CS2002 Practical 2 - x86 Assembler

By: 150013565 Tutor: Juliana Bowles

Date: 27th February 2018

Abstract

The objective of this practical was to understand the assembly language of x86-64, in the AT&T syntax (also known as GNU Assembler, or GAS). This included adding comments to some assembly language and using inline assembler in a C program to let it print stack frames.

The practical was split into three parts: commenting a function in an assembly file, use inline assembler in C to confirm the analysis made on the aforementioned function and then analyze what optimisation flags do to the assembly file to make the program run faster.

1 Stage 1: Commenting Assembly Code

The first stage of the practical was to find and comment the function *sortsub* in an assembly file. All lines bar assembly directives were commented.

The original C code for the *sortsub* function takes in five arguments; arr, size, left, right and comp and recursively calls itself to sort the left and right sides of an array and lastly merges the two sorted sides together.

In assembly, the code for the function appears as a label 'sortsub:' followed by some assembly directives and a push statement. This push statement sets up a new stack frame saves registers *rbp* onto the stack. This *rbp* register represents the *sortsub* function. The stack pointer then points to that register and the stack grows 64 bytes downwards. This allows the program to store 64 bytes worth of data onto the stack. The five arguments from the function are then pushed onto the stack into positions relative to *rbp*. For example, the argument 'arr' is stored -8 bytes from the stack pointer. Once all variables were allocated onto the stack, the variables must be passed to other function calls. In this example, the variables were passed into functions *sortsub* and *merge*. Function calls in assembly were done simply by calling the label which represented the name of the function. Once all of this is done, the function then gives back 64 bytes of the stack space and pops all the elements in the stack so that it is empty. Once empty, the stack pointer now points to the address which represents the top of the stack.

I encountered many difficulties with this section of the practical. One of the difficulties I faced was understanding the flow of data within assembly. Since assembly isn't really human readable, it is hard to keep track of each variable. There is no real visual intuition as to what each component is and what it represents, everything is just a register and locations relative to the stack pointer. Another problem I faced was understanding why assembly seemingly makes redundant calls. For example, on line 77 of *sort-comments.s* there is a move statement which moves the contents of register *rax* into *rdx*. This is unnecessary as it provides nothing to the program. But upon further research, it appears that it does this to avoid running into undefined behaviour.

2 Stage 2: Stack Frames

The second stage of the practical was to use inline assembler to print stack frames corresponding to the *sortsub* function to confirm analysis made in stage 1.

The vast majority of results obtained from my function correspond to my analysis made in Part 1. The locations at -8, -16, -24, -32, -40, -48 and -56 relative to the base pointer of the stack correspond to the arguments of the function. This can be verified using the GNU debugger - GDB. By adding a breakpoint at *sortsub*, individual stack frames for *sortsub* can be looked at and analysed, and indeed, those locations and values match up with what GDB outputs.

The value at location 0 relative to the base pointer points to where the function call came from. It is not abundantly clear in the assembly file how it does this but when printing the first and second stack frame, the value at location 0 relative to the base pointer, refers to the base pointer of the second and third stack frame.

The value at location -64 relative to the base pointer is one that I cannot explain. In the three frames printed, the value at that location is 0. I am not sure what it represents. My guess would be that it is there as a place holder since the stack frame has to be divisible by 16 and there aren't enough instructions in the stack frame to make it divisible by 16 so it adds this to add space.

The value at location -72 relative to the base pointer is another result I cannot explain. At this location it prints out a pointer to somewhere and it is not clear where this is a pointer to. My guess would be that this is the address of the top of the stack.

With the addition of the if condition found in *sort-plus.c*, the layout of the stack frame has not changed. The function still uses the same number of variables, the only difference is that there are now pointers to the *print_stack* function.

3 Stage 3: Other Functions and Optimisations

3.1 Doublewords, Singlewords and Bytes

In the stack frame of *sortsub*, all relevant values are quadwords. This is partly because all variables in *sortsub* are longs. This is realised in assembly by the instruction *movq* where the suffix 'q' represents a quadword. However, if other types like doublewords, single words or bytes, it would be realised in assembly as *movl*, *movw* and *movb* respectively. In other functions like *mergesort*, doublewords are used. In *mergesort* the doubleword is used to zero a register, this is done since the first call of sortsub has the variable 'left' assigned to zero, representing the first column in the array. This is not the most space-efficient method of doing this as zero does not need 32-bits to be represented. In *merge* the byte is used to know when to break out of the while loop.

3.2 Calling 'comp' in Assembly

On line 165 on *sort0.s*, the instruction *callq %rax* is used to call *comp* in assembly. This instruction is a call to the address held in register rax. The value held in register *rax* is an address and the star is used to reference that value as an address so that callq can go to it. This address corresponds to the static method *comp_long* found in *main.c*.

3.3 Assembly 'while' Loop

The while loop is implemented through an accumulator register *%al*, this register contains a byte which determine if the loop should proceed or not. It does this by checking the condition i.e. p1 ¡ mid or p2 ¡ right using the *compq* instruction. The value inside this register is tested using assembly instruction *testb*. If one of the conditions are true, the next instruction is to jump to a label which contains the instructions of the loop. If either are false, the next instruction is to jump to a label which executes instructions outside the loop.

3.4 Optimisations

3.4.1 Sort1

The biggest difference between *sort1.s* and *sort0.s* is that the stack still contains the arguments for the method *mergesort*. Thus, *sortsub* can obtain the values required for its calculations by looking up the stack and taking the values rather than moving values from register to register. This saves time as I/O is a costly operation. Once the function has finished its calculation, it pops the registers it used for the calculation and not the original registers that *mergesort* used. This allows the *merge* function to run quicker as it can also refer back to the values in the stack that *mergesort* produced.

Another difference between *sort1.s* and *sort0.s* is how division of two is implemented. In *sort0.s*, division is implemented using *idivq* whereas in *sort1.s* it is implemented using *shrq*. The difference between the two instructions is that *shrq* implements division of two by shifting the binary digits to the right. For example, the number 8 in binary is 1000, by shifting each binary digit one place to the right, the digit becomes 0100 which is 4.

3.4.2 Sort2

On top of the space and speed optimisation found in *sort1.s*, *sort2.s* does not have an assembly definition for the method *merge*. The advantage of doing this is that there is no need to pop off elements in the stack. All elements used by *merge* are already on the stack, it can just access the arguments simply by utilizing the relative register it requires. This is a quicker operation since there is no need to move values from one register to another, the values are already there.

3.4.3 Sort3

There are no differences between *sort2.s* and *sort3.s*, all possible optimisations have already been made.

3.4.4 Sort-Space

This optimisation function optimises for space. I have added another line to the Makefile which uses this optimisation flag. When compared to *sort2.s*, *sort-space.s* saves space by removing the 16-byte boundary between stack frames. This has been removed a total 5 times when compared to *sort2.s*, thus saving a total of 80 bytes of space.

4 Extension: MIPS

MIPS is an example of a RISC based architecture whereas x86 is a CISC based architecture. Comparing the length of *sort0.s* and *sort-mips.s*, the latter is much longer than the former. This is part of the philosophy behind RISC. RISC processors are designed to have multiple instructions, whereas CISC processors are designed to do multiple things in a single instruction.

In relation to the produced assembler in MIPS and x86, the reference registers in different ways. For MIPS, it references different registers with a $ whereas in x86 it references them with %.

Another difference in the produced assembler is that MIPS can have three registers in one instruction whereas in x86 there are only ever two registers. The three registers in MIPS are wrote as follows *addu $2, $2, $3* which store the result of an unsigned integer addition between registers $2 and *$3* and stores the result in register *$2*. The first *$2* is the register to store the result.

A final difference is the variable name for the base pointer of the stack-frame. In MIPS, the frame base pointer is referenced as *$fp* whereas in x86 it is referenced as *%rbp*.

5 Conclusion

Overall this was a very interesting practical. This provided a real insight into the inner workings of how the CPU works to run the C code we write. At first glance, assembly is quite tricky to understand but once you are familiar the syntax and can picture the variables moving around in registers, the more understandable it becomes. The optimisation flags used for this practical was fascinating. For *sort2.s*, *sort3.s* and *sort-space.s*, an entire method was omitted from the assembly code in favour jumping to local labels.

Given more time, I would have liked to compile the C code to other architectures such as i386, AMDIL and nVidiaPTX which are the Intel, AMD and Nvidia assembly languages respectively but I ran out of time to implement and document it. Another interesting extension would be to further understand what each optimisation flag does i.e. what each flag in the optimisation flag does and how to effectively use this information to write better, faster and more efficient code.