Machine-Level Representation of Programs

Annalise Tarhan

August 27, 2020

3 Homework Problems

3.58 Write C code for *decode2* that will have an effect equivalent to the assembly code shown.

```
long decode2(long x, long y, long z) {
        y -= z;
        x *= y;
        y <<= 63;
        y >>= 63;
        return x^y;
}
```

3.59 Describe the algorithm *store_prod* uses to compute the product and annotate the assembly code to show how it realizes the algorithm.

The 64-bit arguments x and y are first sign extended to 128 bits. Considering the two registers the extended values are stored in, each can then be written $x = 2^{64} * x_h + x_l$ and $y = 2^{64} * y_h + y_l$ where x_h and y_h are the higher bits and x_l and y_l are the lower bits. Their product is then $xy = 2^{128} x_h y_h + 2^{64} x_h y_l + 2^{64} y_h x_l + x_l y_l$. We are only concerned with the lower 128 bits, so the first product drops off, leaving $xy = 2^{64} x_h y_l + 2^{64} y_h x_l + x_l y_l$. We also know that the upper register of each factor is equal to the sign of the factor's lower register.

```
store_prod:
movq %rdx, %rax
cqto                    Extends y: low bits in %rax, high bits in %rdx
movq %rsi, %rcx
sarq $63, %rcx          Calculates x's sign, equivalent to xh
imulq %rax, %rcx        Calculates xh*yl
imulq %rsi, %rdx        Calculates yh*xl
addq %rdx, %rcx         Adds xh*yl+yh*xl
```

mulq %rsi	Calculates xl*yl: low bits in %rax, high bits in %rdx
addq %rcx, %rdx	Adds (xh*yl + yh*xl) to the upper bits of xl*yl
movq %rax, (%rdi)	Moves the low bits of xl*yl to (*dest)
movq %rdx, 8(%rdi)	Moves the sum of the upper bits to 8(*dest)

3.60 Consider the function *loop* and the assembly code it generates.

3.60.1 Which registers hold program values x, n, result, and mask?

x in %rdi, n in %esi, result in %rax, mask in %rdx

3.60.2 What are the initial values of result and mask?

Result starts as 0, mask as 1

3.60.3 What is the test condition for mask?

$mask \neq 0$

3.60.4 How does mask get updated?

Once per loop, it is shifted left by n bits.

3.60.5 How does result get updated?

Every loop, result is updated to $result|(x\&mask)$.

3.60.6 Fill in all the missing parts of the C code.

```
long loop(long x, int n) {
        long result = 0;
        long mask;
        for (mask = 1; mask != 0; mask = mask << n) {
                result |= mask & x;
        }
        return result;
}
```

3.61 Write a C function *cread_alt* that has the same behavior as *cread*, except that it can be compiled to use conditional data transfer.

```
long cread_alt(long *xp) {
        return (!xp ? 0 : *xp);
}
```

3.62 Fill in the missing parts of the C code for *switch3*.

```
typedef enum
         {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

long switch3(long *p1, long *p2, mode_t action) {
        long result = 0;
        switch(action) {
                case MODE_A:
                        result = *p2;
                        action = *p1;
                        *p2 = action;
                        break;
                case MODE_B:
                        result = *p1 + *p2;
                        *p1 = result;
                        break;
                case MODE_C:
                        *p1 = 59;
                        result = *p2;
                        break;
                case MODE_D:
                        result = *p2;
                        *p1 = result;
                case MODE_E:
                        result = 27;
                        break;
                default:
                        result = 12;
                        break;
        }
        return result;
}
```

3.63 Fill in the body of the switch statement with C code that will have the same behavior as the machine code.

```
long switch_prob(long x, long n) {
        long result = x;
        switch(n) {
                case 60:
                case 62:
                        result = x * 8;
                        break;
                case 63:
                        result = x >> 3;
                        break;
                case 64:
                        x *= 15;
                case 65:
                        x *= x;
                default:
                        result = x + 0x4b;
        }
        return result;
}
```

3.64 Consider the function *store_ele*.

3.64.1 Extend equation 3.1 from two dimensions to three to provide a formula for the location of array element A[i][j][k].

For an array declared as A[R][S][T] and beginning at x_A, element A[i][j][k] is located at $x_A + L(S*T*i + T*j + k)$ where L is the size of the array elements' data type.

3.64.2 Determine the values of R, S, and T based on the assembly code.

The total size of the array is 3640 and the size of the data type is 8, so based on the previous formula: R=7, S=5, and T=13.

3.65 Consider the function *transpose*.

3.65.1 Which register holds a pointer to array element A[i][j]?

%rdx

3.65.2 Which register holds a pointer to array element A[j][i]?

%rax

3.65.3 What is the value of M?

15

3.66 Consider the function *sum_col*. What are the definitions of NR and NC?

NR is 3n, NC is 4n+1

3.67 Consider the functions *process* and *eval* and the GCC generated code.

3.67.1 Diagram the stack frame for *eval*, showing the values that it stores on the stack prior to calling process.

%rsp: x, %rsp+8: y, %rsp+16: &z, %rsp+24: z

3.67.2 Which value does *eval* pass in its call to *process*?

It passes s, an instance of strA.

3.67.3 How does the code for *process* access the elements of structure argument s?

process accesses the elements by calculating the offset of the elements relative to the stack pointer. The first element of s is located at 8(%rsp), since the return address for eval is added to the stack when process is called.

3.67.4 How does the code for *process* set the fields of result structure r?

process uses the value in register %rdi as the location for the first element and calculates the offsets of the other elements based on the first.

3.67.5 Complete the diagram of the stack frame for *eval*, showing how *eval* accesses the elements of structure r following the return from *process*.

%rsp: x, %rsp+8: y, %rsp+16: &z, %rsp+24: z, %rsp+64: y, %rsp+72: x, %rsp+80: z.

3.67.6 What general principles can you discern about how structure values are passed as function arguments and how they are returned as function results?

At the beginning of a function, space is allocated for the arguments, as well as the return value, of any other functions called. The arguments are stored at the very top of the stack, the space for the return value further down. The address of the return value is passed via a register.

3.68 Consider the function *setVal*. What are the values of A and B?

setVal accesses $q \rightarrow t$ at 8(%rsi), so the size of char array[B] must be at least 5 but no more than 8, since int t must be aligned at a multiple of 4. Similarly, the size of short s[A] must be at least 13 but not more than 20, since it begins immediately after t ends at 12(%rsi) and long u is accessed at 32 and is aligned at a multiple of 8. Taking the size of char and short into account, this gives $5 \leq B \leq 8$ and $7 \leq A \leq 10$.

The size of int x[A][B] is implied by the location of $p \rightarrow y$ at 184(%rdi). By the same logic as before, $177 \leq 4AB \leq 184$, which reduces to $44.25 \leq AB \leq 46$ or more practically $45 \leq AB \leq 46$. The only integers that satisfy all three inequalities are A=9 and B=5.

3.69 Consider the function *test* and the generated .o code.

3.69.1 What is the value of CNT?

a begins at 8(%rsi) and ends at 288(%rsi), and the ith element of a is accessed by adding 40i to the start position, so the size of *a_struct* is 40 and CNT=7.

3.69.2 Give a complete declaration of structure *a_struct*.

```
typedef struct {
        long idx;
        long x[4];
} a_struct;
```

3.70 Consider the *ele* union declaration.

3.70.1 What are the offsets of the following fields?

e1.p 0

e1.y 8

e2.x 0

e2.next 8

3.70.2 How many total bytes does the structure require?

16

3.70.3 Fill in the procedure based on the assembly code.

```
void proc (union ele *up) {
        up -> e2.x =
                *(up -> e2.next -> e1.p) -
                up -> e2.next -> e1.y;
}
```

3.71 Write a function *good_echo* that reads a line from standard input and writes it to standard output.

```
void good_echo() {
        int size = 100;
        int count = 0;
        int c;
        char buffer[size+1];
        while (count < size
                && (c = getchar()) != '\n'
                && c != EOF) {
                        buffer[count] = c;
                        size++;
        }
        buffer[count+1] = '\0';
        puts(buffer);
}
```

3.72 Consider the function $aframe$.

Note: I really don't get this exercise. Why does it align at 16 and not 8? Why can't the stack pointer point to the same place as p? In other words, why can't $e_1 = 0$? Why do we need so much padding?

3.72.1 Explain the logic in the computation of s_2.

s_2 is computed by subtracting enough space for long i and array p from the frame pointer's value. The space for the array is calculated by multiplying the number of array elements by 8, adding 16, and then rounding up to the nearest multiple of 16. If n is even, $arrayspace = 8n + 16$, and if it is odd, $arrayspace = 8n + 24$. In both cases, $s_2 = s_1 - 16 - arrayspace$.

3.72.2 Explain the logic in the computation of p.

p is set to the first multiple of 16 larger than s_2. First, a bias of 15 is added to s_2. Then, performing an *and* operation with -16 sets the lower four bits to zero, rounding back down to a multiple of 16. $p = ((s_2 + 15)/16) * 16$.

3.72.3 Find values of n and s_1 that lead to minimum and maximum values of e_1.

Similarly to the example, the crucial factors are whether n is even or odd, and whether s_1 is a multiple of 16 or off by one. The combination that minimizes e_1 is an odd n and s_1 that is one greater than a multiple of sixteen. $n = 5$ and $s_1 = 2065$, for example. Conversely, maximize e_1 with an even n and s_1 that is a multiple of 16, such as $n = 4$ and $s_1 = 2064$.

3.72.4 What alignment properties does this code guarantee for the values of s_2 and p?

The alignment of s_2 preserves the offset of s_1 and p is aligned at a multiple of 16.

3.73 Write a function in assembly code that matches the behavior of *find_range*, but uses conditional branches and only one floating-point comparison.

```
.globl _find_range

_find_range:
vxorps   %xmm1, %xmm1, %xmm1
vucomiss     %xmm0, %xmm1
        ja   .L1
        je   .L2
        jb   .L3
        jp   .L4
.L1:
        movl $0, %eax
        jmp .end
.L2:
        movl $1, %eax
        jmp .end
.L3:
        movl $2, %eax
        jmp .end
.L4:
        movl $3, %eax
        jmp .end
.end:
        ret
```

```
int find_range(float x);

...

int main(int argc, char *argv[]) {
        for (int i = INT_MIN; i < INT_MAX; i++) {
                float flt = bits_to_float(i);
                if (boring_way(flt) != find_range(flt))
                printf("Choked on %f\n", flt);
        }
        return 0;
}
```

3.74 Write a function in assembly code that matches the behavior of *find_range*, but uses conditional moves and only one floating-point comparison.

```
_find_range:
        vxorps   %xmm1, %xmm1, %xmm1
        movl $0, %r10d
        movl $1, %r11d
        movl $2, %r12d
        movl $3, %r13d
        vucomiss    %xmm0, %xmm1
        cmova %r10d, %eax
        cmove %r11d, %eax
        cmovb %r12d, %eax
        cmovp %r13d, %eax
        ret
```

3.75 Consider the example functions, which work with complex data.

3.75.1 How are complex arguments passed to a function?

Complex arguments use two media registers each. The real part is stored in the first register, the imaginary part in the second. If there is more than one complex argument, the first register contains R1, the second C1, the third R2, and the fourth C2.

3.75.2 How are complex values returned from a function?

Complex values are returned on the same media registers as the arguments.