TDT4205 Kompilatorteknikk - Problem Set 4 Part 1

Einar Johan Trøan Sømåen
MTDT

Task 1.1.1:
A stack frame is simply an area in memory defined by the stack-top-pointer (ESP) and the stack-bottom-pointer (EBP), the current scope is defined to be in this area, and any local variables exist in this region of memory. (Although they might point to the heap). If you call a function, that function sets up it’s own stack-frame, pushing your old EBP to the stack, and popping it back when it returns, thus allowing it to have local variables that don’t interfer with the ones in the caller (and allowing the caller to continue working with IT’s stack-frame afterwards. This “stacking” (pardon the pun) of stack-frames, allows for recursion too, as the recursive call simply creates another stack frame.

The x86 has laughably few registers, you will indeed have good use of the stack while writing functions. (6 General Purpose Registers ought to be enough for anybody?).

All that aside, the main point of having a stack, is that locals have a location that we can know without hardcoding absolute memory-addresses in our code (which would in practice make all our variables static, and recursion more or less impossible). Since the stack allows us to find our variables relative to the stack-base at any point. (And arguments too)

Task 1.1.2:

OFFSET	Contents
16	y
12	x
8	a
4	old Program Counter
0	old EBP
-4	x1
-8	x2

Although, the actual Stackframe goes only from 0 to -8, there will of course be references to the arguments that must be placed in the relative positions on the stack before performing a call to the function.

Task 1.1.3:
Setting up the Stack Frame is a question of simply expecting the correct data to be placed on the stack when call is run, then we need to push the current base pointer (so that we can reproduce it again on exit), then we let the current top be the new bottom, then we push all local variables that need memory (unless they are variables that are optimized to be in registers for the duration of the function).

After the function is finished, we clean up by incrementing ESP back up to the location of the old EBP on the stack (typically 4 times the amount of locals), then we pop the old EBP back from the stack, and return from our function. (Thus putting back the Program Counter from the stack, and reproducing the stack-state that existed before we were called)

At this point, the calling function once again has it's stack-frame in place, and can continue doing it's business, typically it might like to clean out all the arguments it might have pushed for the call.

The registers outside of EBP/ESP are NOT preserved through the call, unless you follow the __fastcall calling convention, in which case arguments are in registers, and the stack is left untouched (aside from the program-counter which will be pushed and popped). EBP/ ESP are conserved in the fashion described above (EBP is pushed to the stack, and ESP is reproduced by poping that element and returning (which will put it at the exact state it was prior to calling)).

You MAY also preserve all registers, by doing the tedious task of pushing and popping all of them (in which case we should be glad we're talking x86, as it would be much more work and space on a sane architechture), and popping them before return, but then again, you have locals the stack in the calling functions, and they know that the registers will contain garbage after the call, so the convention is to simply not care about preserving registers.

Task 1.2:
See attached file, I decided against optimizing it any further, although as the comments note, I might have reduced the actual use of locals a bit by having i in a register. As a side note, it would be preferable if the delivered makefiles used the -g flags so debuggable executables would be the default.

I also tried to invert my conditionals in the same fashion I know compilers seem to use (thus placing the else-part of an if right after it, and the internals elsewhere, expecting if-s to be untrue).

Task 1.3.1:
Expecting VAR to be 4 bytes in size, the following would be the placement order, also expecting that they are pushed in order of appearance (I couldn’t find any spec for it):
a = ebp - 4
b = ebp - 8 , (and the last four elements on 12, 16, 20, 24)
c = ebp - 28

Task 1.3.2:
We need the lexical depth, to allow for determining our way out of nested scopes, whether they be nested functions, or simply c style scopes: “is the symbol we wanted in our current lexical level? if no, then check the one it is enclosed by, rince, lather repeat”. When using displays for lookup, we wouldn’t have any clue where to start (or indeed if we found the correct instance, without knowledge of the lexical depth of each variable.).