1 – Analysing Unoptimised Assembly Code

Address	Purpose	Value	Size
rbp	saved base-pointer	rbp	64-bit
rbp - 1	return value	0 or 1	8-bit
rbp - 8	current	edi	32-bit
rbp - 12	current	eax	32-bit
rbp - 16	unused padding for 16-bit alignment	0	32-bit
rax + rcx * 4	“current” values in array	edx	32-bit
r8	array_length	ebx	

With no optimisation, I found that the load-store addressing mode was used much more often. For each register which requires a value later, a new stack position is used to store and restore that value. For example, the addresses (rbp – 8) and (rbp – 12) both store a value of current at that point to be restored later, instead of using the same address which would be possible as eax gets the current value from edi once it is finished doing it’s calculations.

2 – Recursion to Iteration

Address	Purpose	Value	Size
rbp	saved base-pointer	rbp	64-bit
rbp - 8	saved rbx	rbx	64-bit
r9 + rbx * 4	“current” values in array	edi	32-bit
r8	array_length	ebx	32-bit

For all optimisation levels, more stack space was required than the red-zone, as space were required to store the array and it’s values in the stack.

With the higher optimisation, the biggest difference in the stack creation is the usage of load-store addressing modes was limited as much as possible. This is because register-register addressing is much faster. This can be seen as the temporary current values are stored in edi and edx where one stores the current value whilst the other does calculations on it.

The function collatz_recurse function does not appear with this optimisation because the current value for collatz_recurse is set using the start value in collatz, and so the compiler just uses the initial argument and changes that using the two loops of current * in_mult + in_add and current / in_div without needing to set or pass through variables to a second function.

I also noticed that there are certain commands which are skipped because they have no affect on the code. On line 136 and 140 of collatz0-x86-commented.s, there is a command “andb $1, %al” which does an “and” logical operation on %al, which means that %al is 1 when 1 & %al are 1, and 0 with any

other values. As 1 is always 1, %al = 1 when %al =1 which means that %al = %al. This is not included in the optimised version.

Also, on line 141, the command “movzbl %al, %eax” is skipped in the optimised version because the register %al is just the first 8-bits of the 32-bit register %eax. Setting the 32-bit register to the 8-bit register will just mean that %eax = %eax.

3 – ARMv8

Command	Meaning	Addressing Mode
stp x29, x30, [sp, #-16]!	Stores values x29 and x30 into the stack space in space between stack pointer (sp) and sp – 16.	Relative
mov x29, sp	Stores the second value sp into the address x29	Register
adrp x8, array_length	Gets the label address (not the value) of array_length and stores it in x8	Relative
ldrsw x9, [x8, :l012:array_length]	Loads the signed word value of array_length by calculating the address and then gets the value, storing in x9	Register
cmp w9, #199	Subtracts 199 from w9 and stores flags without affecting w9	Immediate
b.le .LBB1_2	Checks the flags for signed less-than or equal to 0 and branches (jumps) to .LBB1_2	Relative
add w11, w9, #1	Adds w9 and the number 1 to w11, storing in w11	Immediate
str w0 , [x10, x9, lsl #2]	Stores the value w0 in the stack at x10 + x9 * 2	Immediate
sdiv		

In ARM, the destination and source is often swapped compared to x86, where the destination comes first.

The calling convention can be seen in the storing of x29 and x30 from the stack which correspond to the frame-pointer base-pointer and link-register which are stored at the beginning of the function.

The different addressing modes used were register, which uses the address values stored in a register (such as the ldrsw above); immediate, which use fixed number values (such as the cmp above) without needing to look in any registers; and relative (such as stp above) where the address is calculated using a mix of register value, as well as an offset from that register address.