CS341 HW2

Ryan Scherbarth

October 2024

1. **What is the assembly instruction to move a double word? How many bits does it move?**

   movl is the instruction to move a double word. A word is 16 bits in x86-64, so it would move a total of 32 bits for a double word.

2. **What is the resulting memory address for the following operands assuming the indicated values in the registers?**

   (a) **-1(%rsp, %rcx, 4)**
   %rsp = 0x09
   %rcx = 0x05
   Addr = %rsp + %rcx * 4 - 1
   Addr = 0x21

   (b) **4(%rsp, %rcx, 4)**
   %rsp = 0x08
   %rcx = 0x0A
   Addr = %rsp + % rcx * 4 + 4
   Addr = 0x34

   (c) **256(, %rcx, 2)**
   %rcx = 0x0A
   Addr = %rcx * 2 + 256
   Addr = 0x114

   (d) **0x33(, %rcx, 3)**
   %rcx = 0x0B
   Addr = %rcx * 3 + 0x33
   Addr = 0x54

3. **What is the value in hexadecimal of the high-order 32 bits of register %rax after executing each of the following instructions? Provide an answer for each specific instruction.**

   (a) **movl $0x11, %eax**
   movl would take the value 0x11 and move it to the %eax register, which represents the lower 32 bits of the larger 64 bit %rax register. After the movl command is executed, moving the value to the lower order, the value of the higher order bits would simply be 0x00000000.

   (b) **movl $0x0B, %eax**
   Similar to part a, we will also end up with 0x00000000. Again, we are manipulating values with the %eax register. Whenever we use this register in the x86-86 bit OS it will automatically zero out the higher level bits.

4. **Provide the appropriate suffix and explain what is the instruction doing: mov__ (%rax), %dx**

   The complete instruction is: movw (%rax), %dx
   movw introduces the w suffix, standing for word. Here, this means we are moving the size of a word, defined as 2 bytes (16 bits). %rax is the memory address, and %dx is the 16 bit register, which is the lower quarter (16 bits) of the %rdx register (64 bits).

5. **Consider these two declarations in C: int *x and char *y. Suppose that x is in register %rdi and y is in register %rsi, what is the assembly code to copy the value at the address stored in %rdi to the address stored in %rsi.**

   To copy x from %rdi to %rsi, we have to start by de-referencing the pointer in %rdi (which will hold int *, a 4 byte value) and then copy it to the location %rsi is pointing to (which is holding a char *.

   movl (%rdi), %eax
   movl %eax, (%rsi)

   Will properly copy the int value from the memory location that %rdi is pointing to into the memory location that %rsi is pointing to.

6. **What is the value in register %rdx after executing the following Assembly code**
   leaq -1(%rsp, %rcx, 4), %rdx
   assuming %rsp and %rcx have values 0x09 and %0x05, respectively

   leaq, load effective address, calculates the memory address but does not access that memory, so we are calculating the value and storing it in %rdx.
   %rsp = 0x09
   %rcx = 0x05 * 4 = 0x14
   %rdx = 0x09 + 0x14 - 1 = 0x1C

7. **What is the value (in decimal) in register %rsp after executing the following instructions in Assembly code?**
   movl $1, %esp
   addq $2, %rsp
   incq %rsp
   salq $1, %rsp

   movl $1, %esp moves value 1 into %esp (the lower 32 bits of %rsp). The upper 32 bits of %rsp will be set to 0 whenever we write to %esp.

   addq $2, %rsp is going to add 2 to the 64 bit register %rsp, so the result will now be %rsp = 1+2 = 3.

   incq %rsp increments the register %rsp by 1, so the new value will be %rsp = 3 + 1 = 4.

   salq $1 %rsp will perform a left shift on %rsp by a single bit, which is equivalent to multiplying the decimal value by 2. The result will now be %rsp = 4 * 2 = 8.

   Therefore, the final value will be %rsp = 8.

8. **What is the value (in decimal) in register %rax after executing the following instructions in assembly code?**
movl $4, %eax
imulq $8, %rax
sarq $2, $rax
andq $4, %rax

movl $4, %eax moves the value 4 into the %eax register, which is hte lower 32 bits of %rax. The first 32 bits of %rax will be set to 0.

imulq $8, %rax multiplies the value stored in %rax by 8, so the new value of %rax will now be 32.

sarq $2, %rax jperforms a right shift by 2 bits on the value stored in %rax. The right shift operation will preserve the sign bit, so after shifting the current value, 32, by 2 bits will result in the new value of %rax being 8.

andq $4, %rax performs a bitwise AND between %rax and 4. Converting both to binary, we have 1000 AND 0100. This will result in 0000 since there are no bits the same between the two inputs to the AND operation. The final value stored in %rax will be 0.

9. **What is the value of the SF condition code after executing the following instructions in assembly code?**
movb $0, %b1
cmpb $0x01, %b1

movb $0, %b1 moves the value 0 into the 8 bit register %b1.

cmpb $0x01, %b1 compares the 0x01 with the value in %b1, both of which are equal. The value of the SF flag is determined by comparing 0x01 with 0, and since 0 - 1 is negative, the SF flag gets set to 1 here.

10. **What is the value of the ZF condition code after executing the following instructions in assembly code?**
movb $1, %b1
cmpb $0x01, %b1

movb $0, %b1 moves the value 0 into the 8 bit register %b1.

cmpb $0x01, %b1 compares the 0x01 with the value in %b1, both of which are equal. Here, the ZF condition code will be set to 1 if the result of a subtraction is 0. When we do cmpb we are essentially just subtracting from 1 here, so the result of the ZF flag after this instruction will be set to 1.

11. **Problem 3.63, p. 314-316**
This problem will give you a chance to reverse engineer a switch statement from disassembled machine code. In the following procedure, the body of the switch statement has been omitted.

The jump table resides in a different area of memory. We can see from the indirect jump on line 5 that the jump table begins at address 0x4006f8. Using the GDB debugger, we can examine the six 8-byte words of memory comprising the jump table with the command x/6gx 0x4006f8 . **GBD prints the following:**

```
(gdb) x/6gx 0x4006f8
0x4006f8:      0x00000000004005a1      0x00000000004005c3
0x400708:      0x00000000004005a1      0x00000000004005aa
0x400718:      0x00000000004005b2      0x00000000004005bf
```

Code:

```
long switch_prob(long x, long n){
  long result = x;
  switch(n) {
    /* Fill in code here */
  }
  return result;
}
```

Disassembled Code:

```
1  0000000000400590 <switch_prob>
2  400590: 48 83 ee 3c              sub     $0x3c, %rsi
3  400594: 48 83 fe 05              cmp     $0x5, %rsi
4  400598: 77 29                    ja      4005c3 <switch_prob+0x33>
5  40059a: ff 24 f5 f8 06 40 00     jmpq    *0x4006f8(,%rsi,8)
6  4005a1: 48 8d 04 fd 00 00 00     lea     0x0(,%rdi,8),%rax
7  4005a8: 00
8  4005a9: c3                       retq
9  4005aa: 48 89 f8                 mov     %rdi, %rax
10 4005ad: 48 c1 f8 03              sar     $0x3, %rax
11 4005b1: c3                       retq
12 4005b2: 48 89 f8                 mov     %rdi, %rax
13 4005b5: 48 c1 e0 04              shl     $0x4, %rax
14 4005b9: 48 29 f8                 sub     %rdi, %rax
15 4005bc: 48 89 c7                 mov     %rax, %rdi
16 4005bf: 48 0f af ff              imul    %rdi, %rdi
17 4005c3: 48 8d 47 4b              lea     0x4b(%rdi),%rax
18 4005c7: c3                       retq
```

The default case is covered in lines 3-4
Here, we get another hint on line 3, where we compare n to 5, and then jump to a new location. This tells us there is a total of [0..5] = 6 cases including our base case.

case 0 and 2 correspond to lines 7-9
lea multiplies x by 8 and returns, so this case ends up just being result = x * 8.

case 1 corresponds to lines 6-8
Here, we move x to %rax and perform a right bit-wise shift of 3, which is equivalent to dividing by 8.

case 3 corresponds to lines 12-16
Here, we bitwise-shift x left by 4 (multiplying by 16), then subtract x, then we multiply this result by x. so we get result = (x*16 - x) * x

case 4 is given by lines 17-18
Here we load x + 75 into result, then we return result = x + 75 when case n == 4 is satisfied.

case 5 corresponds to lines 19-21
Here, we simply return result = 12

We can now reconstruct the switch case based off of our findings.

We can use the jump table on page 315 to determine the memory locations of the next entries in the jump table by looking at the output from the GDB command. The next entries (memory addresses) will be
2. 0x400700
4. 0x400710
8. 0x400718

```
switch(n) {
  case 0:
  case 2:
    result = 8x;
    break;
  case 1:
    result = x/8;
    break;
  case 3:
    result = (16x - x) * x;
    break;
  case 4:
    result = x + 75;
    break;
  case 5:
    result = 12;
    break;
  default:
    result = x + 75;
}
```

12. **What is the relationship between $T_{\max}$ and $U_{\max}$? Explain** an n bit signed int has a maximum value $T_{\max} = 2^{n-1} - 1$, since the other $n - 1$ bits are used for the magnitude, and one bit is saved for the sign.

    an n bit unsigned integer, however, is given by $U_{\max} = 2^n - 1$, this is because we use all n bits to represent the magnitude in this case.

    So we can see the relationship between the two numerically is $U_{\max} = 2 * T_{\max} + 1$

13. **Convert the value 0xF1AB to binary and apply the Binary two's complement (B2T) encoding to it, what is the resulting number?**

    0xF = 1111
    0x1 = 0001

0xA = 1010
0xB = 1011
0xF1AB = 1111 0001 1010 1011
$0xF1AB_{\text{inv}}$ = 0000 1110 0101 0100
$0xF1AB_{\text{2's comp}}$ = 0000 1110 0101 0101

14. **What is the value of x? (do it by hand, and show partial results at each step; do not run it in C)** int $x = (0xD2 \ \& \ (1 << 7)) >> 7;$

0xD2 = 1101 0010
1101 0010 & 1000 0000 = 1000 0000
1000 0000 >> 7 = 0000 0001
x = 1

15. **What is the value of x? (do it by hand, and show partial results at each step; do not run it in C)** int $x = (0xD2 \ \& \ (1 << 3)) >> 3;$

0xD2 = 1101 0010
1101 0010 & 0000 1000 = 0000 0000
0000 0000 >> 0000 0000 = 0000 0000
x = 0

16. **Consider the 12-bit IEEE floating-point representation with 4 bits of exponent and 7 bits of fraction.**

- **(a) What is the bias? Show your intermediate work to compute the bias.**

The bias is given by $B = 2^{(k-1)} - 1$.

$$\begin{aligned} B &= 2^{(k-1)} - 1 \\ B &= 2^{(4-1)} - 1 \\ &= 2^3 - 1 \\ &= 7 \end{aligned}$$

So we get a Bias of 7.

- **(b) What is the value of 0111 1000 0000 in this 12-bit representation? Explain all the components of the bit pattern and which of the three cases it falls into.**

In this 12-bit representation we have 1 sign bit, of the 10 remaining we know the bias to be 7, so the exponent is $12 - 1 - 7 = 4$.
sign: 1, exp: 4, mantissa: 7

The sign bit is 0, which means the number is positive.

the exponent bits are 1111, we can compute the exponent by subtracting the bias;
$1111_2 = 15_{10}$, Exp $= 15 - 7 = 8$

Since our exponent is not all 0's or 1's, we have a normalized representation, so the total value will be give by

0111 1000 0000 $= 1.0 \times 2^8 = 256$

- **(c) What is the bit pattern for the decimal value 34.75 in this format?**

The binary representation of the integer portion is $34_2 =$ 0010 0010.
The binary representation of the fraction will be 0.11 since $0.75 = \frac{1}{2^1} + \frac{1}{2^2}$
Now we have a direct binary representation of 0010 0010.11.

We next need to normalize our result. To normalize we will use the function $1.f \times 2^{\text{exp}}$
0010 0010.11 $\times 2^5$

The exponent will be given by the sum of the exponent and the bias; $5 + 127 = 132_{10} = 1000\ 0100_2$.

Finally, we can combine all of the pieces to form our final number;
0 1000 0100 0000 1011 0000 0000 0000 000

17. **Given the following encoding of a single-precision floating point number;**
1 1000 0000 1100 0000 0000 0000 0000 000 ,
What are the following values? (Show all values in decimal; show M in both binary and decimal; show full equation for V)

- **(a) exp=** $1000\ 0000_2 = 128_{10}$
- **(b) Bias =** 127
- **(c) E =** 128 - 127 = 1
- **(d) M =** $1 + 2^{-1} + 2^{-2} = 1.75$
- **(e) V =** $(-1)^S * M * 2^E = -3.5$

18. **Given the following encoding of a single-precision floating point number; 1 1000 0001 1100 0000 0000 0000 0000 000 , What are the following values? (Show all values in decimal; show M in both binary and decimal; show full equation for V)**

- **(a) exp=** $1000\ 0001_2 = 129_{10}$
- **(b) Bias =** 127
- **(c) E =** 129 - 127 = 2
- **(d) M =** $1 + 2^{-1} + 2^{-2} = 1.75$
- **(e) V =** $-1^1(1.75 * 2^2) = -7$

19. **Assuming Little Endian and that you have the addresses 0x404 to 0x407 available, how would the following data be stored in memory, if you start at address 00x404 0x5566AAA19?**
Our least significant bit will be stored at the lowest memory address, so to break up the data given;

0x404: 0x19
0x405: 0xAA
0x406: 0x66
0x407: 0x55

So the final result will be 0x19 0xAA 0x66 0x55

20. **Assuming Little Endian and that you have the addresses 0x404 to 0x407 available, how would the following data be stored in memory, if you start at address 00x404 0x1CFF22?**

0x404: 0x22
0x405: 0xFF
0x406: 0x1C
0x407: 0x00

The final result will be 0x22 0xFF 0x1C