CS107, Lecture 17
Assembly: Arithmetic and Logic, Take II

Reading: B&O 3.5-3.6
Ed Discussion: https://edstem.org/us/courses/46162/discussion/3771766
Assembly Exploration

• Let’s pull these commands together and see how some C code might be translated to assembly.

• Compiler Explorer is a handy website that lets you quickly write C code and see its assembly translation. Let’s check it out!

• https://godbolt.org/z/Ecbde99e3
int calculate(int x, int arr[]) {
    int sum = x;
    sum += arr[0];
    sum <<= x;
    sum &= 512;
    return sum;
}

----------

calculate:
    movl %edi, %ecx
    movl %edi, %eax
    addl (%rsi), %eax
    sal %cl, %eax
    andl $512, %eax
    ret
Large Multiplication

- Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?

- If you specify two operands to `imul`, it multiplies them together and truncates it to fit in the second of the two 64-bit register operands.

\[
\text{imul } S, D \quad D \leftarrow D \ast S
\]

- If you specify one operand, it multiplies that by `%rax`, and splits the product across 2 registers. It puts the high-order 64 bits in `%rdx` and the low-order 64 bits in `%rax`.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imulq S</code></td>
<td>R[%rdx]:R[%rax] ← S x R[%rax]</td>
<td>Signed full multiply</td>
</tr>
<tr>
<td><code>mulq S</code></td>
<td>R[%rdx]:R[%rax] ← S x R[%rax]</td>
<td>Unsigned full multiply</td>
</tr>
</tbody>
</table>
Division and Remainder

• **Terminology:** dividend / divisor = quotient with remainder

• **x86-64** supports dividing up to a 128-bit value by a 64-bit value.

• The high-order 64 bits of the dividend need to be prepared and stored in `%rdx`, the low-order 64 bits in `%rax`. The divisor is the only listed operand.

• The quotient is stored in `%rax`, and the remainder in `%rdx`.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idivq S</td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S</td>
<td>Signed divide</td>
</tr>
<tr>
<td>divq S</td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S</td>
<td>Unsigned divide</td>
</tr>
</tbody>
</table>
**Division and Remainder**

**Terminology:** dividend / divisor = quotient with remainder

The high-order 64 bits of the dividend need to be prepared and stored in %rdx, the low-order 64 bits in %rax. The divisor is the only listed operand.

Most division uses only 64-bit dividends. The `cqto` instruction sign-extends the 64-bit value in %rax into %rdx to fill both registers with the dividend, as the division instruction expects.

**Instruction** | **Effect** | **Description**
---|---|---
`idivq S` | R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] \( \div S \) | Signed divide

`divq S` | R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] \( \div S \) | Unsigned divide

`cqto` | R[%rdx]:R[%rax] ← SignExtend(R[%rax]) | Convert to oct word
Compiler Explorer Demo

https://godbolt.org/z/4cT75M4nd
// Returns x/y, stores remainder in location stored in remainder_ptr
long full_divide(long x, long y, long *remainder_ptr) {
    long quotient = x / y;
    long remainder = x % y;
    *remainder_ptr = remainder;
    return quotient;
}

-------

full_divide:
    movq %rdi, %rax
    movq %rdx, %rcx
    cqto
    idivq %rsi
    movq %rdx, (%rcx)
    ret
Assembly Exercise 1

0000000000040116e <sum_example1>:
  40116e: 8d 04 37          lea (%rdi,%rsi,1),%eax
  401171: c3                  retq

Which of the following is most likely to have generated the above assembly?

// A)
void sum_example1() {
  int x;
  int y;
  int sum = x + y;
}

// B)
int sum_example1(int x, int y) {
  return x + y;
}

// C)
void sum_example1(int x, int y) {
  int sum = x + y;
}
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}

What location or value in the assembly above represents the C code’s sum variable?

%eax
int sum_example2(int arr[]) {  
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}

What location or value in the assembly code above represents the C code’s 6 (as in arr[6])?

0x18
int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}

---------
// x in %edi, arr in %rsi, i in %edx
add_to:
    movslq %edx, %rdx
    movl %edi, %eax
    addl (%rsi,%rdx,4), %eax
    ret
int add_to(int x, int arr[], int i) {
    int sum = ___?
    sum += arr[___?
    return ___?
} 

----------
// x in %edi, arr in %rsi, i in %edx
add_to:
    movslq %edx, %rdx     // sign-extend i into full register
    movl %edi, %eax      // copy x into %eax
    addl (%rsi,%rdx,4), %eax  // add arr[i] to %eax
    ret
int add_to(int x, int arr[], int i) {
    int sum = x;
    sum += arr[i];
    return sum;
}

---------
// x in %edi, arr in %rsi, i in %edx
add_to:
    movslq %edx, %rdx // sign-extend i into full register
    movl %edi, %eax // copy x into %eax
    addl (%rsi,%rdx,4), %eax // add arr[i] to %eax
    ret
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}

----------
// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax
    imull (%rdi), %eax
    subl 4(%rdi), %eax
    sarl $2, %eax
    addl $2, %eax
    ret
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}

----------
// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax        // copy y into %eax
    imull (%rdi), %eax     // multiply %eax by nums[0]
    subl 4(%rdi), %eax     // subtract nums[1] from %eax
    sarl $2, %eax          // shift %eax right by 2
    addl $2, %eax          // add 2 to %eax
    ret
int elem_arithmetic(int nums[], int y) {
    int z = nums[0] * y;
    z -= nums[1];
    z >>= 2;
    return z + 2;
}

// nums in %rdi, y in %esi

elem_arithmetic:
    movl %esi, %eax // copy y into %eax
    imull (%rdi), %eax // multiply %eax by nums[0]
    subl 4(%rdi), %eax // subtract nums[1] from %eax
    sarl $2, %eax // shift %eax right by 2
    addl $2, %eax // add 2 to %eax
    ret
int \texttt{sum\_array}(\texttt{int arr[]}, \texttt{int nelems}) {
    \texttt{int sum} = 0;
    \texttt{for (int i} = 0; \texttt{i < nelems; i++}) {
        \texttt{sum += arr[i];}
    }
    \texttt{return sum;}
}

We’re 1/2 of the way to understanding assembly!

What looks understandable right now?