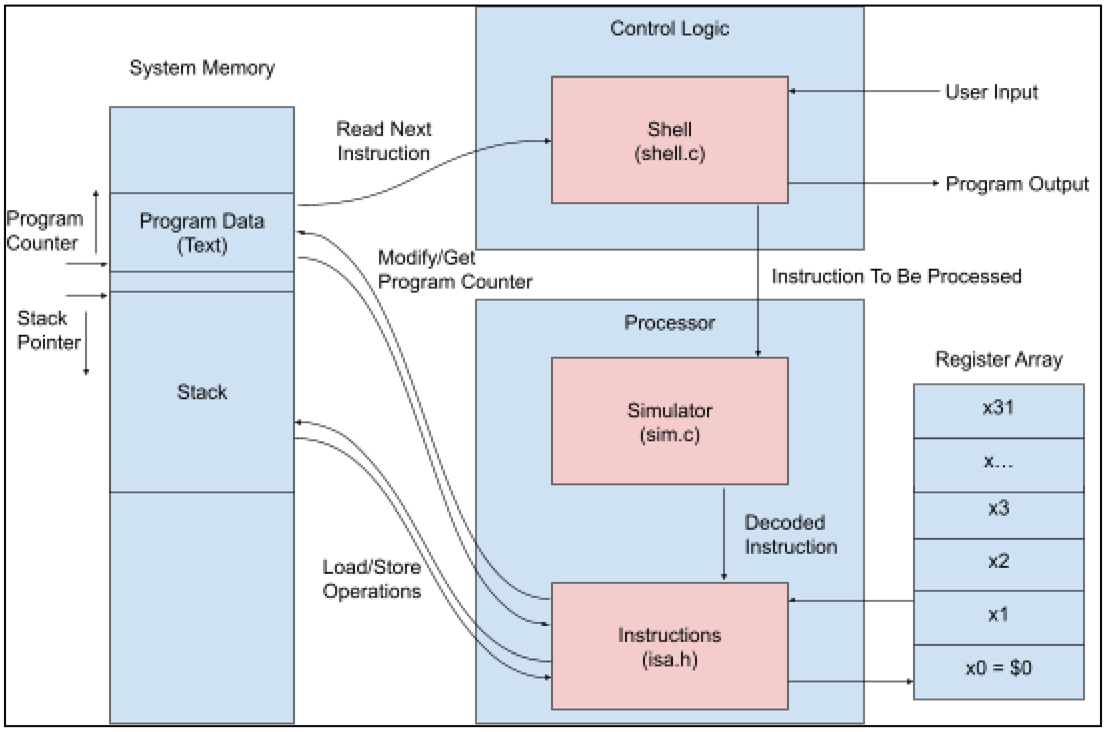


**Figure 1: Diagram of Simulated Processor**

Utilizing a shell script to read each instruction string in our memfile and handle user input, our program can decode these strings and implement their functionality through C operations. Our representation of the RISC-V instruction set in C will interface with our shell to run specified operations when they are provided by the shell. The functionality of our system is split between the simulator (sim.c) which handles the decoding of of a given 32-bit instruction, and an instruction set architecture (isa.h) which will perform the functionality of a given operation called by the simulator. This structure imitates the control and execution of operations in a RISC-V system through the abstraction of C which performs the process of hardware operations. Our simulator implements Von-Neumman architecture in which our system program (text), data, and stack are all within the same memory structure. These are partitioned so as to not overwrite program text in the stack or vice versa.

## Section 3: Detailed Design

Each instruction consists of a function in isa.h and its complementary decoding function in sim.c which allows us to run our machine code test files. These test files contain hexadecimal instructions which are converted into binary and then decoded. First, the processor checks the opcode which will then determine the partitioning of the remaining bits based on the type of instruction which gives us the information needed to run the instructions. In this decoding process, prioritizing the position of the most reused elements is considered while partitioning the decoded string. Throughout each type of encoding, the position of the opcode and each source and destination register is not modified while the immediate is partitioned around these values. Each type of instruction handles different functionalities of the processor, such as modifying the program counter, performing simple operations between registers, and reading or writing to memory.

The R-type instructions consist of simple operations between two source registers that are to be saved in a destination register. This consists of mathematical and logical operations as well as comparative instructions. Many of the instructions utilized a "sign extend" to perform signed operations between numbers with different bit lengths. This is done by repeating the most significant bit of the smaller number (following two's complement rules for negative and positive numbers) until a common size is achieved between both numbers.

For some, the I-type instructions are very similar to the R-types but include an immediate, hardcoded value that is used in the logic of each function. This classification of instructions also contains processes for loading memory from the stack or other parts of system memory. Since RISC-V uses a byte-addressable memory system, but can only read or write a whole word of memory, our address stored in our source register must be split into an aligned address (containing the most significant 30 bits of our effective address), and the 2 least significant bits that represent the byte position in the word we are addressing. This loaded word can then be modified to load the specified byte or half-word to our destination register. The sign extend functionality of our processor is applied to these load instructions and does not apply to the two "load unsigned" instructions (byte and half-word). The effective address supplied to the "load word" instruction is assumed to be aligned and will generate a hardware error otherwise.

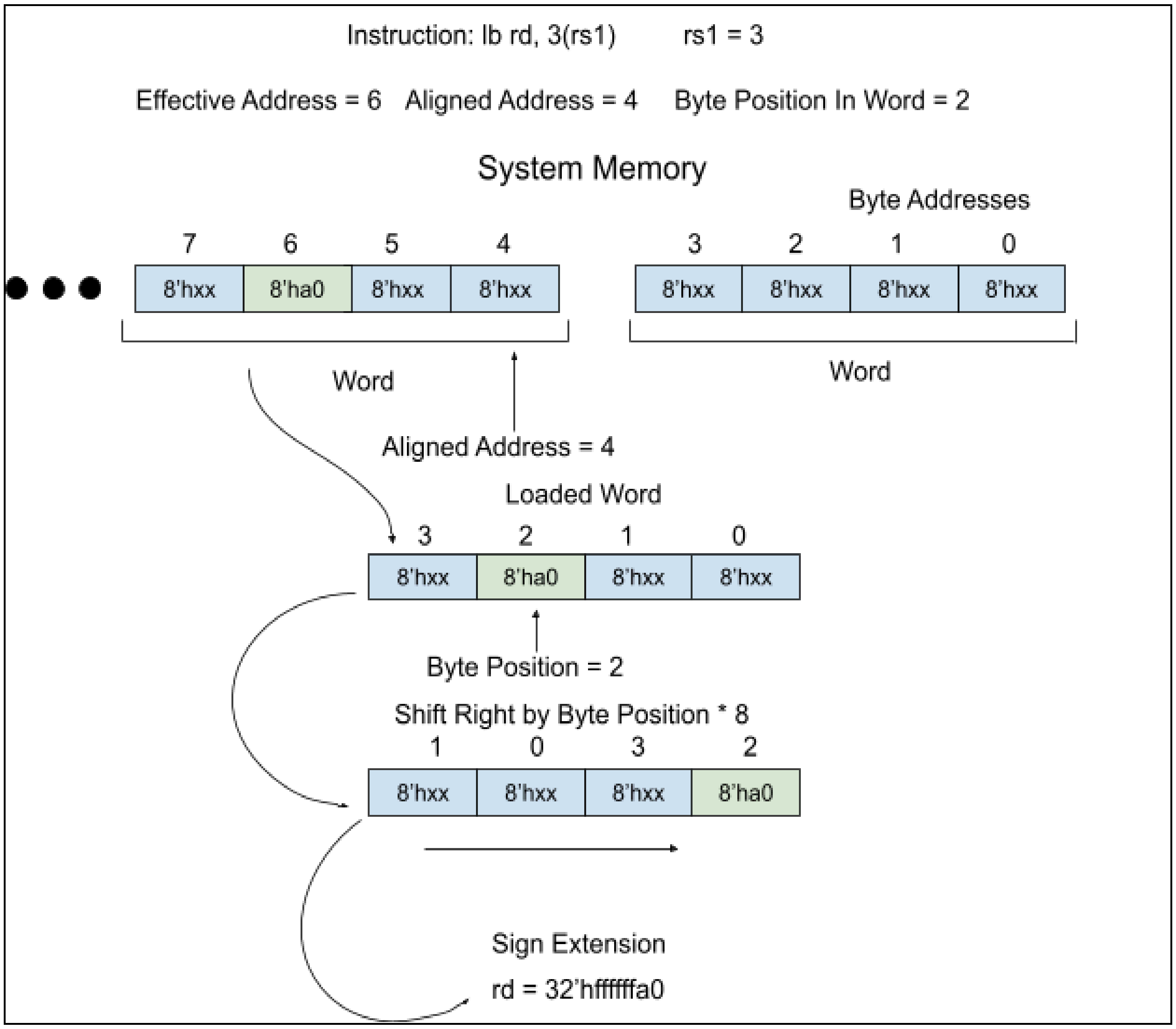


**Figure 2: Example Process for Load Byte Instruction**

Store instructions (S-type) follow a similar process for obtaining the word to be modified by the processor. After loading a word from system memory, the data stored in the source register must be written to the byte position in the word to then be written back to system memory. This can be done in C by first erasing the data at the byte position in the loaded word and then performing an ‘or’ operation between the read word and the data from the source register. Like in the “Load Word” instruction, Store Word assumes the effective address supplied by the first source register and the immediate to be aligned.

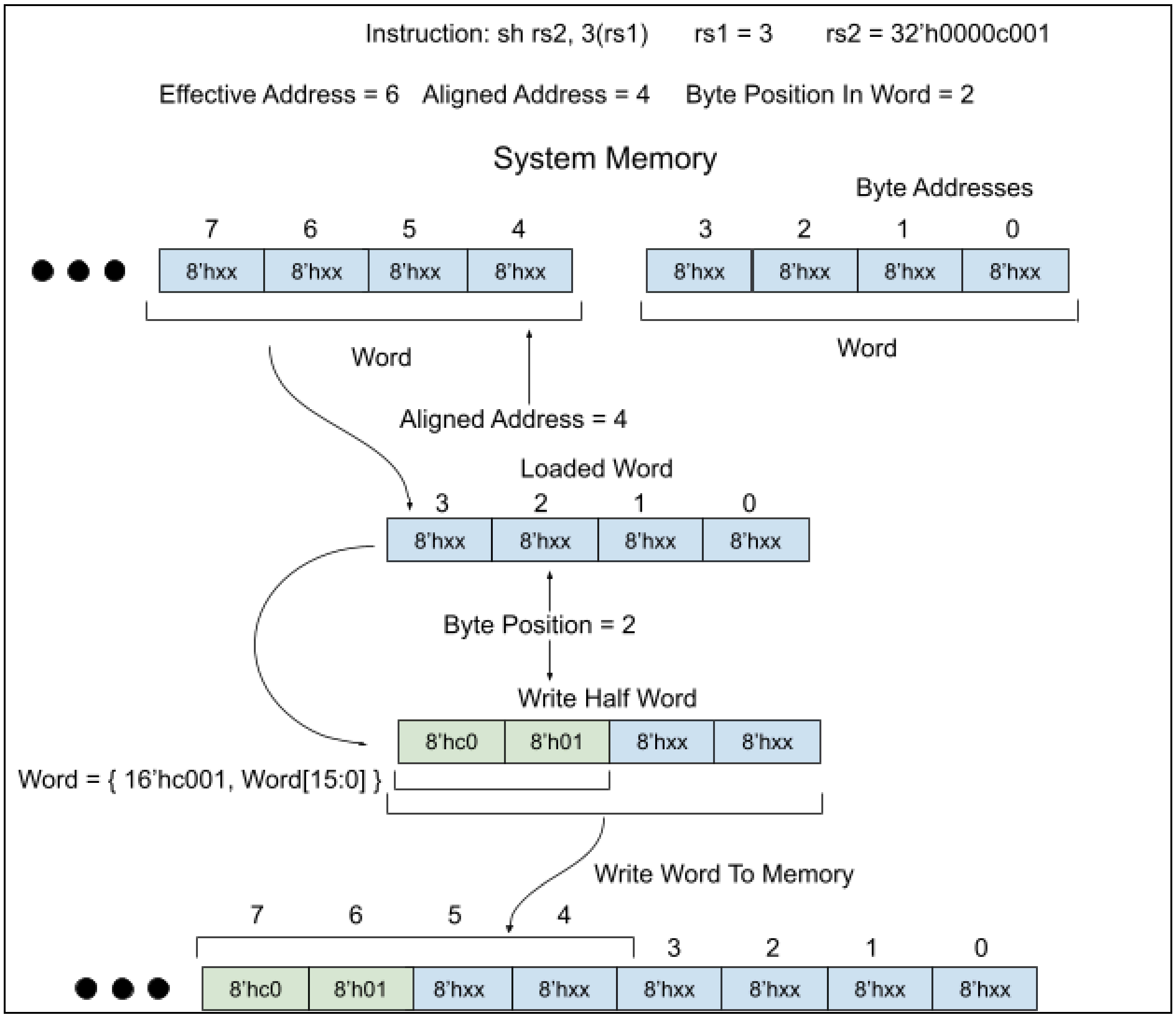
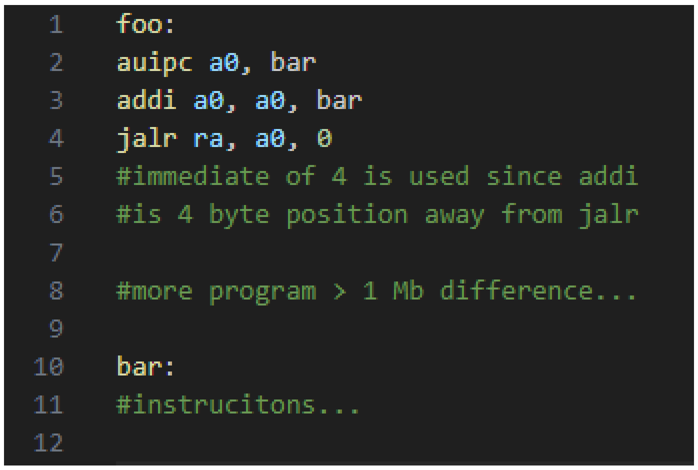


**Figure 3: Example Process for Store Half-Word Instruction**

Branches and jumps in our program allow for more complex assembly programs involving subroutines and the reuse of blocks of code. These types of instructions make up the B-types, J-types, and JALR as the only I-type with jump functionality. Since the program counter stores a byte address of the current instruction in memory, this value can be modified to handle branches and jumps. B-type instructions handle conditional logic that controls the flow of a program based on a comparison of two register values. JAL, the only J-type instruction in the RISC-V 32-bit integer instruction set, will always jump to a position in the program based on the 12-bit immediate label regardless of a condition. This immediate is generated by the assembler to point to the position of a specific label identifier relative to the current instruction (JAL, or any jump utilizing a label). This is crucial for assembly programming as it allows a programmer to implement human-readable routines and jump to this label from a position limited by the 20-bit immediate limit. The immediate value of all B-type instructions and J-types contain a hardcoded '0' in the least significant bit that forces the program to jump, at least, 2 byte positions. These 2 positions are used to compensate for the compressed instruction set which is 2 bytes long, and the 4-byte instructions used by this simulator are unaffected by this limitation. If a label is more than 2^21 byte positions away in memory (or 13-byte position for B-type instructions), an upper immediate must be implemented to jump to an instruction.

This can be handled by our two U-type instructions, AUIPC (add upper immediate to program counter) and LUI (load upper immediate). Both instructions have very similar functionality, but different implementations. LUI will load a 20-bit immediately to the most significant 20 bits of a register. This is helpful for storing and loading addresses in memory to be used across a program. AUIPC can be used to implement a long program jump since JAL and JALR are limited by a 13 and 21-bit immediate respectively (with the last bit being hardcoded to '0'). By adding the immediate value of a label to the program counter using AUIPC, (as seen in Figure 4) and obtaining the least significant bits through an add immediate instruction to the same register, this allows a program to jump a total of 33 byte-addressed positions in the program text.



```
foo:
auipc a0, bar
addi a0, a0, bar
jalr ra, a0, 0
#immediate of 4 is used since addi
#is 4 byte position away from jalr

#more program > 1 Mb difference...

bar:
#instrucitons...

```

**Figure 4: Sample Long Jump in RISC-V Assembly**

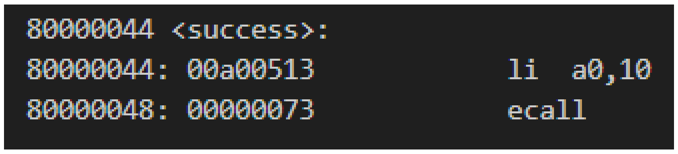
## Section 4: Testing Strategy

To verify the functionality of each instruction in our ISA as well as the instruction decoding in our simulator, we tested each instruction individually using the shell and machine instruction files (.memfile for each instruction) provided for us. This allowed us to incrementally create each instruction and pinpoint which instructions succeeded in their intended operation. Since many of the provided assembled files utilized the U-type instructions, and since those were not provided in isa.h or decoded in sim.c, these were implemented first to begin verifying the rest of the instruction set. The provided disassemble files along with their machine code could be used to pinpoint where our instruction failed when run in the shell. Each test case was marked with an ADDI instruction with the test case number that could be viewed in the shell, and similar markings were implemented for the two "pass" and "fail" branches of the instruction. Each of these test files are self-checking, as in they contained comparisons between hardcoded register values to the register that was modified by the tested instruction. Utilizing these tools, we could confirm that each instruction was performed in the same way as intended by the RISC-V architecture.

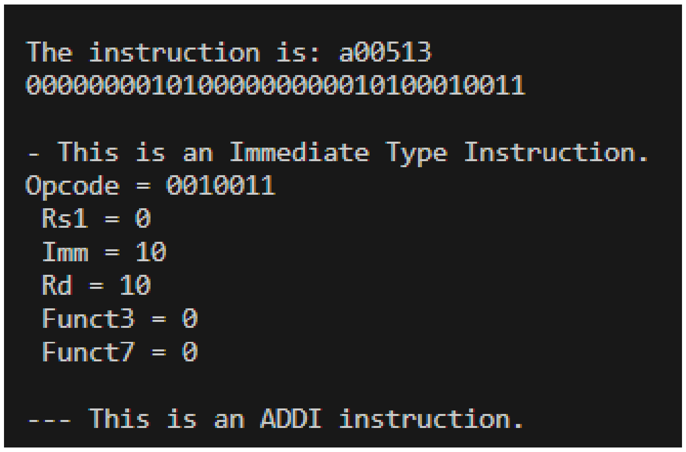
To check every instruction without manually running each of the machine language memfiles, we created a “runall” script to load each object with the “.memfile” extension to the shell. By checking the result on each iteration of this script, the functionality of each instruction could be verified through the implementation of this script.

## Section 5: Evaluation

By using processes similar to hardware implementation in C, we can further grasp the RISC-V instruction set architecture and how a processor using this architecture is developed. Our C implementation of the instructions completes all of their tests without fail and the results match with the registers on all of our test cases. In conclusion, implementing the RISCV instructions in C has given us a higher understanding of how the instructions work for each of the types. R-type instructions are the easiest to implement and understand with them being very simple operations such as adding or anding. U-type instructions were very important for use in other instructions while I-type instructions were important for use in other instructions they present their individual uses as well containing the load instructions as well as simpler ones like addi. S-type instructions are the only instructions to store values in memory and aren’t used as often since accessing/storing to memory takes a long time. B-type and J-type instructions are similar in the fact that they have “goto” functionality either after a condition is met or just to an address without a condition.



```
80000044 <success>:
80000044: 00a00513          li  a0,10
80000048: 00000073          ecall
```



```
The instruction is: a00513
00000000101000000000010100010011

- This is an Immediate Type Instruction.
Opcode = 0010011
 Rs1 = 0
 Imm = 10
 Rd = 10
 Funct3 = 0
 Funct7 = 0

--- This is an ADDI instruction.
```

**Figure 5 A and B: A. Success Case Label and Function   B. Corresponding Instruction**