CS2002 Week 5 Practical

Architecture: Assembly and Compiler Optimisation

22nd February 2021

Matriculation Number: 200002548

Tutor: Marco Caminati

Contents

1 Introduction

The aim of this practical was to investigate and understand the assembly language of both the x86-64 (in AT&T / GNU Assembler Syntax) and ARMv8 (AArch64) architectures. We were provided with a single C source file containing a binary search implementation as well as three assembly files containing optimised and unoptimised x86-64 and ARMv8 assembly code. We were to comment on each assembly instruction in these files, describing the purpose of each instruction, its interaction with registers and memory (stack) as well as the how the assembly relates back to the C program. Further comments were to be included in the report to determine the structure of stack frames and storage of local variables, as well as recursion to iteration optimisations and discussion about the ARMv8 architecture.

2 Analysis of Unoptimised x86 Assembly

Firstly, a stack frame is comprised of the base pointer for the caller, this is needed so that when the callee returns, the base pointer can be restored to bottom of the callers stack frame. In x86-64 the base pointer is pushed onto the stack with the `pushq %rbp` instruction and popped with `popq %rbp`. If a callee has more than 6 variables, the values need to be pushed in reverse order onto the callers stack frame. Then these variables can be accessed by the callee by taking a negative offset to access these variables within the callers stack frame. When the callee is initially called and the base pointer for the caller has been pushed onto the new stack frame the stack pointer is set to the value of the base pointer. Then the stack pointer is decremented by a multiple of 16 (since a stack frame is 16 byte aligned) depending on the local variable storage needed by the callee. This is done using the instruction `subq $48 %rsp` to reserve a 48 byte sized piece of memory in x86-64. After this arguments from the caller held in registers are allocated to the top of the stack at an offset from the base pointer. For example the instruction `movq %rdi -16(%rbp)` stores the first parameter (stored in `rdi` due to calling convension) to an offset of 16 bytes from the base pointer in x86-64. Local variables used within the callee are also created and allocated at an offset from the base pointer. For example `movq $2 -16(%rbp)` stores the value 2 at an offset of 16 bytes from the base pointer. Before returning from the callee and popping the callers base pointer the portion of the stack holding these local variables and arguments is freed by resetting the value of the stack pointer. This is done by adding to the stack pointer, for example freeing a 48 byte block of storage can be done using the x86-64 instruction `addq $48, rsp`. Note that the stack as a whole grows downwards in memory and so stack frames grow downwards into free memory. A return pointer is also present within the stack frame between the functions parameters and the saved base pointer, this return pointer is popped from the stack and put in rip (instruction pointer) when returning from the callee.

ADDRESS	CONTENT	SIZE
4(%rbp)	return address	4 bytes
(%rbp)	saved rbp	n/a
-1(%rbp)	return bool	1 byte
-8(%rbp)	unused	7 bytes
-16(%rbp)	long* array	8 bytes
-24(%rbp)	long elem	8 bytes
-28(%rbp)	int left	4 bytes
-32(%rbp)	int right	4 bytes
-36(%rbp)	int mid (temp.)	4 bytes
-40(%rbp)	int left (temp.)	4 bytes
-44(%rbp)	return bool (temp.)	4 bytes
-48(%rbp)	unused	4 bytes

There are two unique unused blocks of memory on the stack frame. These are in-between the return bool and the array pointer and at the end of the stack. The unused space at the end of the stack arises since it is part of the x86-64 convention that the stack is 16-bit aligned.

3 Analysis of Optimised x86 Assembly

The optimised assembly has turned recursion into iteration. Therefore we do not have any recursive calls and will only get to a stack depth of one. Instead the optimised code jumps from .LBB0_3 to the top of .LBB0_3 if `mid > left` and jumps from .LBB0_3 to .LBB0_7 if `elem < array[mid]`. This produces the same recursive effect as the unoptimised assembly.

The optimised assembly is much more efficient, especially when it comes to dealing with arguments. In the optimised code arguments were copied from the registers specified in the x86-64 calling convention to a position on the stack frame where they would then be manipulated. Furthermore local variables were also put on the stack frame, this made the assembly code very inefficient since these values all had to be copied over to the stack frame. Within the optimised x86 assembly the stack is never used, with all local variable and argument values stored in registers. This is more efficient because slow memory does not need to be accessed and values synchronised with registers.

Furthermore, `shr` is used to divide the number by 2. This is a correct way of dividing by 2 because shifting to the right causes each digit to move down by a power of 2, as a result dividing each digit and therefore the overall value by 2. This is more efficient because the calling convention for `idiv` requires that the divisor and the dividend be stored in `rax` and `rdx`, and that the result be stored in these registers as the quotient and remainder. This causes more move operations to be performed in order to comply with this calling convention and use the result afterwards. Shifting right using `shr` has no calling convention like this and so no move operations are performed.

4 Analysis of ARMv8 Assembly

The various instruction names in ARMv8 and their usage mostly resemble the x86 instruction set. ARMv8 however doesnt have separate instructions that do the same operation with different sized values. For example in x86 `movl` and `moveq` move a 32-bit and 64-bit values respectively, whereas in ARMv8 only mov is used.

ARM uses several addressing modes:

- Base register - `ldr w0, [x1]`
  Load data from the memory address held in `x1` and store in `w0`.
- Offset - `ldr w0, [x1, #4]`
  Load data from the memory address plus an immediate value offset (address `x1`+4) and store in `w0`
- Pre-indexed - `ldr w0, [x1, #4]!`
  Similar to offset addressing, however the pointer `x1` is updated either before the the fetching the value from the memory address `x1`+4.
- Post-indexed - `ldr w0, [x1], #4`
  Similar to offset addressing, however the pointer `x1` is updated either after the the fetching the value from the memory address `x1`+4.

ARM has different register names for storing values of different sizes. In ARMv8 you cannot get registers with a size of less than 32 bits. The prefixes for different sized registers are shown below:

- `W` 32 bits
- `X` 64 bits

ARMv8 features 31 general purpose registers. These are labeled `r0-r30`. The most important registers for calling convention is detailed below:

- `r30` - The link register (holds function return address)
- `r29` - The frame pointer (points to the base of the stack frame)
- `r19`...`r28` Callee-saved registers
- `r9`...`r15` Temporary registers (hold temporary values)
- `r0-r7` Parameter/result registers (hold arguments passed into function and returns results from function)

With regards to stack frames, ARMv8 stores the frame pointer and the link register on the stack at an offset of 16 bytes from the stack pointer using `stp x29, x30, [sp, #-16]!`. The frame pointer is then set to the stack pointer using `mov x29, sp`. This is very similar to x86-64 with the only difference being the return address held in the link register also being included on the stack.

Then when it comes to return from the stack the frame pointer and link register are restored using `ldp x29, x30, [sp], #16`. The `ret` instruction is then executed which returns from the subroutine, branching to the value held in the link register (`x30`).

Division by 2 in ARMv8 is performed much like optimised x86-64. That is, by using the `asr` instruction to shift the whole binary number to the right once so that it becomes twice as small.

References

[1] Odzhan, *A Guide to ARM64 /AArch64 Assembly on Linux with Shellcodes and Cryptography*, 30-10-2018, accessed 21-02-2021,
`https://modexp.wordpress.com/2018/10/30/arm64-assembly`

[2] University of Washington's School of Computer Science, *ARMv8 A64 Quick Reference*, 07-01-2018, accessed 21-02-2021,
`https://courses.cs.washington.edu/courses/cse469/18wi/Materials/arm64.pdf`

[3] ARM, *Procedure Call Standard for the Arm 64-bit Architecture (AArch64)*, 21-12-2020, accessed 21-02-2021,
`https://github.com/ARM-software/abi-aa/releases/download/2020Q4/aapcs64.pdf`

[4] Leo, *ARM Assembler - How do I use CMP, BLT and BGT?*, 15-05-2012, accessed 21-02-2021,
`https://stackoverflow.com/a/10601851/7711225`

[5] Azeria Labs, *ARM Data Types and Registers Part 2*, n.a., accessed 21-02-2021,
`https://azeria-labs.com/arm-data-types-and-registers-part-2/`

[6] ARM, *Registers in AArch64 -General Purpose Registers*, 03-07-2020, accessed 21-02-2021,
`https://developer.arm.com/documentation/102374/0101/`

[7] Jeremy, *Why might one use the xzr register instead of the literal 0 on ARMv8?*, 14-03-2017, accessed 21-02-2021,
`https://stackoverflow.com/a/42794729/7711225`

[8] phuclv, *Assembly registers in 64-bit architecture* , 17-12-2013, accessed 21-02-2021,
`https://stackoverflow.com/a/20637866/7711225`