Name

Owusu Kwabena Joseph

Student ID

15542936

Scientific Protocol: STR/LDR ARMv8

(December 8, 2024)

Hochschule Ravensburg-Weingarten

University of Applied Sciences

B

Abstract

Introduction

Computer architecture is the programmers view of the computer and in oder to speak the language of a computer, there is no way out aside studying the *instruction set architecture* of that processor. Unlike high level programming, arithmetic operands in assembly have limited location built directly in the hardware called registers. The size of a register in ARMv8 is 64 bit and what happens when we exceed this capacity or limit of the hardware? Also what happens when we add negative numbers or a negative(*signed numbers*) to a positive number (*unsigned numbers*) and we get a posistve results? That's the condition we call an overflow or somtimes out of range and at such instance, the computer, *OS*, or the programm running decides how it's going to react or respond. The compiler allocates data structures like arrays and structures to locations in memory and an event of data transfer, the compiler again places the right address in the data transfer instruction. According to([1]) *ISA* which is percived as the language of the processor maybe seen as diverse as those of humans but in principle computer languages are quite similar, more like regional dialect than independent languages. This is because the hardware implementation of computers are more or less quite similar. In effect one can easily learn the next *architecture* if he or she has a good understanding of an *ISA* eg. MIPS, ARMv7 etc under his or her belt. Besides ARM is the most widely populated pocessors around the globe so our main focus in this experiment was the ARM Cortex A53. High level languages like java, c python etc has the flexibility of using many variable but unlike assembly programming which has a fixed number of variables. Memory holds a chunk of data and has a bigger size as compared to registers. Registers are where operands are worked on and could be percived as variables. In contrast to memory, registers are also a smaller memory space and are much more quicker and therefore prefered for data operations like addition, subtraction jumping and branching.The compiler tries to keep frequently used variables on the register and the rest in the memory using *Load* and *store* to move variables between registers and memory. In oder to work on a data within the register, one needs a fetch instruction called load *LDR* in ARM cortex. The reverse of storing data is called store *STR*.

We have a limit amount of memory space and therefore we limited to some number of count 2 to the power 64-1 bits. our main goal in this experiment is to study how the *ARM Cortex A53* could react to overflow, out of range, which may produce error in our codes.If the results of an operation can not be represented then we call in the available space or range representation bits example 64 bits then an overflow has occured and its upto to the operating system, the

programming language and the program to decide what to do.

The CPU doesn’t automatically "spill over" excess bits into other registers. Arithmetic logic is designed to work within the bounds of the register size.

Materials and Methods

For the demand of our experiment, we needed both hardware and software tools in order to have a successful experiment. We had a manual script(see[2]) that symmarized our materials and methods.

Hardware

1. A Raspberry Pi computer with ARMv8 architecture, including all other peripherals such as a monitor, mouse, and keyboard.

Software

1. **Linux Distribution for ARM (e.g., Ubuntu):** We needed a version of the Linux operating system, such as Ubuntu-based RaspbianOS, compatible with the ARM architecture. This is essential for the Raspberry Pi setup.

2. **Development Tools:** These tools are essential for programming in the Linux environment:

    a. **ARMv8 Instruction Set Documentation:** This document serves as the primary guide and reference point for understanding the data management and architecture instructions.

    b. **Text Editor (Geany):** The default text editor for coding.

    c. **Assembler:** For compiling the assembly code into bytecode.

    d. **C Compiler and Linker (gcc):** The GNU project C compiler for C programming.

    e. **GDB Debugger:** GDB command line interface used for debugging your programs step by step and tracking register and memory values.

The Makefile

We needed executable files and therefore all our source codes were compiled, the makefile makes it easier for these compilation processes. If not for the Makefile, one would have to compile to compile individually our c code, assembly and clean them after runtime. The makefile automated all these processes and this done by typing *make*, *make all* and *make clean* on our command line.

The assemblyfile

This file contained our assembly code. In this file, one could see we a list of some registers that we used in the loading and storing process, also for branching and conditonal moves to verify our output of either an overflow occured during our testing or not.

C source code File

This file contained the main program from which the assembly routine called. It waS ofcourse very important for us to understand the c code in order to appreciate the assembly routines.

Methods

In other to make our work easier, one needs to research(see[3]) and understand simple register load and store, pre-indexed load and store, offset load and store, offset shift load and store and post-indexed load and store. Developing an ARMv8 assembly program can be challenging, but following the right procedure ensures the job gets done efficiently. Before diving into the details, it's important to familiarize yourself with some Linux command lines that are essential for compiling, running, and executing C and assembly code.

Understanding the assembly code

Understanding the C Code

First, let's review the C code(Listing 1) to understand how the logic is implemented. After including the necessary libraries, we encounter the `extern` keyword, which links the assembly code to the C program.

In the `main` function, two variables are initialized: `val2str` and `val2ldr`. The C program prompts the user to input a integer value and then saves it to `val2str`. The `val2ldr` is never changed by the c program only the assembly function accesses the value of this variable (writes to it's address in memory actually).

Then it calls the assembly function with two parameters the data of `val2str` and the address of `val2ldr`.

After running the assembly function, the c program prints the value of the `val2ldr` demontrating that the assembly actually change worked, by assigning the data present in

`val2str` to `val2ldr`.

nderstanding the Makefile

Key Linux Terminal Commands

The following GDB commands are important for developing and debugging assembly code:

- `l`: List the available instructions.
- `i r`: List all registers.
- `i r X0`: List the contents of the `X0` register.
- `r`: Run the program or execute our code
- `s`: Step through the program one instruction at a time and this was mainly used during the debugging process.
- `b strldr_asm`: Set a breakpoint at the `strldr_asm` label in the assembly code.
- `p variable`: Display the value of a variable.
- `p &variable`: Display the memory address of a variable.

Throughout our experiment, it was very important for us to know and understand the conversion between decimal, binary, and hexadecimal. This helped us to understand the various changes in the registers during the store and load operations. Before a machine language programmer could start programming, it was also important for us to understand the ARMv8 instruction set architecture and the mechanism between our C code and the assembly code.

The Makefile was essential in automating the build process for software projects, particularly when we were handling multiple source files written in different languages, C and assembly. It defined how to compile and link the various components of a project, allowing developers to manage large projects efficiently without manually compiling each file. The experiment demonstrated how to store and load from main memory in different modes, as provided above. Each code's behavior was different, depending on its mode, and the impact of the initial value in register X6.

For us to enter into the debugging mode we had to type (*make debug*) on our terminal and a breakpoint was set using *b strldr_asm*, then we entered the value to be stored and ran the

program by typing *r* and then press the enter key. For us to list our register we entered i r, and we stepped into our program simply typing *s*. This together with the breakpoint was very important because it helped us to analyse and execute our code line by line. This allowed us to modify the code depending on the required mode, and the experiment was examined successfully in the lab. For control and monitoring of our program, we took screenshot of every *step command* and did analysis to fully understand the various load and store processing and main the major differences between all these methods which we will deeply into the results section. After compiling both our c and assembly code with the *make all* command, we run our code and entered 24 as the value to be stored (see**??**).

Results

Discussion

Computer architecture is the programmer's view of the computer. To speak the language of a computer, one must study the *instruction set architecture* (ISA) of that processor. Unlike high-level programming, arithmetic operands in assembly have limited locations built directly into the hardware called registers. The size of a register in ARMv8 is 64 bits. What happens when we exceed this capacity or limit of the hardware? Additionally, what happens when we add negative numbers (signed numbers) to positive numbers (unsigned numbers) and get a positive result? This condition is called an overflow or sometimes out of range. In such instances, the computer, operating system (OS), or the running program decides how to react or respond.

The compiler allocates data structures like arrays and structures to locations in memory. During data transfer, the compiler places the correct address in the data transfer instruction. According to [1], the ISA, perceived as the language of the processor, may seem as diverse as human languages. However, in principle, computer languages are quite similar, more like regional dialects than independent languages. This similarity is because the hardware implementation of computers is more or less quite similar. Consequently, one can easily learn a new architecture if they have a good understanding of an ISA, such as MIPS or ARMv7. Besides, ARM is the most widely used processor globally, so our main focus in this experiment was the ARM Cortex A53.

High-level languages like Java, C, and Python offer the flexibility of using many variables, unlike assembly programming, which has a fixed number of variables. Memory holds a large amount of data and is bigger compared to registers. Registers, where operands are worked on, can be perceived as variables. In contrast to memory, registers are smaller and much quicker, making them preferred for data operations like addition, subtraction, jumping, and branching. The compiler tries to keep frequently used variables in the registers and the rest in memory, using *Load* and *Store* instructions to move variables between registers and memory.

To work on data within the register, one needs a fetch instruction called load (*LDR*) in ARM Cortex. The reverse of storing data is called store (*STR*).

We have a limited amount of memory space, restricted to a count of $2^{64} - 1$ bits. Our main goal in this experiment is to study how the ARM Cortex A53 reacts to overflow or out-of-range conditions, which may produce errors in our code. If the results of an operation cannot be represented within the available space or range of 64 bits, an overflow occurs. It is then up to the operating system, programming language, and the program to decide what to do.

The CPU doesn’t automatically "spill over" excess bits into other registers. Arithmetic logic is designed to work within the bounds of the register size.

Appendix

```c
#include <stdio.h>
#include <stdlib.h>

extern long strldr_asm (long val01, long *val02);

int main()
{
    long val2str, val2ldr;
    int error;

    printf("Please give the value to be stored -> ");
    scanf("%ld", &val2str);
    printf("Value to be stored: %ld\n", val2str);

    error = strldr_asm(val2str, &val2ldr);   // Calls the
       assembly program

    if (error == -1)
        printf("Error!\n");
    else
        printf("Value read is -> %ld\n", val2ldr);

    return 0;
}
```

Listing 1: Sample C Program with Assembly Call

```
MOV X0, #10
MOV X1, #20
ADD X2, X0, X1
```

Listing 2: First Assembly Code Example

```
1 /* strldr_asm.asm */
2   /* X1 = address/pointer */
3   .global strldr_asm
4   strldr_asm:
5             MOV X6,X0
6             MOV X7,X1                /* Address for storing
                 the value; take address from X1 */
7             ADD X7,X7,#8             /* ... and add 8. (X7=0
                 x7ffffff33c) */
8             MOV X9,#0x8              /* Could act as an
                 address offset */
9             /* do your work*/
10            Str X6, [X7]             /* (1) store with
                 register */
11            Ldr X8, [X7]             /* load */
12
13            Str X6, [X7, #8]!        /* (2) pre-indexed */
14            Ldr X8, [X7]             /* load */
15
16            Str X6, [X7, X9, LSL #0] /* (3) offset */
17            Ldr X6, [X7, X9, LSL #0] /* load with offset */
18
19            Str X6, [X7, X9, LSL #3] /* (4) offset with shift
                 */
20            Ldr X6, [X7, X9, LSL #3]
21            Str X6, [X7], #8
22
23            /finish/
24            MOV X0,#0
25            BR LR
```

Listing 3: ARM Assembly Code for strldr_asm.asm

Bibliography

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware Software Interface, ARM Edition* (The Morgan Kaufmann Series in Computer Architecture and Design). Morgan Kaufmann, 2016.

[2] P. D.-I. A. Siggelkow, *Computer technology lab script*, Accessed: 2024-10-16, 2024. [Online]. Available: https://elearning.rwu.de/pluginfile.php/250305/mod_resource/content/3/00_rechlab_top.pdf.

[3] A. Developer, *Loads and stores*, Accessed: 2024-10-16, Accessed: 2024. [Online]. Available: https://developer.arm.com/documentation/102374/0102/Loads-and-stores.