CS201- Lecture 7
IA32 Data Access and Operations Part II

RAOUL RIVAS

PORTLAND STATE UNIVERSITY

Announcements

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

- `movq 8(%rdx),%rax`
- `movq (%rdx,%rcx),%rax`
- `movq (%rdx,%rcx,4),%rax`

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Compute Effective Address

- **`leaq` *Src*, *Dst***
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression

- Uses
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

- Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Basic Arithmetic Operations

- Two Operand Instructions:

Format	*Computation*		
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest – Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic***
`shrq`	*Src,Dest*	Dest = Dest >> Src	***Logical***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- Watch out for argument order!
- Notice the accumulator format
  - What's the C equivalent of this accumulator operators?

Unary Operators

- One Operand Instructions

`incq`	*Dest*	*Dest = Dest + 1*
`decq`	*Dest*	*Dest = Dest – 1*
`negq`	*Dest*	*Dest = – Dest*
`notq`	*Dest*	*Dest = ~Dest*

Translating from C

```
long easyOp(long x, long y)
{
  long rval;

  x = x + 2;
  y = y * x;
  rval = y;
  return rval;
}
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`

Translating from C

```
long easyOp(long x, long y)
{
  long rval;

  x = x + 2;
  y = y * x;
  rval = y;
  return rval;
}
```

```
easyOp:
   addq    $2, %rdi
   imulq   %rdi, %rsi
   movq    %rsi, %rax
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	rval

If you find accumulator arithmetic tricky try writing some C programs in this way

Accumulators

- Can we do the same in Accumulator and in Non-Accumulator architectures?
  - Yes. They are Turing Equivalent
    - Computer P can simulate computer Q and computer Q can simulate computer P
    - Any program that runs on P can be rewritten to run in Q

IBM 701
38 bit single accumulator register

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   movq     %rdi, %rax
   addq     %rsi, %rax
   addq     %rdx, %rax
   movq     %rdi, %rcx
   addq     $4, %rcx
   imulq    $48, %rsi
   addq     %rsi,%rcx
   imulq    %rcx, %rax
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**

Arithmetic Expression Optimization

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z

```
arith:
   leaq    (%rdi,%rsi), %rax
   addq    %rdx, %rax
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx
   leaq    4(%rdi,%rdx), %rcx
   imulq   %rcx, %rax
   ret
```

Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
  - But, only used once

Arithmetic Expression Optimization

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	**t1**, **t2**, **rval**
%rdx	**t4**
%rcx	**t5**

Arithmetic tricks

- Shift to replace multiplication
  - **`salq $4, %rdx`**

- Use of LEA to replace complex arithmetic
  - **`leaq (%rsi,%rsi,2), %rdx`**

- XOR to replace initialization to zero
  - **`xorq %rax,  %rax`**

What's Next?

- At this point we can write some simple Assembly functions
  - Read and Write from Memory
    - Variables, Pointers, Arrays, Structures
  - Logic and Arithmetic Operations
- What's missing?
  - Comparisons
  - Control Statements
    - If/Else, While, Do/While
  - Function Calls

Summary

- LEA instruction is used to compute addresses without a memory reference
- LEA is also used to optimize certain patterns of arithmetic computations
- Arithmetic Operations use an accumulator based notation