Lab #2 - Risc-V Assembly Programming

Sam Van Nes, Hunter Mason, Brigid Kelly
ECEGR 2200-02 30APR2018

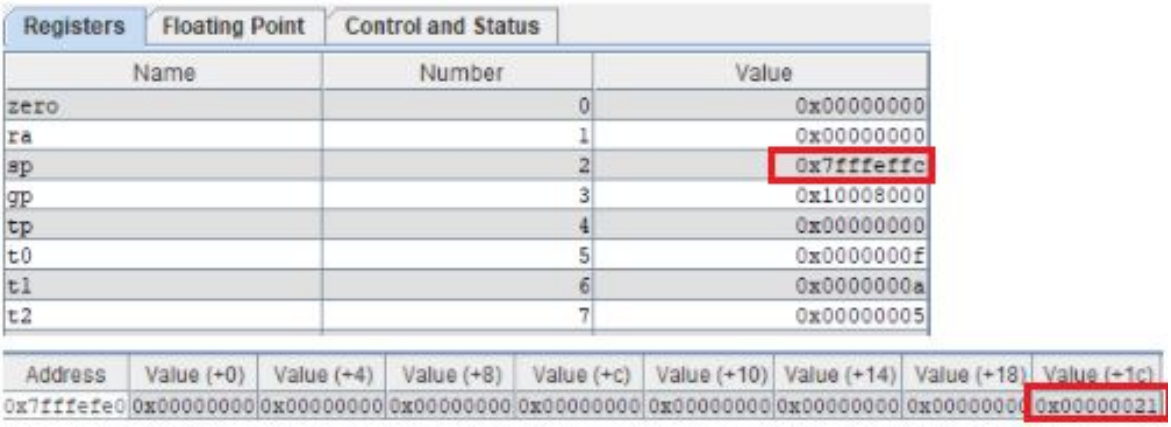
Lab 2- Introduction:

This lab focuses on translating C-Code into RISC-V assembly, essentially acting as a basic C-Code compiler. Each of the 5 parts presents a different unique challenge of coding in RISC-V assembly, from basic math, to array indexing, and function calls. The results of each problem are presented as descriptions of the challenges of the problem, followed by a discussion of the results and context for the images shown.

The RISC-V assembly code in this lab was run on the RARS simulator available here. Many of the responses to the problems in this lab use pseudo-instructions included in the RARS package. The source code of the response to each section can be found in any of the GitHub Repositories linked at the bottom of this document.

Part 1 - Arithmetic:

This part of the lab was simple to implement was it was composed of basic arithmetic operations, as well as divide and multiply. The code portion is implemented as the program is described. Shown below are screenshots showing the location of Z in memory, which is at the address 0x7fffeffc. At this location, the values 0x00000021 is stored. This is what we expect; Z was saved and stored to the sp register, and it saved the proper result of the program, which is 33 in decimal. This shows that our program is working correctly, as this is the result if we did this program by hand. The registers t0-t5 were used to store A-F, and the final result was stored in t6 before being pushed to sp.



Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x0000000f
t1	6	0x0000000a
t2	7	0x00000005

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefe0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000021

Lab Part 2 – Branches:

The challenge to this translation was ensuring that each section of code was excited and jumps to the appropriate label. There was many trial runs where the program continued after code and ran into piece that were only meant to be jumped to if a condition was met. Utilizing the Branch pseudo-instruction available to us in RARS was also greatly beneficial in this section, as it makes the code much easier to come back to and understand.

There is likely a way to make this code more efficient. It has become evident throughout this lab that the most readable version of assembly code is most likely not the most efficient in terms of instruction count and number of jumps/returns.

The given inputs to the code expects an output of -1(0Xffffffff) into the address 0x1001000c (Var Z) shown below in figure 2.1, this value is shown at this address in figure 2.2.

Figure 2.1: Variable Z is pointing to address 0x1001000c

Figure 2.1: Two complement form of -1(0xffffffff) stored at Z's address.

Lab Part 3 – Loops:

As a result of this program containing three loops, the approach was to inspect and create each loop individually. In doing so, this program will be created in chunks rather than one continuous flow. Values 'Z' and 'I' are variables stored in memory that are interacting with three different loops. Once the program is done running, 'Z' is returned to memory in sp as 0x0000004d, which is shown below. This is the value that we calculated; it is 77 in decimal. Both 'Z' and 'I 'are stored in the sp variable, which has an address of 0x7fffefe0. This means that the program works as expected. T0-t5 were used throughout this program, as well as I and Z as words, in order to achieve the desired output.

Lab Part 4 - Arrays:

This part involved the use of both pointers and values within memory and registers as well as the use of equivalency ‘branch’ statements. The intended function was achieved by loading two registers with pointers directed to the first element of each array. Once the addresses were loaded, the first words at the directed addresses were then loaded into separate registers for arithmetical manipulation. The registers containing the addresses of the first element of each array can be seen in *Figure 1* below.

Figure 4.1: Registers a2 and a3 contain the addresses of the first elements in arrays A and B

In the ‘for-loop’ section, the first element addressed by pointer a3 (B[i]) is loaded by value into register a4. Once the value is loaded, the add immediate instruction is called with an immediate value of -1. The result is then placed back into register a4 and the store word instruction is called to place the calculated value into the memory address pointed to by register a2 (This is accomplished by using the SW-Store word instruction). Registers a2 and a3 then each have their values incremented by 4 to address the next element within the array. The final instruction of the ‘for:’ section is a ‘branch-less-than’ instruction, which compares the value of i (stored in register a0) and the value of n (in this case, 5, stored in a1). If the value of i is less than 5, the ‘for:’ section is then called again. The figures below graphically illustrate the for-loop section functioning.

Figure 4.2: Initial state of memory before the for-loop.

Figure 4.3: State of memory after the execution of the for-loop.

Figure 4.4: Registers a2 and a3 now point 20 bytes beyond their initial addresses, indicating the loop has been executed 5 times.

After the execution of the for-loop, 'i' was decremented by 1, n is assigned the value of 5, and a while loop is executed in a similar fashion to the for-loop. The only difference in this section was slightly more complicated arithmetic manipulation and the use of the blt command, comparing x0 (zero) to the value of i as it is counted down from its maximum value to -1, and the addresses pointed to in a2 and a3 have 4 subtracted from them to go back down the element of the arrays.. Figure 4.3 shows the state of memory at the beginning of the while loop. Figure 4.5 below shows the final state of memory at the end of the program and figure 4.6 shows the final state of the registers.

FIgure 4.5: Final state of memory after the completion of the while loop.

Figure 4.6: Final state of registers after completion of the entire program.

Lessons learned included using BLT at the end of the while-loop to branch back to the top instead of using a BGE instruction at the top, as using jump and bge caused the program to loop infinitely due to the loop being in the following instructions after the BGE statement. Another lesson learned is to possibly use temporary registers for items being constantly updated instead of using main registers such as a0 and a1 for trivial values.

Lab Part 5 – Function Call:

This part was very linear to implement. The hardest part was making sure that all registers that need to be preserved were stored at call time. The calling conventions needed for this code were only those of the callers responsibility for its temp variables. No other registers were in use or in danger of overwriting as there was no nested functions.

A code trace gave me an expected result of 70_{10} which is shown in the address 0x10010008(Variable C) below(fig 5.1) in hex(70_{10} = 0x46). The code also keeps i and j in t0,t1

while in the main body and the function calls. This demonstrates that the storage of this data to stack memory is being done correctly at the time of each call.

Figure 5.1: Memory address of Variable C.

Figure 5.2: The correct output shown in Variable C's memory address.

Lab 2 - Conclusion:

Coding in assembly is very different than coding in higher level languages; it requires planning of registers and implementing each step of a program or function individually, whereas in higher levels, the compiler takes care of many of the intermediary steps. This sort of programming contains much simpler instructions compared to higher levels, and does not allow for the automatic conversion of variable types or functions. Understanding the basis of how a compiler translates code is a very integral part of learning design and optimization of hardware and parts. In essence, assembly code is much faster and very specific built for a single purpose, much like a race car, whereas higher level code is more versatile and contains many more well-rounded features for a greater range of functions and uses, such as being able to use templated objects.

Lab 2 - GitHub Repositories:

Hunter Mason: https://github.com/HunterM982/Labs
Brigid Kelly: https://github.com/BAKelly12/Labs
Samuel Van Nes: https://github.com/SamuelVanNes/Labs