Unoptimized x 86 Assembly

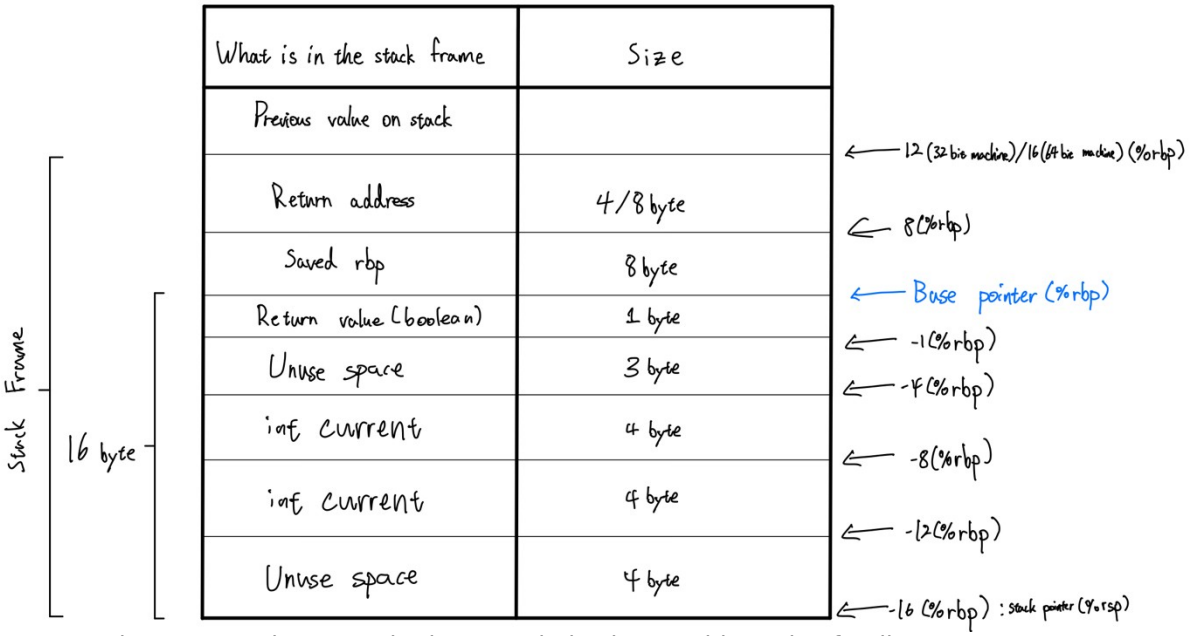

Figure 1: Stack Frames in the Unoptimised Assembly Code of Collatz_Recurse

Figure 1 represents the stack frame of collatz_recurse in the unoptimised assembly code. What the stack stored is shown in the left column and the size is shown in the right column. The return Boolean is stored at -1(%rbp) and has a size of 1 byte, while the int argument ‘current’ is stored at both -8(%rbp) and -12(%rbp) and has a size of 4 byte since for the one at -8(%rbp) it is stored in a 32 bit register %edi, which is use for passing the first argument (int current) to functions, and the ‘current’ at -12(%rbp) is stored in another 32 bit register %eax, which is use as an accumulator.

The return address is store in the current stack. The 64-bit values is used for the array (%rax) and we also extends the internet sign bit of the 32-bit array_length to use it as the index for the 64-bit array (line 90). Moreover, when we convert the signed long in %eax to a signed double in %edx:%eax by extending the most significant bit, the sign bit, go %eax into all bits of %edx (sign extend) for the preparation of division, we combined two 32-bit registers (%edx and %eax) to form a 64-bit value. We stored our variable to 32-bit register such as %eax and %edi and use it throughout the code. Lasty, for the 8-bit values, we only used it in the register %al, which store 1 or 0 as the return value which will be sign extend to %eax at the end of the code to gives the return boolean.

The reasons why there are some bytes unused in the stack is because the stack pointer can only be push down of a multiple of 16, since at line 13 the alignment is given as p2align 4, which means that the alignment must be a multiple of 16 bytes. This mean that the smallest amount we can push is -16 bytes, and since we only use 9 bytes for the collatz_recurse, we are left we 7 bytes of unused space.

The unused space between -1(%rbp) and -4%(rbp) is there because we first pushed a 1 byte return boolean on to the stack and we then pushed a 4 byte ‘current’ on to the stack. Since current is for byte, it must be stored at an address that is a multiple of four. Current can’t be store at -4(%rbp) since it’ll overlap with the return boolean, so we then store it at -8%(%rbp), which causes the 3 bytes unused space. The 4 bytes unused space between -12(%rbp) and -16(rbp) is caused by the fact that we do not have enough variables to push onto the stack.

Optimised x86 Assembly

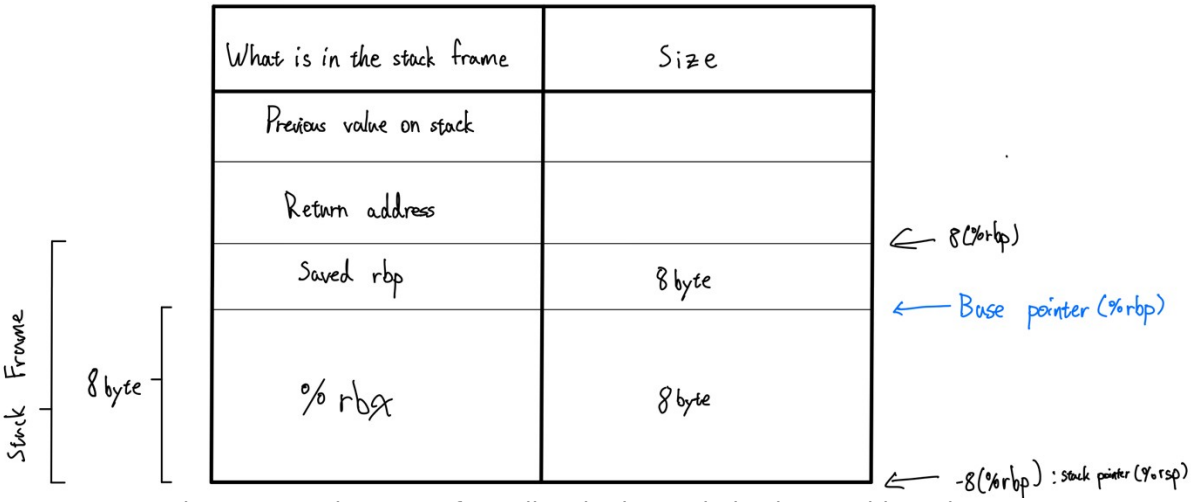

Figure 2. Stack Frames for Collatz in the Optimised assembly code

Like the stack frames for collatz_recurse in part 1, the return address is pushed on to the stack and then we pushed the base pointer to the stack (%rbp) as shown in Figure 2. Unlike in Figure 1, we then pushed %rbx (64-bit callee saved register) on to the stack which will hold array_length. This causes us to push the stack pointer out of the 16 bytes alignment since now the offset from the previous stack stack pointer is set to 24 bytes. However, this will be solved later on in the code where we manually set the %rsp to a offset of 8 from the previous stack’s %rsp(line72). This means that when return address is pop at the very end the offset of %rsp will go back to 0, which will match the 16 bytes alignment. We saved the value for array_length on the stack since it is a static variable and we need to keep track of it.

For the optimised assembly code, all the calculation and reassignment are handle via the registers and are not pushed onto the stack (no copies of them on the stack). For example, current is no longer stored on a stack, instead we change the value of that store in the 32-bit register %edi over the course of iteration (we no longer call the collatz_recurse in the assembly code), which will be the new “current”, which means the 1st argument that will be pass to the collatz_recurse in the c program.

We use the 64-bit values which we store in the 64-but registers such as %rbx, the static int array_length. The 32-bit values we’ve used are store in 32-bit registers such as %r10d and %ebx. We use the 8-bit value in the same way as

part 1, which is stored in %al, and it is either 1 or 0 represent the boolean value that the functions will return.

The optimised assembly code is iterative since we no longer store the variables and call the collatz_recurse, instead we update the value for %edi and uses the label and jumps to go back to the top of the collatz_recurse function in the c file. For example, in line 41-45 it check that if current%div == 0 , then if it fits the condition it jumps to .LBB0_2 (label 2) which set the %edi to current/div, and since there's no jump instruction at the end of label 2 it automatically moved to the next label .LLB0_3 (label 3). In label 3, it loops back to the top of the collatz_recurse function in the collatz.c, which do the first conditional check with array_length against MAXARRAYSIZE. Every time there is a recursive call in callatz_recurse function in the c file the optimised assembly code will update the value of %edi and loop back to the top of collatz_recurse function by jumping to label 3.

Instructions

Most instructions were similar to their x86 counterpart; 'mov', 'cmp', 'sub', 'add', 'and', etc, all do the same thing. The difference is that ARM instructions appear to take in 3 operands, the first being the destination register to save the calculated value to, and the second and third operands being registers or values that undergo the calculation.

Jumps use a different name and syntax. In ARM they are simply called 'b', for branch. They are then followed by the comparison of choice, which ARM provides as a separate set of instructions that can be used not just for conditional jumps, but conditional instructions as well.

For example, 'b.lt' is a 'if less than' jump, and 'b.eq' is a 'if equals' jump. These, like in x86, check the appropriate flags set by a compare or other appropriate instruction to determine an operands relationship to another operand.

Moreover, there exists a zero register (wzr) which when copied into a register, it sets the value store in the register to zero.

Furthermore there are some instruction that is quite different for example:

adrp x8, array_length

which set up the relative address for x8 as array_length

and

sdiv w8,w0,w9

Which is use for getting the result of current (w0)/ div(w9) and disregard the remainder then saved into register w8

Because it disregards the remainder, we do not need to extend to two 32-bit registers with one hold the quotient and one hold the remainder. However, this means we must calculate the remainder with the use of **msub**. For example:

msub w9, w8, w9, w0 means x9 = w0-(w8*w9)

Lastly, **bl** collatz_recurse means that it loads the collatz_recursen function address and branch to it, so is like calling the collatz_recursen in x86.

Addressing Mode

Indirect addressing mode

ARM supports a memory-addressing mode where the effective address of an operand is computed by adding the content of a register and a literal offset coded into load/store instruction.

```
str  w0, [x10, x9, lsl #2]
```

Where we take x10 than added x9*4 (2^2)

Pre-indexed Addressing Mode

In this addressing mode a pointer register is used to hold the base address. An offset can be added to achieve the effective address.

For example, in line 67 : stp x29, x30, [sp, #-16]!
[sp, #-16]!' means that -16 bytes below the current stack pointer (sp), and the exclamation means sp current value is also updated to be sp - 16. The stack pointer holds an address and this address is incremented by a given displacement, to provide a new address, so in this case so is the pointer register that hold the base address and -16 is the offset.

Post-Indexing Addressing Mode

This addressing mode accesses the operand at the location pointed by the base register, then increments the base register. This is similar to the pre-indexed mode, but the adjustment of the pointer happens after accessing the operand.

For example in line 109:

```
ldp  x29, x30, [sp], #16
```

We load the frame pointer and link register values from previous stack frame located at sp and back into respective registers, then increment sp by 16 bytes.This shows that we access the operant at the sp first then we move the so with the given offset which is 16.

Register to Register Addressing Mode

This is a direct addressing mode, where all the operands are registers.

For example in line 104 where we do

mov w0, w8

w0 and w8 are both register and we move the value in w8 into w0.

Registers

The naming convention for registers in ARM is much nicer, simply beginning with ‘x’ followed by the number of a register, which ranges from 0-31. The ‘x’ refers to the full 64-bit register, but it is possible to access the lower 32-bits of each register by using ‘w’ instead of ‘x’.
Different registers have different purposes however, and this does make it a little annoying to remember what number does what when %rbp and %rsp are quite aptly named. The ones in use in
The given ARMv8 file are:

x0 - x7 : the argument and return registers, with the arguments stored in order left to right starting from x0.

x8 - x18 : temporary, general-purpose registers

x29: the stack pointer register which stores the address of the bottom of the stack frame

x30: the link register

The link register is something different from x86. Its purpose is to save the address to return to once a subroutine has been finished.

Stack Frames

For ARMv8 assembly the base pointer and stack pointer are store in the bottom of the current stack instead of the bottom of the previous stack frame

In the ARMv8 assembly code, ‘stp’ is used to store the stack pointer of the previous stack and the value in the link register to the current stack. Then right before the function returns, ‘ldp’ is used to

restore the saved values back into their respective registers to return to the state the stack was on before the current stack frame was made.

Optimisation Level Difference

In level 1, no values from the registers are saved in the stack, apart from the stack and base pointers used to manage the stack frame. This is a perfect midway between level 0 optimisation where all argument registers had their values copied to the stack frame, and level 2 optimisation where there was no stack frame. This is shown in part 2, where there is no stack frame for collatz_recurse, so instead we explains the stack frame for collatz.

Conclusion

In conclusion, I fulfilled all the requirements on the specification. I commented all three assembly files, and analysed the stack frames and the assembly code thoroughly. It took me more time to analyse how the second x86 file iterate through the collatz recurse than understanding ARMv8, because I found it hard to find how the assembly code loop through the collatz_recurse function at the beginning. If I have more time, I would like to find a way to optimise the ARMv8 assembly code so it will be iterative.

Reference

Used for X86:

Brown University. *X64 Cheat Sheet*. https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

Contributors to Wikimedia. “X86 Assembly/x86 Architecture.” *Wikibooks, Open Books for an Open World*, Wikimedia Foundation, Inc., 5 Nov. 2021, https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture

AMD. “X86-64 Architecture Guide.” *X86-64 Architecture Guide*,http://6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html

Used for callee saved register:

nicoabienicoabie 2. “What Are Callee and Caller Saved Registers?” Edited by Peter CordesPeter Cordes 275k4141 gold badges497497 silver badges700700 bronze badges, *Stack Overflow*, 28 Apr. 2013, https://stackoverflow.com/questions/9268586/what-are-callee-and-caller-saved-registers

Used for xorl:

cnicutarcnicutar. “Xorl %EAX - Instruction Set Architecture in IA-32.” Edited by Community Bot, *Stack Overflow*, 10 May 2014, https://stackoverflow.com/questions/23575787/xorl-eax-instruction-set-architecture-in-ia-32

Used for idiv:

Unkown. “X86-Assembly/Instructions/Idiv.” *Aldeid*, 28 Nov. 2015, https://www.aldeid.com/wiki/X86-assembly/Instructions/idiv

Used for cltd:

MichaelMichael. “Trying to Understand the Assembly Instruction: CLTD on x86.” *Stack Overflow*, 2013, https://stackoverflow.com/questions/17170388/trying-to-understand-the-assembly-instruction-cltd-on-x86

Used for ARMv8:

“A Guide to ARM64 / aarch64 Assembly on Linux with Shellcodes and Cryptography.” *Modexp*, 9 Apr. 2019, https://modexp.wordpress.com/2018/10/30/arm64-assembly/

ARMv8 A64 Quick Reference Conditional Instructions. 2018, https://courses.cs.washington.edu/courses/cse469/18wi/Materials/arm64.pdf

Used for ldrsw:

Arm Developer. “LDRSW (Register).” *ARM Compiler Armasm User Guide Version 6.6*, https://www.aldeid.com/wiki/X86-assembly/Instructions/idivdeveloper.arm.com/documentation/dui0801/g/A64-Data-Transfer-Instructions/LDRSW--register-.