Organisation und Architektur von Rechnern

Lecture 05

Instructor:

Reinhard v. Hanxleden

http://www.informatik.uni-kiel.de/rtsys/teaching/v-sysinf2

These slides are used with kind permission from the Carnegie Mellon University

The 5 Minute Review Session

1. **What is the default type of C constants?**
2. **What happens when signed and unsigned integers are mixed in a single expression?**
3. **What are the mathematical properties of modular addition?**
4. **What happens if the result of an integer multiplication is too large to be represented? How can this result in a security vulnerability?**
5. **What is the state of a computer visible to an assembly program?**

Last Time: Machine Programming, Basics

- **History of Intel processors and architectures**
- **C, assembly, machine code**
- **Assembly (IA32):**
  - Registers
  - Operands
  - Move (what's the `l` in `movl`?)

```
movl $0x4,%eax
movl %eax,%edx
movl (%eax),%edx
```

`%eax`
`%ecx`
`%edx`
`%ebx`
`%esi`
`%edi`
`%esp`
`%ebp`

Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

Complete Memory Addressing Modes

- **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 8 integer registers
  - Ri: Index register: Any, except for **`%esp`**
    - Unlikely you'd use **`%ebp`**, either
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**

  (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`

Expression	Address Computation	Address
`0x8(%edx)`	will disappear blackboard?	
`(%edx,%ecx)`		
`(%edx,%ecx,4)`		
`0x80(,%edx,2)`		

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x100`

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 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**

Today

- Complete addressing mode, address computation (`leal`)
- **Arithmetic operations**
- x86-64
- Control: Condition codes
- Conditional branches
- While loops

Some Arithmetic Operations

- **Two Operand 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*	

- **No distinction between signed and unsigned int (why?)**

Some Arithmetic Operations

- **One Operand Instructions**

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

- **See book for more instructions**

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;
}
```

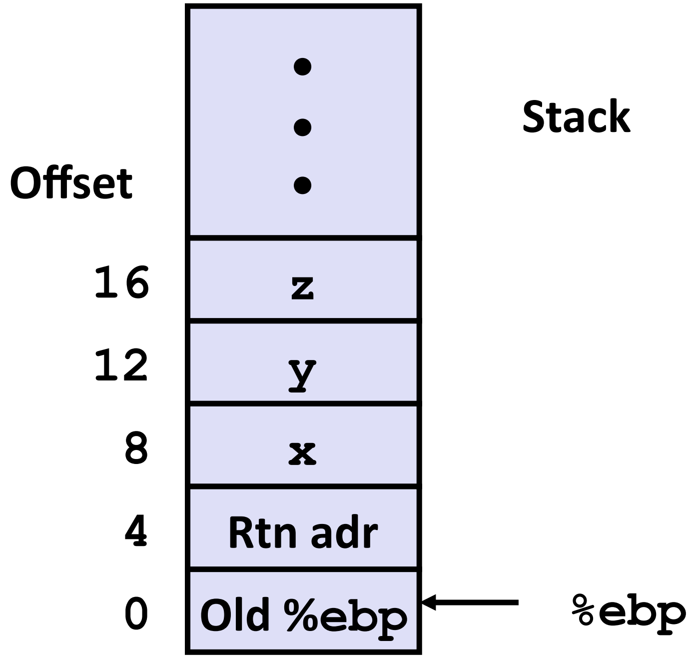
```
arith:
    pushl %ebp                  } Set
    movl %esp,%ebp              } Up

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

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

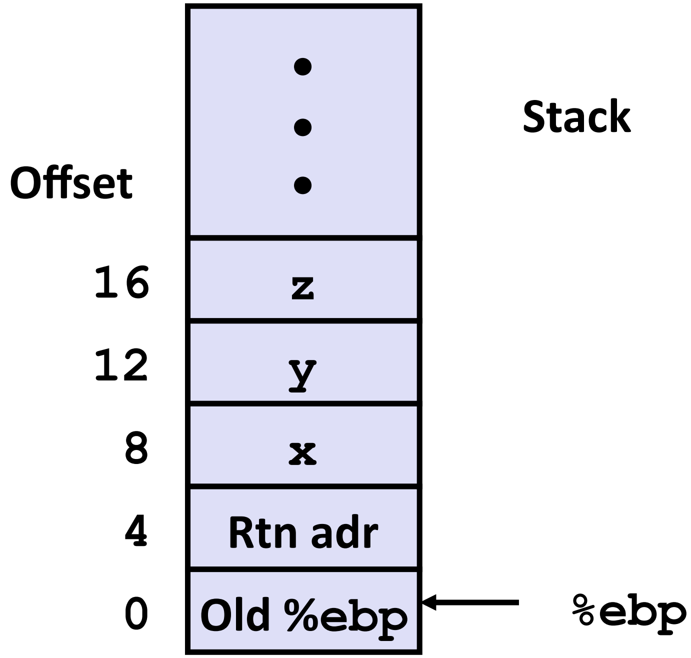



```
movl  8(%ebp),%eax
movl  12(%ebp),%edx
leal  (%edx,%eax),%ecx
leal  (%edx,%edx,2),%edx
sall  $4,%edx
addl  16(%ebp),%ecx
leal  4(%edx,%eax),%eax
imull  %ecx,%eax
```

will disappear
blackboard?

Understanding `arith`

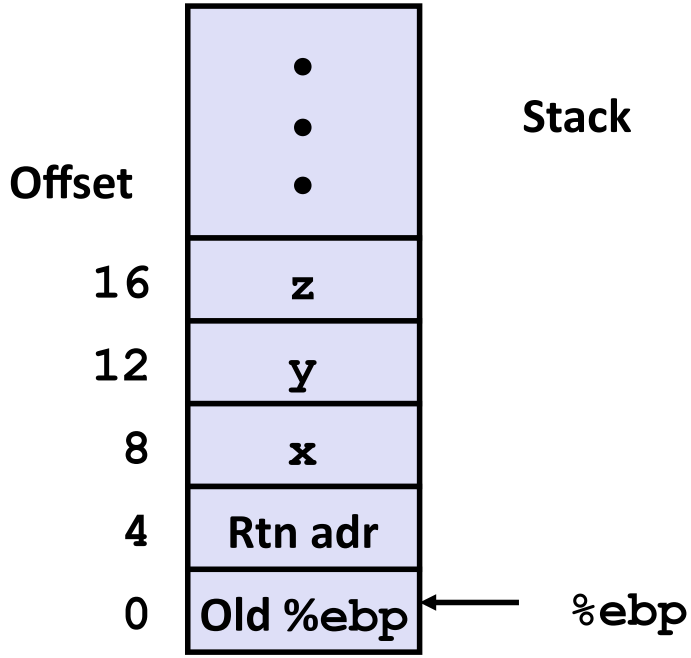
```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
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 = 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`

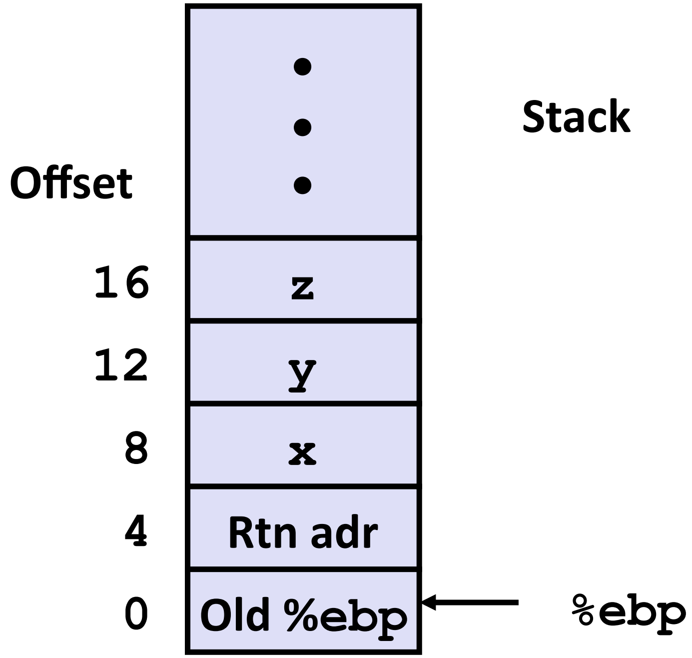
```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
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 = 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`

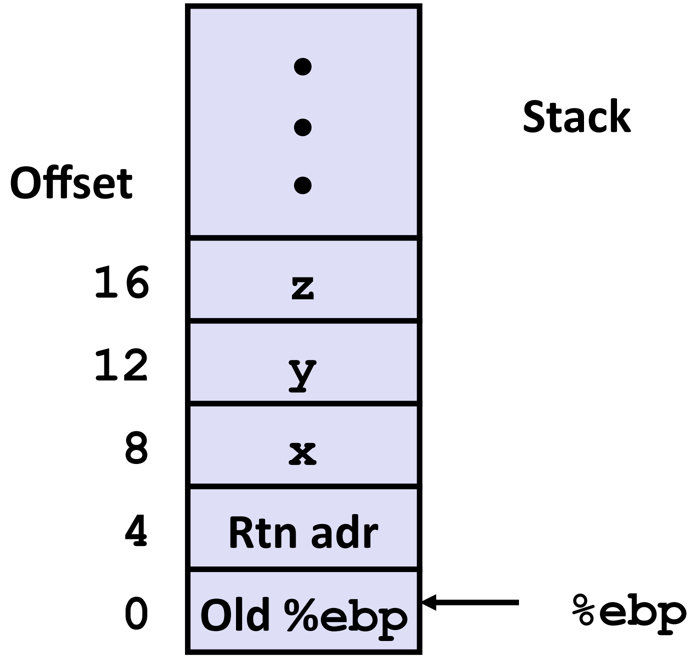
```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
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 = 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 = 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
    movl %esp,%ebp          } Up

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

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

```
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 (pushl, movl %esp,%ebp)

Body (movl 8(%ebp) … andl $8185)

Finish (movl %ebp,%esp … ret)

$2^{13} = 8192$, $2^{13} – 7 = 8185$

```
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
```

Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- **x86-64**
- Control: Condition codes
- Conditional branches
- While loops

Data Representations: IA32 + x86-64

- **Sizes of C Objects (in Bytes)**

C Data Type	*Typical 32-bit*	*x86-64*	*Intel IA32*
unsigned	4	4	4
int	4	4	4
long int	4	8	4
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	16	10/12
char *	4	8	4

Or any other pointer

x86-64 Integer Registers

`%rax`	`%eax`	`%r8`	`%r8d`
`%rbx`	`%ebx`	`%r9`	`%r9d`
`%rcx`	`%ecx`	`%r10`	`%r10d`
`%rdx`	`%edx`	`%r11`	`%r11d`
`%rsi`	`%esi`	`%r12`	`%r12d`
`%rdi`	`%edi`	`%r13`	`%r13d`
`%rsp`	`%esp`	`%r14`	`%r14d`
`%rbp`	`%ebp`	`%r15`	`%r15d`

- Extend existing registers. Add 8 new ones.
- Make **`%ebp/%rbp`** general purpose

Instructions

- **Long word `l` (4 Bytes) ↔ Quad word `q` (8 Bytes)**

- **New instructions:**
  - **`movl`** → **`movq`**
  - **`addl`** → **`addq`**
  - **`sall`** → **`salq`**
  - etc.

- **32-bit instructions that generate 32-bit results**
  - Set higher order bits of destination register to 0
  - Example: **`addl`**

Swap in 32-bit Mode

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Setup (pushl %ebp, movl %esp,%ebp, pushl %ebx)

Body (movl 12(%ebp),%ecx through movl %ebx,(%ecx))

Finish (movl -4(%ebp),%ebx through ret)

Swap in 64-bit Mode

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movl    (%rdi), %edx
    movl    (%rsi), %eax
    movl    %eax, (%rdi)
    movl    %edx, (%rsi)
    retq
```

- **Operands passed in registers (why useful?)**
  - First (**`xp`**) in **`%rdi`**, second (**`yp`**) in **`%rsi`**
  - 64-bit pointers
- **No stack operations required**
- **32-bit data**
  - Data held in registers **`%eax`** and **`%edx`**
  - **`movl`** operation

Swap Long Ints in 64-bit Mode

```
void swap_l
  (long int *xp, long int *yp)
{
  long int t0 = *xp;
  long int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap_l:
   movq    (%rdi), %rdx
   movq    (%rsi), %rax
   movq    %rax, (%rdi)
   movq    %rdx, (%rsi)
   retq
```

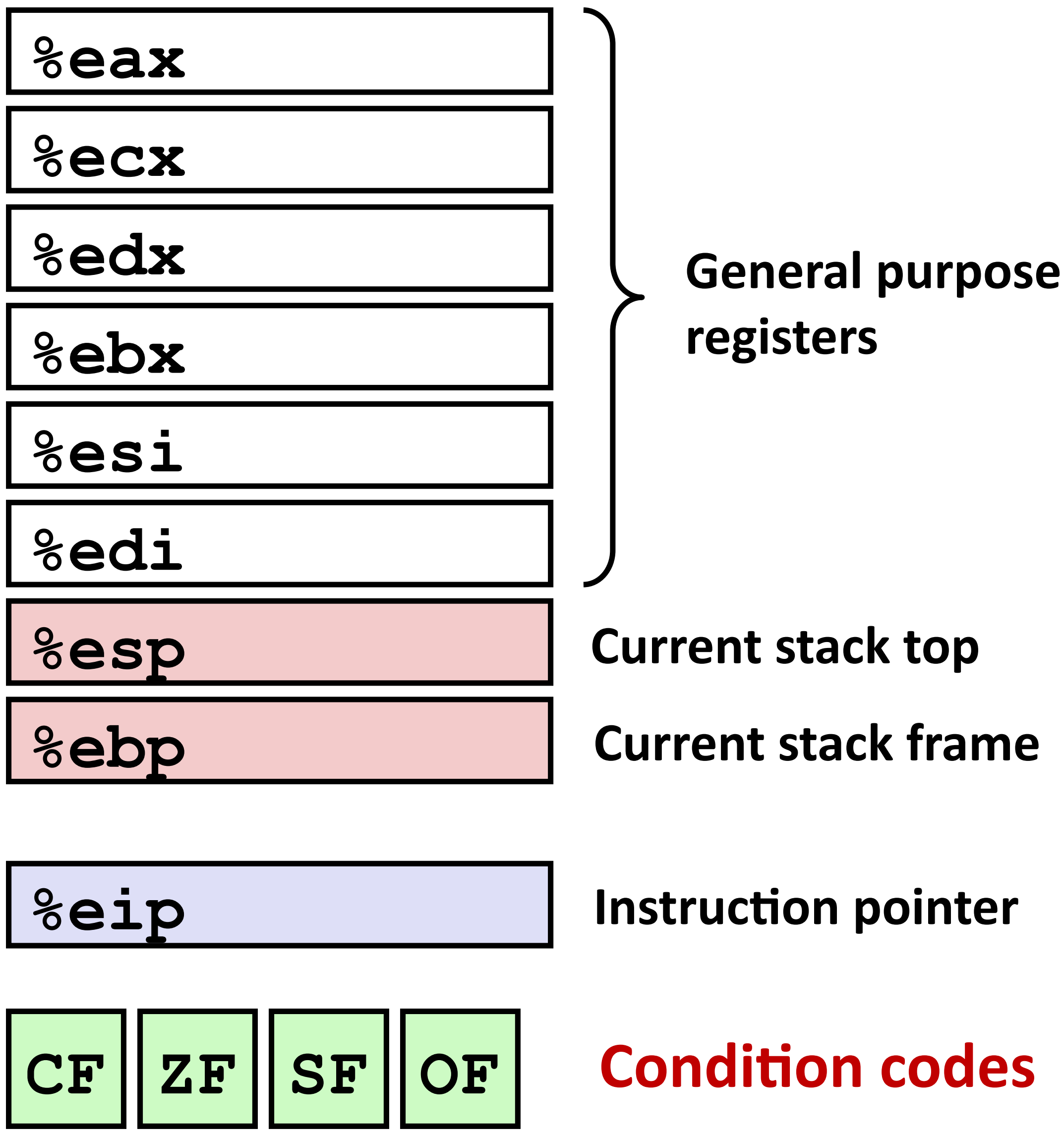
- **64-bit data**
  - Data held in registers `%rax` and `%rdx`
  - `movq` operation
  - "q" stands for quad-word

Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- **Control: Condition codes**
- Conditional branches
- While loops

Processor State (IA32, Partial)

- **Information about currently executing program**
  - Temporary data (`%eax`, …)
  - Location of runtime stack (`%ebp`,`%esp`)
  - Location of current code control point (`%eip`, …)
  - Status of recent tests (`CF`,`ZF`,`SF`,`OF`)



Condition Codes (Implicit Setting)

- **Single bit registers**

  `CF` Carry Flag (for unsigned)

  `ZF` Zero Flag

  `SF` Sign Flag (for signed)

  `OF` Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**

  Example: **`addl/addq`** *Src,Dest* ↔ **`t = a+b`**

  - **CF set** if carry out from most significant bit (unsigned overflow)
  - **ZF set** if `t == 0`
  - **SF set** if `t < 0` (as signed)
  - **OF set** if two's complement (signed) overflow
    `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- ***Not*** **set by `lea` instruction**
- **Full documentation (IA32), link also on course website**

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**

  `cmpl/cmpq` *Src2,Src1*

  `cmpl b,a` like computing **`a-b`** without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if **`a == b`**
  - **SF set** if **`(a-b) < 0`** (as signed)
  - **OF set** if two's complement (signed) overflow
    `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**

  `testl`/**`testq`** *Src2,Src1*

  `testl b,a` like computing **`a&b`** without setting destination

  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask

  - ZF set when **`a&b == 0`**
  - SF set when **`a&b < 0`**

Reading Condition Codes

- **SetX Instructions**
  - Set single byte based on combinations of condition codes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Reading Condition Codes (Cont.)

- **SetX Instructions:**
  Set single byte based on combination of condition codes
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use **`movzbl`** to finish job

```
int gt (int x, int y)
{
  return x > y;
}
```

%eax	%ah	%al
%ecx	%ch	%cl
%edx	%dh	%dl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

Body

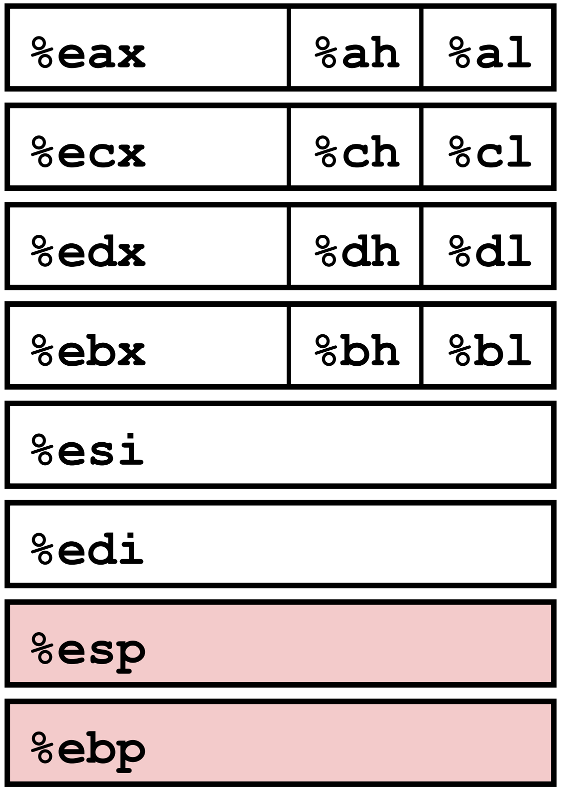
```
movl 12(%ebp),%eax
cmpl %eax,8(%ebp)
setg %al
movzbl %al,%eax
```

Will disappear Blackboard?

Reading Condition Codes (Cont.)

- **SetX Instructions:**
  Set single byte based on combination of condition codes
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use **`movzbl`** to finish job

```
int gt (int x, int y)
{
  return x > y;
}
```



Body

```
movl 12(%ebp),%eax       # eax = y
cmpl %eax,8(%ebp)        # Compare x and y
setg %al                 # al = x > y
movzbl %al,%eax          # Zero rest of %eax
```

Note inverted ordering! (refers to `cmpl %eax,8(%ebp)`)

Reading Condition Codes: x86-64

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

```
int gt (long x, long y)
{
  return x > y;
}
```

```
long lgt (long x, long y)
{
  return x > y;
}
```

Body (same for both)

```
xorl %eax, %eax
cmpq %rsi, %rdi
setg %al
```

Will disappear
Blackboard?

Reading Condition Codes: x86-64

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

```
int gt (long x, long y)
{
  return x > y;
}
```

```
long lgt (long x, long y)
{
  return x > y;
}
```

Body (same for both)

```
xorl %eax, %eax          # eax = 0
cmpq %rsi, %rdi          # Compare x and y
setg %al                 # al = x > y
```

Is `%rax` zero?

Yes: 32-bit instructions set high order 32 bits to 0!

Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	**Unconditional**
`je`	`ZF`	**Equal / Zero**
`jne`	`~ZF`	**Not Equal / Not Zero**
`js`	`SF`	**Negative**
`jns`	`~SF`	**Nonnegative**
`jg`	`~(SF^OF)&~ZF`	**Greater (Signed)**
`jge`	`~(SF^OF)`	**Greater or Equal (Signed)**
`jl`	`(SF^OF)`	**Less (Signed)**
`jle`	`(SF^OF)|ZF`	**Less or Equal (Signed)**
`ja`	`~CF&~ZF`	**Above (unsigned)**
`jb`	`CF`	**Below (unsigned)**

Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- **Conditional branches**
- While loops

Conditional Branch Example

```
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff:
    pushl   %ebp                    Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx           Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave                           Finish
    ret
.L7:
    subl    %edx, %eax              Body2
    jmp     .L8
```

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

- **C allows “goto” as means of transferring control**
  - Closer to machine-level programming style
- **Generally considered bad coding style**

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

General Conditional Expression Translation

C Code

```
val = Test ? Then-Expr : Else-Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
   nt = !Test;
   if (nt) goto Else;
   val = Then-Expr;
Done:
   . . .
Else:
   val = Else-Expr;
   goto Done;
```

- *Test* is expression returning integer
  - = 0 interpreted as false
  - ≠0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

Conditionals: x86-64

```
int absdiff(
    int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff: # x in %edi, y in %esi
  movl   %edi, %eax
  movl   %esi, %edx
  subl   %esi, %eax
  subl   %edi, %edx
  cmpl   %esi, %edi
  cmovle %edx, %eax
  ret
```

Will disappear
Blackboard?

Conditionals: x86-64

```
int absdiff(
    int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff: # x in %edi, y in %esi
  movl   %edi, %eax  # eax = x
  movl   %esi, %edx  # edx = y
  subl   %esi, %eax  # eax = x-y
  subl   %edi, %edx  # edx = y-x
  cmpl   %esi, %edi  # x:y
  cmovle %edx, %eax  # eax=edx if <=
  ret
```

- **Conditional move instruction**
  - **cmov*C*** src, dest
  - Move value from src to dest if condition *C* holds
  - More efficient than conditional branching (simple control flow)
  - But overhead: both branches are evaluated

General Form with Conditional Move

C Code

```
val = Test ? Then-Expr : Else-Expr;
```

Conditional Move Version

```
val1  = Then-Expr;
val2  = Else-Expr;
val1  = val2 if !Test;
```

- **Both values get computed**
- **Overwrite then-value with else-value if condition doesn't hold**
- **Don't use when:**
  - Then or else expression have side effects
  - Then and else expression are to expensive

Today

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- **While loops**

"Do-While" Loop Example

C Code

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);

  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Use backward branch to continue looping
- Only take branch when "while" condition holds

"Do-While" Loop Compilation

Registers:

`%edx`	`x`
`%eax`	`result`

Goto Version

```
int
fact_goto(int x)
{
  int result = 1;


loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

Assembly

```
fact_goto:
  pushl %ebp
  movl %esp,%ebp
  movl $1,%eax
  movl 8(%ebp),%edx

.L11:
  imull %edx,%eax
  decl %edx
  cmpl $1,%edx
  jg .L11

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

Will disappear Blackboard?

"Do-While" Loop Compilation

Registers:

%edx	x
%eax	result

Goto Version

```
int
fact_goto(int x)
{
  int result = 1;


loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

Assembly

```
fact_goto:
  pushl %ebp             # Setup
  movl %esp,%ebp         # Setup
  movl $1,%eax           # eax = 1
  movl 8(%ebp),%edx      # edx = x

.L11:
  imull %edx,%eax        # result *= x
  decl %edx              # x--
  cmpl $1,%edx           # Compare x : 1
  jg .L11                # if > goto loop

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

General "Do-While" Translation

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- ***Body:*** {

  $Statement_1$;
  $Statement_2$;
  …
  $Statement_n$;

  }

- ***Test*** **returns integer**

  = 0 interpreted as false

  ≠0 interpreted as true

"While" Loop Example

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {


    result *= x;
    x = x-1;
  };

  return result;
}
```

Goto Version #1

```
int fact_while_goto(int x)
{
  int result = 1;
loop:
  if (!(x > 1))
    goto done;
  result *= x;
  x = x-1;
  goto loop;
done:
  return result;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails

Alternative "While" Loop Translation

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test

Goto Version #2

```
int fact_while_goto2(int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

General "While" Translation

While version

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

New Style "While" Loop Translation

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

Goto Version

```
int fact_while_goto3(int x)
{
  int result = 1;
  goto middle;
loop:
  result *= x;
  x = x-1;
middle:
  if (x > 1)
    goto loop;
  return result;
}
```

- **Recent technique for GCC**
  - Both IA32 & x86-64
- **First iteration jumps over body computation within loop**

Jump-to-Middle While Translation

C Code

```
while (Test)
  Body
```

- Avoids duplicating test code
- Unconditional `goto` incurs no significant performance penalty
- `for` loops compiled in similar fashion

Goto Version

```
goto middle;
loop:
  Body
middle:
  if (Test)
    goto loop;
```

Goto (Previous) Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

Jump-to-Middle Example

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x--;
  };
  return result;
}
```

```
# x in %edx, result in %eax
  jmp   .L34          #   goto Middle
.L35:                 # Loop:
  imull %edx, %eax    #   result *= x
  decl  %edx          #   x--
.L34:                 # Middle:
  cmpl  $1, %edx      #   x:1
  jg    .L35          #   if >, goto Loop
```

Implementing Loops

- **IA32**
  - All loops translated into form based on "do-while"
- **x86-64**
  - Also make use of "jump to middle"
- **Why the difference**
  - IA32 compiler developed for machine where all operations costly
  - x86-64 compiler developed for machine where unconditional branches incur (almost) no overhead