TDT4205 Compilers
Exercise 4

Stian Hvatum (hvatum)
MTDT

March 4, 2012

Contents

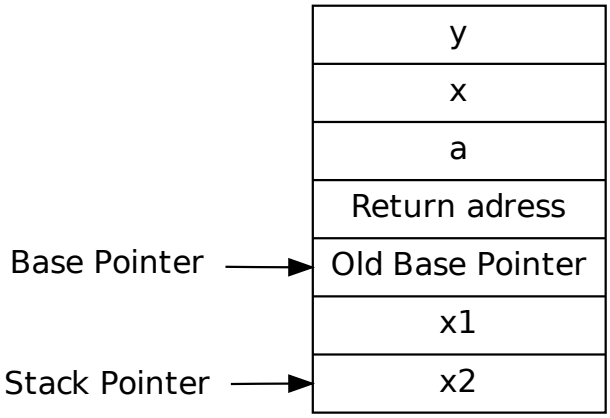
PART 1 Theory and Assembly Programming

Task 1.1 Stack Frames

1. What is a stack frame

A *stack frame* is a location in a programs logical memory, or more precicely, a location on the *stack*, where a function keeps its current local variables. The stack frame grows as local variables are added, and shrinks as they are poped of, eg. if they are not going to be used any more.

2. Stack frame illustration



Here we see a model of a stack frame at the top of the program stack. The stack grows downwards, so a new variable will be placed below x2. In that case, the stack pointer would be decreased accordingly. Each block will have a size of 4 bytes, given a 32-bit system.

3. Setting up and tearing down stack frames

To call a function, the caller will put the functions arguments on the top of its stack, and then call the function. This way, a function will always find its arguments right above its stack frame. After the function call, we need to increment the stack pointer to "clean" the stack of the call arguments. The top of the stack is pointed to by the $esp-register, and the stack base is pointed to by the $ebp-register.

```
1       /* Call printf(string, l_var_1) */
2       pushl   -4(%ebp)
3       pushl   $.STRING0
4       call    printf
5
6       /* Clean up on stack */
7       addl    $8, %esp
```

When a function is called it is itself responsible for setting up and tearing down its own stack frame. The function will do the following statements to set up the stack frame:

```
1   func:
2       /* Store old base pointer on top of stack */
3       pushl   %ebp
4
5       /* Set new stack base (ebp) to old top-of-stack (esp) */
6       movl    %esp, %ebp
```

Here we see that the function pushes the old base to the top of the stack, so that it know how to restore the previous stack frame on return. It also creates its own base by setting the stack base to point at the old top of stack. Now, base and top is at the same location, and we basically have our new stack frame, with base, stack pointer and function arguments. New local variables will be placed at the position below the stack pointer, and then the stack pointer is decremented with the size of the variable. Typically 4 bytes.

To return to the caller, the function will have to tear down its own stack. We state `leave` to tear down our own stack frame, and `ret` to return to the point where the current function was called. We see that in x86-assembly, the stack operations are supported by

```
1    /* Clean up stack frame */
2    leave
3
4    /* Return home */
5    ret
```

Now, we are back at the caller, who must remember to clean the stack of the arguments, as stated above. We must also notice that a function may alter the registers to its need, so before calling a function, we must store the registers that keeps usefull values, else the values may be lost.

Notes: `leave` is equivalent to[1]

```
1    mov %esp, %ebp
2    pop %ebp
```

and the stack-setup could also be done utilitizing the `enter`-command.

[1] http://en.wikipedia.org/wiki/X86_instruction_listings

Task 1.2 x86 Assembly Programming

The complete file *foo.s* is attached with the delivery of this file. This is the function I have implemented:

```
1  foo:
2      /* Store old base pointer on top of stack */
3      pushl   %ebp
4
5      /* Set new stack base (ebp) to old top-of-stack (esp) */
6      movl    %esp, %ebp
7
8      /* Store 0 in ecx  (loop starts at 1, but is incremented in first test) */
9      movl    $0,   %ecx
10
11     /* Store 0 on the stack, our sum value */
12     pushl   $0
13
14     /* And start loop-test */
15     jmp tst_lp
16
17 lbody:
18     /* Loop body */
19     /* Modulo = divide and check rest-register */
20
21
22     /* Check for input divisible by 3 */
23     movl    %ecx, %eax
24     movl    $3,   %ebx
25     cdq
26     idiv    %ebx
27     /* edx now contains ecx mod 3 */
28     cmp     $0,   %edx
29     jz tst_ok /* Test true */
30
31      /* Check for input divisible by 5 */
32     movl    %ecx, %eax
33     movl    $5,   %ebx
34     cdq
35     idiv    %ebx
36     /* edx now contains ebx mod 5 */
37     cmp     $0,   %edx
38     jnz tst_lp /* Test false */
39 tst_ok:
40     addl    %ecx, -4(%ebp)
41
42 tst_lp:
43     /* Get the function argument and store in ebx */
44     movl    8(%ebp), %ebx
45
46     /* Increment and test */
47     inc %ecx
48     /* if ebx < ecx => jump to start of loop */
49     cmp %ebx, %ecx
50     jl lbody
51
52     pushl -4(%ebp)
53     /* Print results */
54     /* sum is on top of stack */
55     pushl   $.STRING0
56     call    printf
57
58     /* Clean up on stack */
59     addl    $8, %esp
60     /* Clean up stack frame */
61     leave
62
63     /* Return home */
64     ret
```

Task 1.3 Symbol Tables

1. Stack offset

a -4

b -8

c -28

since previous started at -8 and offset is 5 times the size of 4 bytes.
Mathematically: $-8 - (4 \cdot 5) = -8 - 20 = -28$

2. Lexical depth

We need to know the lexical depth of a symbol in order to get to the right symbol in the current scope. I give you an example:

```
1  FUNC depth_printer()  {
2      VAR a;
3      a = 1;
4      {
5          PRINT a;
6          VAR a;
7          a = 2;
8          PRINT a;
9      }
10     PRINT a;
11 }
```

If we are to know inside the scope at line 4-9 which variable to print at first and last `PRINT`, we need to know what lexical depth we are in, and what variables are available to us. It may be more important at the last print (on line 10), since the inner variable `a = 2` is more recent than `a = 1`, and thus we would print 2 if we were not aware of lexical depth. If we just knew the offset on the stack frame, we would not know what variable to use in each print, as we would not be aware of scope, and would possibly print the wrong variable.