# Complete Memory Addressing Modes

- **Remember, the addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways**
- **Most General Form:**

| | |
|---|---|
| **D(Rb,Ri,S)** | **Mem[Reg[Rb] + S*Reg[Ri] + D]** |

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of the 8/16 integer registers
  - Ri: Index register: Any, except for `%esp` or `%rsp`
    - Unlikely you'd use `%ebp`, either
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*) arrays
- **Special Cases: can use any combination of D, Rb, Ri and S**

| | |
|---|---|
| **(Rb,Ri)** | **Mem[Reg[Rb]+Reg[Ri]]** |
| **D(Rb,Ri)** | **Mem[Reg[Rb]+Reg[Ri]+D]** |
| **(Rb,Ri,S)** | **Mem[Reg[Rb]+S*Reg[Ri]]** |

# Address Computation Examples

| | |
|---|---|
| `%edx` | `0xf000` |
| `%ecx` | `0x100` |

| | |
|---|---|
| (Rb,Ri) | Mem[Reg[Rb]+Reg[Ri]] |
| D(,Ri,S) | Mem[S*Reg[Ri]+D] |
| (Rb,Ri,S) | Mem[Reg[Rb]+S*Reg[Ri]] |
| D(Rb) | Mem[Reg[Rb] +D] |

| Expression | Address Computation | Address |
|---|---|---|
| `0x8(%edx)` | `0xf000 + 0x8` | `0xf008` |
| `(%edx,%ecx)` | `0xf000 + 0x100` | `0xf100` |
| `(%edx,%ecx,4)` | `0xf000 + 4*0x100` | `0xf400` |
| `0x80(,%edx,2)` | `2*0xf000 + 0x80` | `0x1e080` |

# Address Computation Instruction

- **`leal` *Src,Dest***
  - *Src* is address mode expression
  - Set *Dest* to address computed by expression
    - (lea stands for *load effective address)*
  - Example: **`leal (%edx,%ecx,4), %eax`**

%eax = %edx + 4 * %ecx

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

# Some Arithmetic Operations

- **Two Operand (Binary) Instructions:**

| *Format* | | *Computation* | |
|---|---|---|---|
| `addl` | *Src,Dest* | *Dest = Dest + Src* | |
| `subl` | *Src,Dest* | *Dest = Dest − Src* | |
| `imull` | *Src,Dest* | *Dest = Dest * Src* | |
| `sall` | *Src,Dest* | *Dest = Dest << Src* | *Also called shll* |
| `sarl` | *Src,Dest* | *Dest = Dest >> Src* | *Arithmetic* |
| `shrl` | *Src,Dest* | *Dest = Dest >> Src* | *Logical* |
| `xorl` | *Src,Dest* | *Dest = Dest ^ Src* | |
| `andl` | *Src,Dest* | *Dest = Dest & Src* | |
| `orl` | *Src,Dest* | *Dest = Dest \| Src* | |

- **Watch out for argument order! (especially `subl`)**
- **No distinction between signed and unsigned int (why?)**

# Some Arithmetic Operations

- **One Operand (Unary) Instructions**

| | | |
|---|---|---|
| **`incl`** | *Dest* | *Dest* = *Dest* + 1 |
| **`decl`** | *Dest* | *Dest* = *Dest* - 1 |
| **`negl`** | *Dest* | *Dest* = -*Dest* |
| **`notl`** | *Dest* | *Dest* = ~*Dest* |

- **See textbook section 3.5.5 for more instructions: `mull`, `cltd`, `idivl`, `divl`**

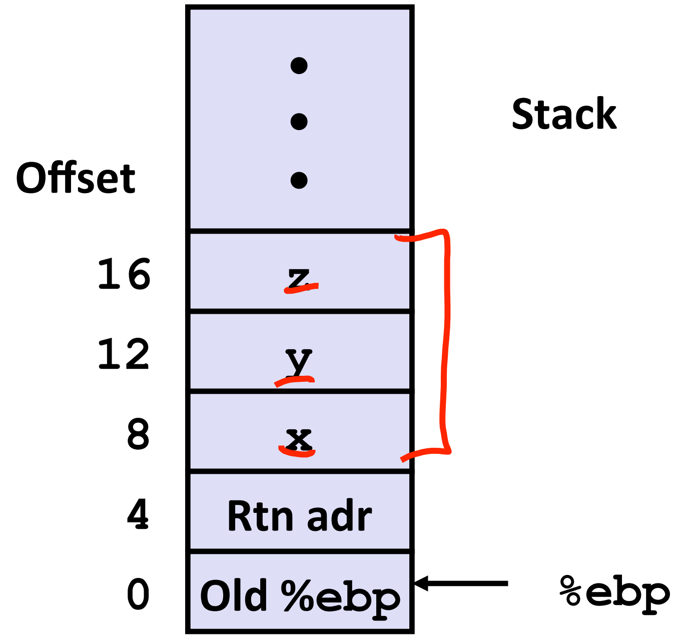
# Using `leal` for Arithmetic Expressions

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

```
arith:
    pushl %ebp                  } Set Up
    movl %esp,%ebp

    movl 8(%ebp),%eax           } Body
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

    movl %ebp,%esp              } Finish
    popl %ebp
    ret
```

# Understanding `arith`

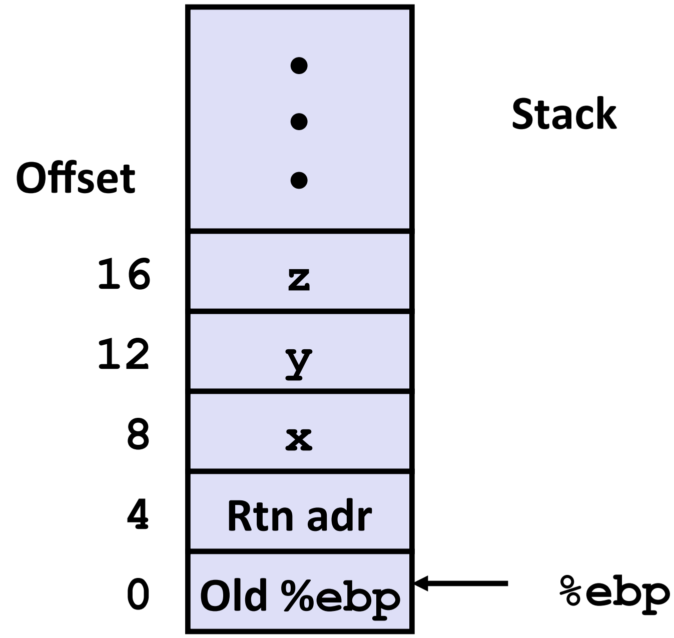
```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
movl  8(%ebp),%eax          # eax = x
movl  12(%ebp),%edx         # edx = y
leal  (%edx,%eax),%ecx      # ecx = x+y  (t1)
leal  (%edx,%edx,2),%edx    # edx = y + 2*y = 3*y
sall  $4,%edx               # edx = 48*y (t4)
addl  16(%ebp),%ecx         # ecx = z+t1 (t2)
leal  4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax             # eax = t5*t2 (rval)
```

# Understanding `arith`

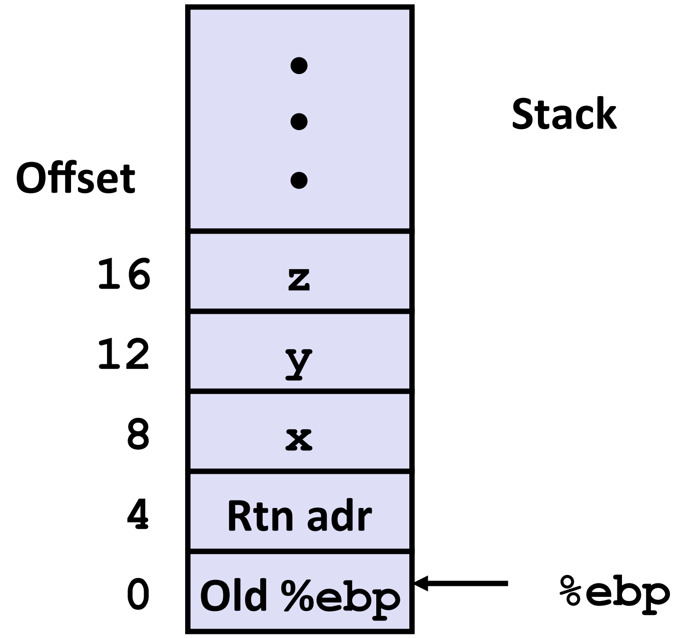
```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
movl  8(%ebp),%eax          # eax = x
movl  12(%ebp),%edx         # edx = y
leal  (%edx,%eax),%ecx      # ecx = x+y  (t1)
leal  (%edx,%edx,2),%edx    # edx = y + 2*y = 3*y
sall  $4,%edx               # edx = 48*y (t4)
addl  16(%ebp),%ecx         # ecx = z+t1 (t2)
leal  4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax             # eax = t5*t2 (rval)
```

# Understanding `arith`

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




```
movl  8(%ebp),%eax          # eax = x
movl  12(%ebp),%edx         # edx = y
leal  (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx    # edx = y + 2*y = 3*y
sall  $4,%edx               # edx = 48*y  (t4)
addl  16(%ebp),%ecx         # ecx = z+t1  (t2)
leal  4(%edx,%eax),%eax     # eax = 4+t4+x  (t5)
imull %ecx,%eax             # eax = t5*t2  (rval)
```

16x3=48

# Understanding `arith`

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

Stack

| Offset | |
|---|---|
| | • • • |
| 16 | z |
| 12 | y |
| 8 | x |
| 4 | Rtn adr |
| 0 | Old %ebp ← %ebp |

```
movl  8(%ebp),%eax           # eax = x
movl  12(%ebp),%edx          # edx = y
leal  (%edx,%eax),%ecx       # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx     # edx = y + 2*y = 3*y
sall  $4,%edx                # edx = 48*y  (t4)
addl  16(%ebp),%ecx          # ecx = z+t1  (t2)
leal  4(%edx,%eax),%eax      # eax = 4+t4+x  (t5)
imull  %ecx,%eax             # eax = t5*t2  (rval)
```

# Observations about `arith`

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

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
- `(x+y+z)*(x+4+48*y)`

```
movl  8(%ebp),%eax         # eax = x
movl  12(%ebp),%edx        # edx = y
leal  (%edx,%eax),%ecx     # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx   # edx = y + 2*y = 3*y
sall  $4,%edx              # edx = 48*y  (t4)
addl  16(%ebp),%ecx        # ecx = z+t1  (t2)
leal  4(%edx,%eax),%eax    # eax = 4+t4+x  (t5)
imull %ecx,%eax            # eax = t5*t2  (rval)
```

# Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

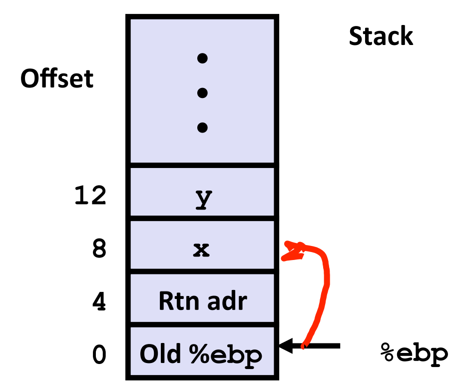
```
logical:
    pushl %ebp              # Set Up
    movl %esp,%ebp

    movl 8(%ebp),%eax       # Body
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp          # Finish
    popl %ebp
    ret
```

Set Up

Body

Finish

```
movl  8(%ebp),%eax      # eax = x
xorl  12(%ebp),%eax     # eax = x^y
sarl  $17,%eax          # eax = t1>>17
andl  $8185,%eax        # eax = t2 & 8185
```



# Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp              } Set
    movl %esp,%ebp          } Up

    movl 8(%ebp),%eax       }
    xorl 12(%ebp),%eax      }
    sarl $17,%eax           }
    andl $8185,%eax         } Body

    movl %ebp,%esp          }
    popl %ebp               } Finish
    ret                     }
```

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y       (t1)
sarl $17,%eax          eax = t1>>17    (t2)
andl $8185,%eax        eax = t2 & 8185
```

# Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp            } Set
    movl %esp,%ebp        } Up

    movl 8(%ebp),%eax     }
    xorl 12(%ebp),%eax    }
    sarl $17,%eax         }
    andl $8185,%eax       } Body

    movl %ebp,%esp        }
    popl %ebp             } Finish
    ret                   }
```

```
movl 8(%ebp),%eax     eax = x
xorl 12(%ebp),%eax    eax = x^y      (t1)
sarl $17,%eax         eax = t1>>17   (t2)
andl $8185,%eax       eax = t2 & 8185
```

# Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl  %esp,%ebp

    movl  8(%ebp),%eax
    xorl  12(%ebp),%eax
    sarl  $17,%eax
    andl  $8185,%eax

    movl  %ebp,%esp
    popl  %ebp
    ret
```

Set Up

Body

Finish

$2^{13} = 8192$, $2^{13} - 7 = 8185$

…0010000000000000, …0001111111111001

```
movl  8(%ebp),%eax      eax = x
xorl  12(%ebp),%eax     eax = x^y       (t1)
sarl  $17,%eax          eax = t1>>17    (t2)
andl  $8185,%eax        eax = t2 & 8185
```