Machine Programming II: Instructions

- Move instructions, registers, and operands
- Complete addressing mode, address computation (`leal`)
- Arithmetic operations (including some x86-64 instructions)
- Condition codes
- Control, unconditional and conditional branches
- While loops

Integer Registers (IA32)

	32-bit register	16-bit virtual register	High byte	Low byte	Origin (mostly obsolete)
general purpose	`%eax`	`%ax`	`%ah`	`%al`	*accumulate*
general purpose	`%ecx`	`%cx`	`%ch`	`%cl`	*counter*
general purpose	`%edx`	`%dx`	`%dh`	`%dl`	*data*
general purpose	`%ebx`	`%bx`	`%bh`	`%bl`	*base*
general purpose	`%esi`	`%si`			*source index*
general purpose	`%edi`	`%di`			*destination index*
	`%esp`	`%sp`			***stack pointer***
	`%ebp`	`%bp`			***base pointer***

16-bit virtual registers
(backwards compatibility)

Moving Data: IA32

- Moving Data
  - `movx` ***Source*, *Dest***
  - `x` is one of {`b, w, l`}

  - `movl` ***Source*, *Dest***:
    Move 4-byte "long word"
  - `movw` ***Source*, *Dest***:
    Move 2-byte "word"
  - `movb` ***Source*, *Dest***:
    Move 1-byte "byte"

- **Lots of these in typical code**

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

Moving Data: IA32

- Moving Data
  `movl` ***Source*, *Dest***:

- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: `$0x400`, `$-533`
    - Like C constant, but prefixed with `'$'`
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 8 integer registers
    - Example: `%eax, %edx`
    - But `%esp` and `%ebp` reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 4 consecutive bytes of memory at address given by register
    - Simplest example: `(%eax)`
    - Various other "address modes"

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

`movl` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address

```
movl (%ecx),%eax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movl 8(%ebp),%edx
```

Using Simple Addressing Modes

```
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
```

Set Up

Body

Finish

Using Simple Addressing Modes

```
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
```

Set Up

Body

Finish

Understanding Swap

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

Stack (in memory)

Offset		
	• • •	
12	yp	
8	xp	
4	Rtn adr	
0	Old %ebp	← %ebp
-4	Old %ebx	

Register	Value
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```


Understanding Swap

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap

Register	Value
%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset	Stack	Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```


Understanding Swap

Register	Value
%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset	Stack	Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```


Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	Offset	Value	Address
		456	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```


Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	Offset	Value	Address
		456	0x124
		123	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

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)`		
`(%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`

University of Washington

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*i
    - k = 1, 2, 4, or 8

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
    movl %esp,%ebp
```

Set Up

```
    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
```

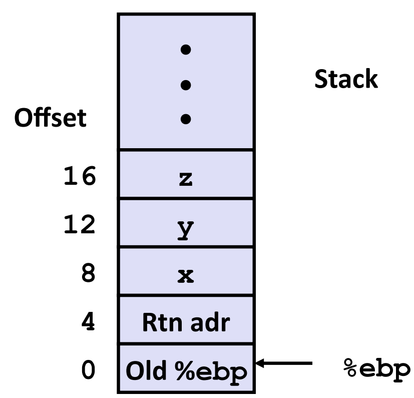
Body

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

Finish

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;
}
```

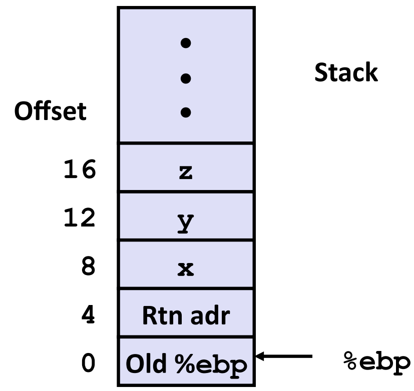



```
movl 8(%ebp),%eax
movl 12(%ebp),%edx
leal (%edx,%eax),%ecx
leal (%edx,%edx,2),%edx
sall $4,%edx
addl 16(%ebp),%ecx
leal 4(%edx,%eax),%eax
imull %ecx,%eax
```

What does each of these instructions mean?

Understanding `arith`

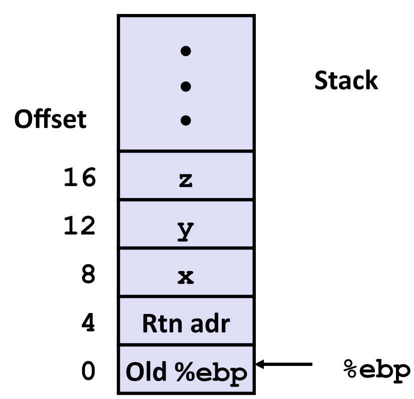
```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
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`

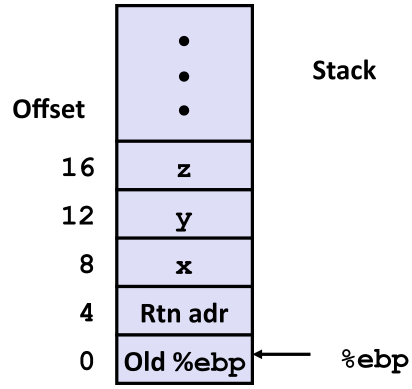
```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
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`

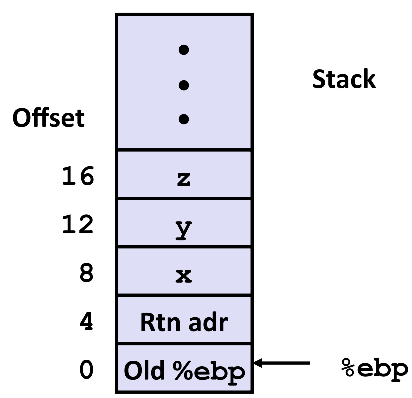
```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
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
    movl %esp,%ebp

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

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

$\left.\right\}$ Set Up

$\left.\right\}$ Body

$\left.\right\}$ 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
    movl %esp,%ebp

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

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

$\left.\right\}$ Set Up

$\left.\right\}$ Body

$\left.\right\}$ Finish

```
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 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
```

```
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;
}
```

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

```
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
```

```
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
```