C + x86 Assembly : Machine-Level Representations of Programs

By Amanda Falke
September 2014
Abstractmachines at gmail

Questions which I try to answer from :
Bryant and O'Halloran's *"Computer Systems: A Programmer's Perspective"*

3.54: **Write C code for this prototype and assembly equivalent. Return %eax.**

int decode2 (int x, int y, int z);

ASSEMBLER:	**MY ANSWERS:**	
movl 12(%ebp), %edx	# puts y into edx -- gets y:	***int a = y;***
subl 16(%ebp), %edx	# this is edx - 16%ebp, or y - z:	***a = a - z;***
movl %edx, %eax	# copy y into new variable:	***int b = a;***
sall $31,%eax	# shift eax logic left: set sign bit:	***b <<= 31;***
sarl $31, %eax	# shift eax arith right: sign extend:	***b >>= 31;***
imull 8(%ebp),%edx	# multiply x (%ebx+8) by a (%edx):	***a = a * x;***
xorl %edx,%eax	# b = b XOR a:	***b ^= a;***
	# return %eax (b):	***return b;***

3.58: Fill in the missing parts of the C code. Watch out for cases that fall through. Register %edx corresponds to program variable result and is initialized to – 1.

Answers are in BOLD.

```
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D,
MODE_E} mode_t;
int switch3(int *p1, int *p2, mode_t action)
{
int result = 0;
switch(action) {
case MODE_A: // .L13
        int a = *p1; // edx
        int b = *p2; // eax
        int c = a; // ecx
        *c = b;
        // * FALL THROUGH:  jump to default .L19
case MODE_B: // .L14
        int a = *p2; // edx
        int b = *p1; // ecx
        a += b; // edx += ecx
        int c = *p2;
        c = a; // eax = edx copy
        // * FALL THROUGH:  jump to default .L19

case MODE_C:
        int a = *p2; // edx
        a = 15; // edx = 15
        int b = *p1; // ecx
        a = b; // edx = ecx copy
        // * FALL THROUGH:  jump to default .L19
case MODE_D:
        int a = *p1; // eax
        int b = *p2; // ecx
        b = a; // ecx = eax copy
        *p1 = 17; // edx ... and note that edx is return value....
        // * FALL THROUGH:  jump to default .L19
case MODE_E:
        *p1 = 17; // edx ... and note that edx is return value....
        // * FALL THROUGH:  jump to default .L19
default:
        c = *p1; // copy edx into eax
        result = c; //  set return value
}
return result;
}
```

Arguments: p1 at %ebp+8, p2 at %ebp+12, action at %ebp+16
Registers: result in %edx (initialized to -1)
The jump targets:

```
.L17: MODE_E
movl $17, %edx
jmp .L19

.L13: MODE_A
movl 8(%ebp), %eax
movl (%eax), %edx
movl 12(%ebp), %ecx
movl (%ecx), %eax
movl 8(%ebp), %ecx
movl %eax, (%ecx)
jmp .L19

.L14: MODE_B
movl 12(%ebp), %edx
movl (%edx), %eax
movl %eax, %edx
movl 8(%ebp), %ecx
addl (%ecx), %edx
movl 12(%ebp), %eax
movl %edx, (%eax)
jmp .L19

.L15: MODE_C
movl 12(%ebp), %edx
movl $15, (%edx)
movl 8(%ebp), %ecx
movl (%ecx), %edx
jmp .L19

.L16: MODE_D
movl 8(%ebp), %edx
movl (%edx), %eax
movl 12(%ebp), %ecx
movl %eax, (%ecx)
movl $17, %edx

.L19: default
movl %edx, %eax Set return value
```

3.59: **Answers are in BOLD.**

Fill in the body of the switch statement with C code that will have the same behavior as the machine code. Parameters x and n are loaded into registers %eax and %edx , respectively.

The jump table resides in a different area of memory. We can see from the indirect jump on line 9 that the jump table begins at address 0x80485d0 . Using the gdb debugger, we can examine the six 4-byte words of memory comprising the jump table with the command x/6w 0x80485d0.

gdb prints the following:

```
(gdb) x/6w 0x80485d0
0x80485d0: 0x08048438 0x08048448 0x08048438 0x0804843d // 0x32, 0x33, 0x34, 0x35
// Note that 1st and 3rd, or 0x32 and 0x34, are at same address, so same jump location!
0x80485e0: 0x08048442 0x08048445 // 0x36, 0x37
```

```
1 int switch_prob(int x, int n)
2 {
3 int result = x;
4
5 switch(n) {
6
7 /* Fill in code here */

 case 0x32:
 case 0x34:
 x <<= 2;
 break;

case 0x35:
x >>> = 2; // arith is 3 >>> like this
// OR:
x >> 2; // architecture dependent!
 break;

case 0x36:
x *= 3;
// FALL THROUGH TO NEXT
CASE!

case 0x37:
x *= x;

case 0x33:
default:
x += 10;
x = result;


8 }
9
10 return result;
11 }
```

```
1 08048420 <switch_prob>:

2 8048420: 55              push %ebp # push stack frame
3 8048421: 89 e5           mov %esp,%ebp # frame, stack ptrs
4 8048423: 8b 45 08        mov 0x8(%ebp),%eax # get x
5 8048426: 8b 55 0c        mov 0xc(%ebp),%edx # get n
6 8048429: 83 ea 32        sub $0x32,%edx
    #minimum case value is 50 because 0x32 = 50
7 804842c: 83 fa 05        cmp $0x5,%edx
    # max value is 0x37 because 0x32 + 0x5 = 0x37
8 804842f: 77 17               ja 8048448 <switch_prob+0x28>
     # ja jump ahead to 0x8048448 : the default case

9 8048431: ff 24 95 d0 85 04 08          jmp *0x80485d0(,%edx,4)
 # indirect jump to first case at 0x32
10 8048438: c1 e0 02                     shl $0x2,%eax
 # shift eax ( x ) logic left by 2
11 804843b: eb 0e                jmp 804844b <switch_prob+0x2b>
 #jump to pop of stack frame (this means a BREAK!)


12 804843d: c1 f8 02          sar $0x2,%eax
# 0x804843d is 0x35,
# so we shift eax (x) arithmetic right by  # 2
13 8048440: eb 09           jmp 804844b <switch_prob+0x2b>
 # jump to pop of stack frame (this means a BREAK!)

14 8048442: 8d 04 40          lea (%eax,%eax,2),%eax
# load effective address at 0x36, simple arithmetic
# of eax = eax *3, or eax = eax+eax+eax, or x *= 3;
#note the FALL THROUGH here, no BREAK!

15 8048445: 0f af c0          imul %eax,%eax
# 8048445 is 0x37, and we multiply eax ( x) by itself x *=x
16 8048448: 83 c0 0a         add $0xa,%eax
 # 0x8048448 is the default case 0x33, and we add 10 to eax #
hence, eax += 10, hence x += 10
17 804844b: 5d                pop %ebp
#pop stack frame off stack and prepare for return
18 804844c: c3                 ret
# return
```


3.66: **Answers are in BOLD.**

You are charged with maintaining a large C program, and you come across the following code:

```
typedef struct {
int left;
a_struct a[CNT];
int right;
} b_struct;

void test(int i, b_struct *bp)
{
int n = bp->left + bp->right;
a_struct *ap = &bp->a[i];
ap->x[ap->idx] = n;
}
```

```
00000000 <test>:
0: 55                          push %ebp
1: 89 e5                       mov %esp,%ebp
3: 8b 45 08                    mov 0x8(%ebp),%eax
 # get int left
6: 8b 4d 0c                    mov 0xc(%ebp),%ecx
 # get struct a
9: 8d 04 80                    lea (%eax,%eax,4),%eax
 # eax * 5: left *= 5;
c: 03 44 81 04                 add 0x4(%ecx,%eax,4),%eax
 # get int right

10: 8b 91 b8 00 00 00           mov 0xb8(%ecx),%edx
 # edx = ecx + 4 bits, or edx new var = a + 8 bits
16: 03 11                      add (%ecx),%edx
 # edx += ecx, or new var = var + a;
18: 89 54 81 08                 mov %edx,0x8(%ecx,%eax,4)
 # 4*eax + ecx is where edx+8 gets copied into,
# or

1c: 5d                         pop %ebp
1d: c3                         ret
```

The declarations of the compile-time constant CNT and the structure a_struct are in a file for which you do not have the necessary access privilege. Fortunately, you have a copy of the '.o' version of code, which you are able to disassemble with the objdump program, yielding the disassembly shown in Figure 3.45.

Using your reverse engineering skills, deduce the following.

A. The value of CNT.

FIVE

B. A complete declaration of structure a_struct. Assume that the only fields in this structure are idx and x.

K&R problem 5.5

Exercise 5-5. Write versions of the library functions strncpy, strncat, and strncmp, which operate on at most the first n characters of their argument strings. For example, strncpy(s,t,n) copies at most n characters of t to s. Full descriptions are in Appendix B.

// strncpy:

```
char *  strncpy( char * destination, char * source, size_t n)
{

  size_t i; // size_t is size of data type, of course!

  for ( i = 0; i < n && source[i] != ‘\0’, i++ ) // (1) until source char array hits n characters,
  {
          destination[i] = source[i];              // (2) copy each array element over

          for ( ; i < n; i++ )                 // (3) set remainder (beyond the n limit) to NULL bytes
          {
            destination[i] = ‘\0’;
           }
   } // for
   return destination;                              // (4) return final destination string

} // function ends
```

// strncat:

```
#include <string.h>

int  result;
char * source;
char * destination;
int n = 5; // just an example!

 char * strncat (char * destination, const char *source, size_t n)
  {
        size_t destination_length = strlen(destination);
         size_t i; // size_t is size of data type, of course!

for ( i = 0 ; i < n && source[i] != '\0' ; i++ ) // (1) until source char array hits n characters,
          {
             destination[destination_length + i] = source[i]; // (2) copy over
             destination[destination_length+ i] = '\0';    // (3) set remaining to NULL bytes
          }
           return destination;
```

```
   } // function ends
```

// strncmp:

```
#include <string.h>
char * source;
char * destination;
int n = 5; // just an example!

int strncpy ( destination, source, n)
{

 for ( i = 0; i < n && source[i] != ‘\0’, i++ ) // (1) until source char array hits n characters,
 {
        int count;
        int forEach = ( destination[i] == source[i] ); (2) count number of chars that match

        if ( forEach == 1) { ++count; } // (3) increment counter if THESE elements match
        else {}

  } // for

 if ( count == ( n-1 ) // (4) if total number counted matches n elements
 { return 1; }              // (5) then return success

else
 { return 0; }              // (6) else return failure

} // function ends
```

K&R problem 5.7

Exercise 5-7. Rewrite readlines to store lines in an array supplied by main, rather than calling alloc to maintain storage. How much faster is the program?

BEFORE:

```
 /* readlines: read input lines */
 int readlines(char *lineptr[], int maxlines)
{
int len, nlines;
char *p, line[MAXLEN];
nlines = 0;
while ((len = getline(line, MAXLEN)) > 0)
if (nlines >= maxlines || p = alloc(len) == NULL)
return -1;
else {
line[len-1] = '\0'; /* delete newline */
strcpy(p, line);
lineptr[nlines++] = p;
}
return nlines;
}
```

AFTER:

```
 /* readlines: read input lines */
 int readlines(char *lineptr[], int maxlines, char * mainArray)
{
int len, nlines;
char *p, line[MAXLEN];
nlines = 0;

p = someChar + strlen(mainArray);            // char * p is same size as array in main! no “alloc”!!!

while ((len = getline(line, MAXLEN)) > 0)
//if (nlines >= maxlines || p = alloc(len) == NULL)
if  (nlines >= maxlines || strlen(mainArray) > ALLSTRINGSMAXVAL)
// THIS MAX #DEFINE: for size of all strings
return -1;
else {
line[len-1] = '\0'; /* delete newline */
strcpy(p, line);
lineptr[nlines++] = p;
}
return nlines;
}
```

Additional problem:

Answers are in bold and blue.

Consider the following assembly representation of a function foo containing a for loop:

```
foo:
  pushl %ebp                # push function onto stack frame
  movl %esp,%ebp            # now stack pointer points at stack frame base pointer ebp
  pushl %ebx                # push ebx onto stack
  movl 8(%ebp),%ebx         # ebx is now the variable a, which is ebp+8
  leal 2(%ebx),%edx         # new variable edx is ebx+2: so int i = a + 2; i is edx now!
  xorl %ecx,%ecx            # new variable ecx is XOR'ed with itself: ecx ^= ecx; int result ^= result;
  cmpl %ebx,%ecx            # compare ebx:ecx, so if ebx < ecx ... if a < result
  jge .L4                   # jump into loop if comparison returns true

.L6:                        # loop if comparison above returns FALSE
  leal 5(%ecx,%edx),%edx          # variable edx is edx+ecx +5:          i = i + result + 5;
  leal 3(%ecx),%eax         # new variable eax is ecx +3:                 int j = result + 3;
  imull %eax,%edx           # edx is eax*edx: edx *= ecx:                 i *= j;
  incl %ecx                 # increment ecx by one               ++j;
  cmpl %ebx,%ecx            # compare ebx: ecx                   a < result
  jl .L6                    # jump again to top of loop IF comparison returns true

.L4:                        # this is the end of the loop, where control is after comparison ret false
  movl %edx,%eax            # copy edx into eax: copy i into j:     j = i;
  popl %ebx                 # pop variable ebx (a) local variable off stack/stack frame
  movl %ebp,%esp            # prepare stack for return by resetting stack and base pointers
  popl %ebp                 # pop function's stack frame off of the stack
  ret                       # return result
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. To ensure full credit, explain your analysis of the assembly code. Note: you may only use the symbolic variables a, i, and result in your expressions below. Do not use register names.

```
int foo(int a)
{
   int i;
  i = a + 2;
   int result = _______________; int result ^= result;

if ( a < result )
{
        for ( i = i + result + 5; int j = result + 3; ++j; a < result )
        {
                i *= j;
        }
j = i;
   return result;
}
```