z/Architecture and Assembly

Bringing the *rush* Programming Language to the S/390X Architecture

Mik Müller

Hasso Plattner Institute for Digital Engineering
mik.mueller@student.hpi.de

1 Introduction

> [System/360] was the biggest, riskiest decision I ever made, and I agonized about it for weeks, but deep down I believed there was nothing IBM couldn't do.
>
> *Tom Watson, Jr. IBM Chairman and CEO 1961-1971*

The S/390 architecture is the longest-living CPU architecture with the longest heritage that is still in use today. Its core was first designed when compilers were very new technology, and the majority of programs were manually written in assembler language [6, p.1]. One reason for this is that the IBM System Z family, which is being used today, is fully backwards compatible with earlier versions of IBM's mainframe computers [5]. In fact, System/Z is a direct descendant of the original System/360, which made its debut all the way back in 1964. Over the years, S/360 and its successors have been used for all kinds of computational work, spanning from military applications in the *Cold War* to scientific calculations powering the moon landing at the end of that century [8]. Today, six decades later, a large portion of applications written in the 1960s can still run unmodified on the latest bleeding-edge System/Z computers [1, p.11]. These newer System/Z machines still run on the System/390 CPU architecture, which was introduced in the 90s [4, 10]. This architecture is the direct successor of System/370, which was the direct successor of the original System/360 [9]. Therefore, this article will use "S/390X" in order to describe the CPU architecture used by Z systems, with the "X" standing for the 64-bit variant of the original architecture. This article presents an overview about the S/390X CPU architecture. In Section 2, the reader will gain insights into general traits of the architecture, followed by some assembler language basics. Later, Section 3 addresses quirks of the architecture and presents a compiler backend for the "rush" programming language in order to showcase how certain traits of the architecture impact compilers as well as humans producing S/390X assembly code.

2 S/390X Architecture Overview

The S/390x architecture is based on S/360, which has received numerous updates and enhancements in order to become the architecture that is being used on System/Z today. Among others, those changes include 64-bit registers, and a multitude of novel instructions. Nowadays, S/390X provides 16 general purpose registers (this article uses $\texttt{GR}_x$ for general register x). For floating-point processing, another set of 16 float registers is available. The integer and floating-point registers are all numbered 0-15, respectively. In the original S/360 architecture, only 4 floating-point registers were available due to the costly nature of floating-point arithmetic. Even though not all general purpose registers are semantically equivalent, there are no reserved register names, like '`rip`' on x86 systems [2, p.52]. Since S/390X is the 64-bit variant of S/390, all registers are able to hold 8 bytes (64 bits). The architecture is structured in a typical 2-address fashion, where for most instructions, the first operand address is also the destination of the operation. Few instructions, like those used for division, only operate on pairs of registers [6]. This behavior is later discussed in Section 3. For ELF binaries on Linux, the ABI dictates that $\texttt{GR}_{15}$ is to be used as the stack pointer. Furthermore, $\texttt{GR}_{14}$ is to be used as a function call return address.

2.1 Instructions

Table 1 – Common instruction formats on S/390X [2, p.51]

Short	Long
`RR`	register-register
`RX`	reg-idx-storage
`RS`	register-storage
`SI`	storage-immediate
`SS`	storage-storage

Apart from the few original instruction formats defined by S/360, the S/390 architecture now defines well over 30 different variants. Some basic examples are '`RR`'-type instructions, meaning *r*egister-to-*r*egister. For instance, there are variants of addition or subtraction instructions that operate on two registers and store the result in the first operand. Furthermore, '`RX`'-type instructions use one *r*egister and inde*x*ed storage. Among others, most load / store instructions typically fall into that category, as they might, in case of load, fetch data from memory and store it in a destination register. As a general rule, most ALU operations can be performed using '`RR`' and '`RX`' instructions. For those, it depends on how they need to integrate into the program, as a register-storage operation tends to spare the programmer from inserting redundant load / store instructions. The existence of rather complex register-storage instructions is often a typical characteristic of CISC architectures, like S/390X or x86. Other commonly used instruction formats are displayed in table2. The word size on S/390X is 32 bits. Other variants, like a halfword (16 bits) or quadword (128 bits), can be derived easily. Most unspecialized instructions come in different variants, depending on the size of the operand data. Common instruction variants include byte-, halfword-, word-, and doubleword-operands. As for instruction encoding, an instruction's first byte always determines the operation code, whereas the second byte usually provides information about location, type, or length of the processed data. In some newer instruction formats, another part of the opcode can be found in the second and even sixth byte. The first two bits of the opcode reveal how many bytes the CPU has fetch from memory. However, the CPU fetches at least two bytes per instruction eagerly [2, p.52]. Alongside many other modern architectures, S/390X can be seen as a von Neumann architecture. Every processor needs to keep track of the instruction that is currently being executed. In most of today's architectures, this is achieved by storing its address in a special register called the *instruction pointer*. However, S/390X does not dedicate an entire register to store this counter. Instead, it is embedded into a special-purpose register called the *program-status-word* (PSW), which comprises multiple fields. Among other fields, the so-called *instruction address* (IA), is stored at the rightmost position. Furthermore, a 2-bit field called the *condition-code* (CC) resides in the upper portion of the PSW. The CC is set, and queried by the CPU in order to keep track of side effects caused by different operations. For instance, the CC might be set if the result of an arithmetic operation has caused an overflow that requires attention. Furthermore, the CC might be queried to determine if a jump, shall be taken or skipped. Since those conditional jumps mostly occur immediately after a comparison instruction, which was used to set the CC, they are predominantly used in order to check whether a given relation exists between two operands [2, p.85].

2.2 Memory

Since 1964, the original 8KB of memory usable on S/360 have now evolved to the 40 TB of memory usable on the IBM z16 today [7, 8]. In order to address these vast amounts of memory, sophisticated addressing is required. Even though modern IBM System/Z configurations support addressing memory belonging to a different CPU, the following subsection will focus on the basics of memory addressing, which are used when writing assembler programs. Most instructions, except RR-type instructions, utilize an *addressing halfword* in order to compute the address of operands. When writing assembler programs, the programmer specifies a "base register", which holds parts of the so-called "effective address" (EA). The effective address is stored inside the internal 'EAR' register. Apart from addressing operands in memory, some instructions, like shift operations, use the 'EAR' in order to calculate their shift amount. The base register contains a base value, which is used to calculate the effective address. A smaller immediate value (12 bits), called the *displacement*, in the instruction is then added onto the base value to form the effective address. This way, common operations, like storing a value on the stack using an offset, can be performed with ease. In case the base

Table 2 – Common instr. on S/390X [2, p.51]

Short	Long
`stg` r_x, `0(`r_y`)`	Store (64)
`lg` r_x, `0(`r_y`)`	Load (64)
`lgr` r_x, r_y	Load (64)
`agr` r_x, r_y	Add (64)
`agfi` r_x, `imm`	Add Immediate (64)
`brasl` r_x, `lbl`	Branch Relative and Save Long
`br` r_x	Branch
`cgr` r_x, r_y	Compare (64)
`j lbl`	Jump

register is $\mathtt{GR}_0$, it is dismissed, and no base register is used during EA calculation. As as result of this, the EAR will be equivalent to the displacement. In Section 3, this idiosyncrasy will be discussed in greater detail. An example of using this addressing technique is the following instruction: `st %r1, 8(%r15)` . This instruction would access 4 bytes of the $\mathtt{GR}_1$ and store them at the address of the stack pointer, with an offset of 8 ($\mathtt{GR}_{15} + 8$). Like explained previously, $\mathtt{GR}_{15}$, is used to denote the stack pointer address, which serves as a base register for most instructions which need to address memory [2, p.62]. Exceptions to this rule are mostly pointer dereferences. In that case, a register $\mathtt{GR}_x$ is populated with data that *is* saved on the stack, only to be used as a base register for another load operation, where $\mathtt{GR}_x$ is the base register in question. An example would be `l %r1, 16(%r16)` , followed by `l %r2, 0(%r1)` , where the address of the variable the pointer points to is loaded into $\mathtt{GR}_1$ (which is the previous $\mathtt{GR}_x$). Then, this address is used as a base to perform the load operation (into $\mathtt{GR}_2$) which fetches the actual data stored in the variable behind the pointer. When discussing memory, it is of significance to point out that S/390 is one of the few big-endian architectures still in use today, as the majority of the world's systems has settled on using little-endian. Obviously, this creates an impact for programs dealing with raw binary data. Almost always, a memory operand is referred to by its lowest-addressed byte. In case that operand is a binary integer, that byte contains the most significant (high-order) byte of the integer. Another scenario where caution is advised arises from the usage of data-labels[1], which are often used for floating-point constants and global variables.

2.3 Assembler Programming

As usual, the first part of an assembler instruction specifies the opcode mnemonic, which is later translated into the actual opcode by the assembler. Customarily, IBM System/Z uses Z/OS[2] as its operating system. In order to facilitate programming, Z/OS ships a myriad of programs, to which UNIX programmers are usually not accustomed. One of those specialties is the assembler distributed with Z/OS. This software is called "IBM High Level Assembler for z/OS & z/VM & z/VSE", also known as "HLASM" and is commonly used to write S/390X assembly programs on Z/OS. However, this software tool carries a nontrivial amount of legacy constraints that may feel awkward for programmers coming from more modern environments. For legacy reasons, using "IBM Cards" (punch cards) as the program source code, each line may span over a maximum of 80 characters. Furthermore, statements, comments, and data are to be used solely in their designated columns, otherwise causing the input program to be rejected by the assembler. Since those constraints might become quite cumbersome in practice, the rest of this article will use the GNU Assembler (GAS) as a replacement for HLASM. A workflow used to develop S/390X programs with relative ease is discussed in Section 3.

[1]A label in the '.data' or '.rodata' section of an ELF-binary, which is followed by a data constant.
[2]It is to be noted that there is another report which discusses this topic.

3 S/390X Quirks: Impact on Compilers and Programmers

3.1 Workflow

For this report, I implemented an additional compiler backend for the *rush* programming language [3] so that it supports cross compilation to S/390X assembly. The language features a static type system, six different data types[3], arithmetic operators, logical operators, local and global variables, pointers, if-else expressions, loops, and functions. To introduce the language, the reader might consider the code in Listing 1.1.

```
fn main() {
    exit(fib(10))
}

fn fib(n: int) -> int {
    if n < 2 {
        n
    } else {
        fib(n - 2) +
        fib(n - 1)
    }
}
```

Listing 1.1 – Recursive Fibonacci Number Generation in rush.

For assembler development on S/390X, I have used the GNU binutils suite in order to perform compiling, assembling, linking, and archiving[4]. All typical components of the GNU binutils have long been ported to support S/390X. In the following, those software tools have either been used like a cross-compiler (on an x86 machine, targeting S/390X), or on S/390X directly. A compiler must be able to correctly translate its input into another representation, in this case S/390X assembly. In order to produce correct output, all special characteristics of the target architecture must be known to the compiler developer. For instance, compilation of *if-else* expressions require the following building-blocks: **(1) Boolean Expressions**, which are mostly comparisons performed on other data types, like integers. Compilation of these expressions is required so that they can be used as the condition of the *if-else* expression. **(2) Branching** in order to take the *if* or *else* branch, depending on whether the condition is *true* or *false*. Therefore, complete research is needed on how to apply these basic operations to assembler language. Once they are learned, they can be used to implement the compiler features, in that case, *if-else* expressions. Sometimes, the manuals were not intuitive enough and additional support was needed to understand how

[3]Meaning the *int*, *float*, *bool*, *char*, *unit*, and *never* type.
[4]Tools used: *GCC*, *GAS*, *LD*, *AR*, respectively

operations could be performed. For this, I utilized the S/390X port of GCC in order to compile a small C program, which used C constructs that I suspected to compile to the output I required, to S/390X assembler language. Then, I would carefully bisect the assembler program, removing any unimportant instructions and compiler boilerplate until I had a program that either caused an exception or completed as I expected. This process would be repeated until the smallest possible subset of instructions needed to implement a feature could be identified. Then, I would look up the used instructions in the two primary sources of documentation I used, in order to fully understand their purpose and whether they suited my purposes. Most of the time, this worked relatively well. Sometimes, an assembler program would cause unexpected exceptions, mostly segmentation faults, even though the program appeared to be valid. In those cases, I utilized the GNU debugger (GDB) for S/390X. Since it would be straight forward to execute GDB on the target machine, access to a mainframe was needed. For more basic tasks, like simple S/390X ELF binary execution, I utilized QEMU, which made the development cycle much swifter. Thankfully, IBM runs a program called the "LinuxONE Community Cloud", which is hosted by Marist College in Poughkeepsie. Therefore, I could access a private Virtual Machine running a Linux port for s/390X in order to perform those more demanding tasks. As the goal of rush was to implement many execution backends, the project features a rich test suite so that a backend could be at least partially validated to be correct. For most language features (such as *if-else* constructs), a separate test case is provided. This way, each language feature could be first implemented in the compiler, then evaluated for correctness using the test case. Since a compiler can be a highly-interconnected piece of software, the compilation of construct A might impact the compilation of construct B negatively, causing the compilation output to be invalid. In order to test against such cases, multiple test cases using all language features together are provided. For other programming languages, I have developed a system called "semantic fuzzing", which obfuscates an input program with a known output arbitrarily. However, the semantic meaning must be preserved during the obfuscation process, so that the outputs of both the obfuscated and original program can be compared for equality.

3.2 The rush Compiler Backend

```
 1  #...
 2  main..fib:
 3      aghi %r15, -40
 4      stg %r14, 0(%r15)
 5      stg %r2, 8(%r15)
 6      lg %r0, 8(%r15)
 7      lgfi %r1, 2
 8      cgr %r0, %r1
 9      jl comparison_true_0
10      lghi %r0, 0
11      j comparison_merge_0
12
13  comparison_true_0:
14      lghi %r0, 1
15
16  comparison_merge_0:
17      chi %r0, 0
18      je else_0
19      lg %r0, 8(%r15)
20      j merge_0
21
22  else_0:
23      lg %r0, 8(%r15)
24      lgfi %r1, 1
25      sgr %r0, %r1
26      lgr %r2, %r0
27      brasl %r14, main..fib
28      stg %r2, 24(%r15)
29      lg %r0, 8(%r15)
30      lgfi %r1, 2
31      sgr %r0, %r1
32      lgr %r2, %r0
33      brasl %r14, main..fib
34      lgr %r0, %r2
35      lg %r2, 24(%r15)
36      agr %r2, %r0
37      lgr %r0, %r2
38      j merge_0
39
40  merge_0:
41      lgr %r2, %r0
42
43  epilogue_1:
44      lg %r14, 0(%r15)
45      aghi %r15, 40
46      br %r14
```

Listing 1.2 – Compiler Output from Listing 1.1 (trimmed).

Listing 1.2 shows parts of the compiler output generated by the rush S/390X backend using Listing 1.1 as the input file. In line 2, the *fib* function is defined. Since this is a recursive function, at least the return address, which is saved in GR$_{14}$, needs to be saved on the stack. For this, the `stg` instruction is used in line 4. Here, GR$_{15}$ is used as the base register of the destination address since it specifies the stack pointer address. However, the stack pointer must first be decremented (line 3) in order to *allocate* space for values saved by the function, like the return address. In line 5, GR$_{2}$, which contains the input parameter, is saved on the stack as it will be overwritten as soon as the recursive call places its return value in it. Since *fib* is called two consecutive times in the source program, the original value of the parameter (n) must be preserved. In line 7, the value 2 is loaded into GR$_{1}$, which is used in line 8 in the `cgr` instruction. This instruction is used in order to compare the value 2 and the input parameter (n). The lines 9-16 are boilerplate used in order to evaluate the result of the comparison. This quirk is later discussed in Subsection 3.3. In lines 17 and 18, the result of the comparison is used in order to take the *if* or *else* branch. For this, the `je` (*jump-equal*) instruction is used to take the *else* branch if the result of the comparison is equal to *zero*, meaning *false*. Otherwise, the *if* branch is taken. This branch is rather trivial, as it merely returns the input n again.

The other branch is more intricate, as it calls the *fib* function recursively. Therefore, we will now consider the code following lines 22. Lines 23-26 are used in order to load the value of n and decrement it by 1. Then, the result of the subtraction is loaded into GR$_{2}$ in order to use it as the input parameter of the recursive call. In line 27, the *fib* function is called recursively using the `brasl` instruction. Like Table **??** suggests, `brasl` is used in order to jump to an address and store the return address in the given register $\texttt{r}_x$, which is $\texttt{GR}_{14}$ by convention. In lines 28-32, this pattern is found again, only that 2 is being subtracted from the input paramter (n). The following lines are then responsible for adding the results of both recursive calls together, with the `agr` instruction in line 36 performing the actual addition. All remaining instructions form the so-called *epilogue*, which is responsible for restoring the return address and stack pointer.

3.3 Quirks

During the development of the rush compiler backend, several quirks of the S/390X architecture have been discovered. Most of them did have an impact on how the compiler was implemented. Like hinted at in Subsection 2.2, $\texttt{GR}_0$ cannot be used as a base register in effective address calculation. Since this quirk was not immediately apparent while studying the manuals, the compiler initially used to emit code that used $\texttt{GR}_0$ for this purpose. When testing the generated code, it became apparent that it caused a *segmentation fault*. This is due to the fact that the base register was omitted and only the immediate was used in the effective address generation. Therefore, the program attempted to access code at mostly very low memory addresses, which are likely reserved for firmware and kernel use. After studying the manuals further, the isssue became apparent and was resolved by modifying the compiler. For almost all operations, the compiler uses an internal method to select a spare register, so that it can be used as an operand for an instruction. In this case, the method could return $\texttt{GR}_0$ even though it was called in order to acquire a base register. Parts of the compiler emitting instructions which acccess memory were then patched in order to always use a register higher than $\texttt{GR}_0$. Furthermore, evaluation of comparison infix expressions proved to be more intricate than originally anticipated. Ideally, one could use an instruction that compares two registers and stores the result of the comparison as a boolean value in another register. This pattern is found in RISC–V, but not in S/390X. Even though the `cgr` instruction causes the CC to be set, there is no trivial way of accessing it. The method implemented in the compiler emits a compare instruction, immediately followed by a conditional jump, which is dependent on the *infix-operator* being used. For instance, a `jl` instruction is emitted if the infix-operator is '<'. The branch then either jumps to a *true* or *false* block. Those blocks either place the value 0 or 1 in a destination register, depending on whether *true* or *false* is to be represented. Any following code can then access the destination register in order to perform further operations on it. Those expressions are often utilized as the condition of *if-else* expressions. Therefore, the compilation of the those two components creates tremendously inefficient code. As a rule of thumb, branches slow down a program significantly, as they cause pipeline-stalls in the CPU. However, the emitted code contains an additional conditional branch, which only serves to compute the boolean infix expression. The result of this expression is then again tested against 0 so that the *if* or *else* branch can be taken. An obvious compiler optimization would be to perform pattern matching on the condition subtree of the *if-else* expression so that the conditional jumps that are currently emitted by the infix expression can target the *if* and *else* branches directly. Moreover, implementing division was quite interesting. In RISC–V, division can be achieved by using the appropriate instruction, which uses a destination- and two source registers. Other architectures, like x86 or S/390X behave differently. Frequently, the `dsgr` instruction is used to perform integer division on S/390X. Specific input and output restrictions lead to this instruction being rather special. The architecture mandates that dividend and divisor are supplied in even-numbered registers, like $\texttt{GR}_2$ or $\texttt{GR}_4$, respectively. After the instruction has executed without any interruptions, the quotient is found in the odd-numbered register belonging to the dividend, meaning $\texttt{GR}_3$ in this case. In case one intends to access the remainder, the same register used for the dividend will contain this value. Apart from these quirks, development of the compiler was relatively straight-forward. Overall, I think that for its age, S/390X is a relatively intuitive architecture.

4 Related Work

The previously mentioned *rush* programming language was initially developed in order to compare different methods of execution. As this report is based on a rush compiler targeting S/390X, it is recommended to peruse the original rush paper [3].

Furthermore, it is recommended to take a look at the other reports, especially "Linux on IBM Z and LinuxOne" by Felix von Oertzen and "IBM z/OS" by Tobias Kuffert. If interested in more special features of the newer mainframe architecture, consider reading "IBM Z's Artificial Intelligence Stack" by Robert Nolting.

In their article "Porting GCC" Ulrich Weigand and Hartmut Penner have summarized the process of porting GCC to support S/390X [6]. It is worth pointing out that Mr. Weigand, along with other IBM employees, has visited Hasso Plattner Institute in March 2024. During that time, I managed to gain special insights into the process of porting GCC and LLVM to S/390X.

5 Conclusion

To conclude, I have found that for its age, S/390X is a suprisingly usable architecture. Especially by providing modern features, like 64-bit registers and floating-point support, S/390X can still be considered a competitive architecture from a compiler and assembler-programmer viewpoint. Since Linux and most of its coreutils have been ported to S/390X, the platform is also viable for modern cloud- and server applications.

References

[1] C. Boyer. “The 360 Revolution”. In: (Apr. 2004).

[2] J. R. Ehrman. *Assembler Language Programming for IBM System Z TM Servers*. 2nd edition. 555 Bailey Avenue San Jose, CA 95141: IBM Silicon Valley Lab, Feb. 2016.

[3] S. Groh and M. Müller. *The Conversion of Source Code to Machine Code. Explaining the Basics of Compiler Construction Using a Self-Made Compiler*. Wuppertal, Germany, May 2023.

[4] *IBM zSystem. The world's most powerful mainframes continue to push technological boundaries while powering global finance and online commerce*. Sept. 2024.

[5] *Mainframe strength: Continuing compatibility*. z/OS basic skills information center., Mar. 2024.

[6] H. Penner and U. Weigand. “Porting GCC to the IBM S/390 platform”. In: (Sept. 2019).

[7] K. Stine. “IBM z16 Technical Overview”. In: (2022).

[8] *The IBM System/360. The 5-billion-dollar gamble that changed the trajectory of IBM*. Sept. 2024.

[9] *The IBM System/370. It introduced computers to the Silicon Age*. Sept. 2024.

[10] *The IBM System/390. IBM debuted a mainframe for an internet world*. Sept. 2024.