Matthew Hileman
Professor Gedare Bloom
Computer Architecture - CS4200
September 25, 2021
Project 1 Report

RISC-V (RV32I) Assembler

Introduction

The RISC-V architecture is an open source instruction set developed by UC Berkeley. This project's goal is to create an assembler for the architecture. We were given a parser that reads assembly language into a custom structure, as provided by professor Gedare Bloom from UCCS. We then completed the assembler by interpreting instructions and data and converting them to raw numeric form. The assembler takes an example program and turns it into a machine readable language.

The soul project member is Matthew Hileman, who was in charge of creating and refining all aspects of the assembler. The assembler is organized into two different fields, each with their own sub functions - the assembler, and the linker. The assembler converts the program into binary, while the linker handles linking addresses and labels.

The Assembler

Starting the project was a matter of planning and understanding what needed to be done. The task seemed overwhelming, and it was decided the best way to start was with interpreting raw instructions into binary. First, getting the standard base integer instructions to output the correct result, then the pseudo, and finally interpreting the .data section. The files under RISCV_32I_Assembler covers this decoding.

The assembler has one main loop under the method called “assemble_program.” This iterates over one line of parsed information at a time, then proceeds to call the proper methods to handle the information given. It first passes through a state machine that determines if the line is in the .data or .text segment. If a line is in neither .text or .data, it is discarded. The method _token_list_to_array then takes the linked list of arguments stored in the line and converts it to an array so it can be passed to a struct with the number of arguments in the given line. Supporting methods are called if the line is a data type or text type.

There are two methods to handle instructions in the .text segment - _instruction_to_binary and _pseudo_to_binary, depending on if the instruction is a pseudo instruction or a base instruction. These methods then search for the instruction in a large if/else statement then assemble that instruction according to its type/specifics. Since there are many r-type and i-type instructions, methods were created to store this process (_assemble_r_type and _assemble_i_type), where the op code and function codes were sent through.

At the bottom of RISCV_32I_Assembler.c, there are numerous _bind_ methods. The idea was to create the placement of arguments and immediates in constant-like methods. This is where the encoding happens and the proper numeric values are placed in their proper locations in an instruction’s binary. Some instructions had

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

The formats used to encode the _bind_ methods. Some arguments required heavy operations to insert, such as the J-type immediate.

Chart courtesy of James Zhu <jameszhu@berkeley.edu>.

some unique immediate placement and required their own personal methods (such as srai).
The _pseudo_to_binary function handled very similarly to the base instruction method, except it would call the base method with the arguments required for the pseudo instruction. In the case of multiple instructions, a trigger was set in the main public loop to handle it.

After instructions in the .text segment are converted to their numeric representation, they are then stored in a structure called sAssembledInstruction. If the line had a label, it is also stored in this structure. A linker flag or code is set for later if needed, as is the case with an instruction with a target label.

Now that the .text segment is stored in an organized struct, the .data also needs to be. Another method handles .data lines, similar to the .text's, and is called _data_to_binary. This method first checks what data type is being called - align, asciiz, space, or word, and then converts to that line's numeric representation, similar to _instruction_to_binary. The data is then stored in another structured called sAssembledData, as like with sAssembledInstruction. Finally, the binary from both these structs is stored in a final structured called sAssembledProgram to be used by the linker.

Challenges Creating Assembler

The assembler was the most difficult part of this project, and the problems were numerous. Having not practiced C in numerous years, the team was struggling with both the understanding of the chosen language and the RISV architecture. After having completed the project, the team's understanding of C has greatly increased, even if it was not the main purpose behind such an assignment. As for specific

issues behind the assembler portion, for the sake of length, only a few will be named.

The reason behind _get_reg: as we were working out arguments, a way to convert a register from string to numeric was needed - but this also became a way to parse out a string from an argument with ()'s. For example, lw t1, 4(a0) is parsed in as: lw t1 4(a0). In order to separate 4 and a0 into their own portions, C methods known as strtol and strstr and was utilized. This first gets the immediate out front, and the second gets the string without the ()'s. However, there is a condition to this workaround - the increment in _get_reg had to be reversed as to not find registers contained in the name of other registers first, such as finding the register x1 in x11. This also has an issue with faulty declaration with one of these ()'s instructions, such as lw t1 4(a0 x11) would return a0 as a0 is found first. Not ideal, but it works.

The method _get_imm is has a well designed functionality. At first, the assembler only supported base 10 immediates. It was then realized that this method already converts from a string in base 10, so all that needed happen was a strncmp to also support binary and hex immediates. Much pain and not needed functionality when into accepting binary and hex immediate values before the simplicity of this method was realized.

The linked list construction was difficult to understand at first - and it was especially hard to understand how to incorporate a pseudo instruction that interpreted to multiple standard instructions into this custom linked list structure. Eventually, the solution was condensed into a one line loop that iterated through the returned multi-instruction link.

The Linker

In comparison to the assembler, the linker was quite simple. All the work has been done and stored for the linker to run smoothly. Stored in files called Linker.c/h, the main purpose is to link labels with addresses after the program has all been assembled into an organized structure. This program only consists of one main public function that does just this.

The assembled data is first looped through. In this initial loop, every time a label is read, its address is recorded. Then, the data is put into memory at its correct

address (0x10000000). Each type of data element changed the address, and thus a condition was needed to interpret if the data element was a space, align n, or other.

After the .data completed, the .text segment was looped through and every time a label appeared, its address was also recorded. Each instruction in .text is always 4 bytes, so incrementing was simple in comparison to .data. After all label addresses have been recorded, we loop one final time through .text to insert the addresses into any target labels, where we then insert the final form of the instruction into its proper place in memory.

Issues

After running an example and generating the respective .mxe, the disassemble tool was then used to verify correctness; however, one particular instruction is never interpreted correctly - and that is srai. Upon checking the code for the disassemble tool, it seems that srai was left out.

The assembler does not check meticulously for correctness in instruction usage, and assumes that the instruction is mostly using the correct formatting. Basic usage such as correct address, numeric value, and argument type is present, but argument count checking has not been thoroughly implemented. A basic method calling on each instruction to check the correct number of arguments would do, but would take time.

Conclusions

The assembler is functional. There has been one mega-test created with every instruction required, although full of garbage and not actually a program, it tests the functionality perfectly. The struggle in producing such an intense assembler from little experience in RISC-V and C was draining. The majority of the past few weeks were spent on this assignment alone, and I think it should have been given more time to complete - at least 4 or 5 weeks. For someone experienced in C or RISC-V, the time seemed reasonable.

Resources

The main resource used for this assignment was a RISC-V reference sheet by James Zhu, found here:
https://github.com/jameslzhu/riscv-card/blob/master/riscv-card.pdf

The main source of RISC-V information was from the book “Concepts in Programming Languages,” by John C. Mitchell (2003).

As for other references, countless look ups for C functionality and structure on google.