x86 Programming
CS 740
Sept. 12, 2007

Topics
- Basics
- Accessing and Moving Data
- Arithmetic operations
- Control Flow
- Procedures
- Data Structures

x86 Processors

Ubiquitous in the desktop, laptop & server markets

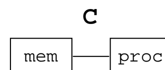
Instruction set has evolved over the past ~30 years
- 8086 (1978) was a 16-bit processor
- 386 (1985) extended to 32-bits with a flat address space
  - Capable of running UNIX/Linux
  - 32-bit Linux/gcc uses no instructions introduced in later models
- 64-bit extensions (2003-2004):
  - AMD's x86-64 and Intel's "Intel 64" are nearly identical
  - (not to be confused with Intel's IA-64 in the Itanium machines)

Constraints on the original x86 instruction set:
- Limited memory and silicon space
- Features to facilitate assembly-language programming
  - More recent (RISC) ISAs focus on compiler-generated code

Abstract Machines

Machine Model

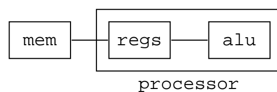
C



Data
1) char
2) int, float
3) double
4) struct, array
5) pointer

Control
1) loops
2) conditionals
3) goto
4) Proc. call
5) Proc. return

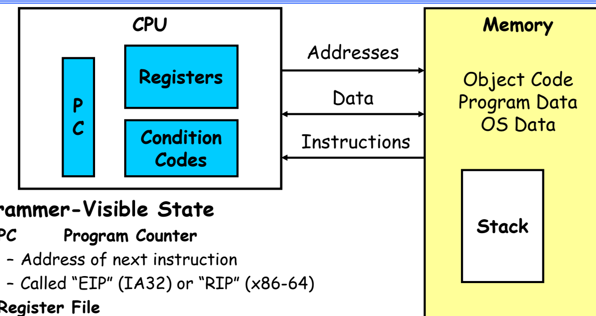
ASM



Data
1) byte
2) word
3) doubleword
4) contiguous word allocation
5) address of initial byte

Control
3) branch/jump
4) jump & link

Assembly Programmer's View

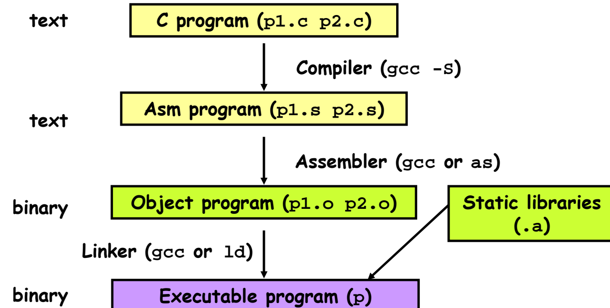


Programmer-Visible State
- **PC** Program Counter
  - Address of next instruction
  - Called "EIP" (IA32) or "RIP" (x86-64)
- **Register File**
  - Heavily used program data
- **Condition Codes**
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures

Translation Process

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated IA32 Assembly

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

`gcc -O -S code.c`

Produces file `code.s`

Object Code

Code for `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`

Assembler

- Translates `.s` into `.o`
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

Linker

- Resolves references between files
- Combines with static run-time libraries
  - E.g., code for `malloc`, `printf`
- Some libraries are *dynamically linked*
  - Linking occurs when program begins execution

Machine Instruction Example

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression:

`x += y`

Or

```
int eax;
int *ebp;
eax += ebp[2]
```

```
0x401046:    03 45 08
```

C Code

- Add two signed integers

Assembly

- Add 2 4-byte integers
  - "Long" words in GCC parlance
  - Same instruction whether signed or unsigned
- Operands:
  - `x`: Register `%eax`
  - `y`: Memory M[`%ebp+8`]
  - `t`: Register `%eax`
    - » Return function value in `%eax`

Object Code

- 3-byte instruction
- Stored at address `0x401046`

Disassembling Object Code

Disassembled

```
00401040 <_sum>:
   0:       55              push   %ebp
   1:       89 e5           mov    %esp,%ebp
   3:       8b 45 0c        mov    0xc(%ebp),%eax
   6:       03 45 08        add    0x8(%ebp),%eax
   9:       89 ec           mov    %ebp,%esp
   b:       5d              pop    %ebp
   c:       c3              ret
   d:       8d 76 00        lea    0x0(%esi),%esi
```

Disassembler

`objdump -d p`

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Disassembled

```
0x401040 <sum>:       push   %ebp
0x401041 <sum+1>:     mov    %esp,%ebp
0x401043 <sum+3>:     mov    0xc(%ebp),%eax
0x401046 <sum+6>:     add    0x8(%ebp),%eax
0x401049 <sum+9>:     mov    %ebp,%esp
0x40104b <sum+11>:    pop    %ebp
0x40104c <sum+12>:    ret
0x40104d <sum+13>:    lea    0x0(%esi),%esi
```

Within gdb Debugger

`gdb p`

`disassemble sum`

- Disassemble procedure

`x/13b sum`

- Examine the 13 bytes starting at `sum`

Moving Data: IA32

Moving Data

`movl` *Source,Dest*:

- Move 4-byte ("long") word
- Lots of these in typical code

Operand Types

- Immediate: Constant integer data
  - Like C constant, but prefixed with '$'
  - E.g., `$0x400`, `$-533`
  - Encoded with 1, 2, or 4 bytes
- Register: One of 8 integer registers
  - But `%esp` and `%ebp` reserved for special use
  - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
  - Various "address modes"

%eax
%edx
%ecx
%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 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
```

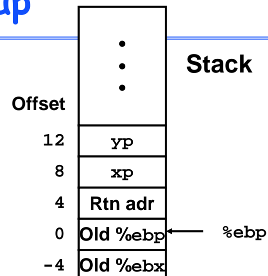

Using Simple Addressing Modes

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

```
swap:
   pushl %ebp            # Set Up
   movl  %esp,%ebp
   pushl %ebx

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

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


Understanding Swap

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



Register	Variable
%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
```


Indexed 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

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	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.

Some Arithmetic Operations

Format		Computation	
Two Operand Instructions			
`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*	

Some Arithmetic Operations

Format		Computation
One Operand Instructions		
`incl`	*Dest*	*Dest* = *Dest* + 1
`decl`	*Dest*	*Dest* = *Dest* - 1
`negl`	*Dest*	*Dest* = - *Dest*
`notl`	*Dest*	*Dest* = ~ *Dest*

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     } Body
   sall $4,%edx                }
   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;
}
```

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 = 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          } Body
   sarl $17,%eax               }
   andl $8185,%eax             }

   movl %ebp,%esp              }
   popl %ebp                   } Finish
   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 (rval)
```


Data Representations: IA32 + x86-64

Sizes of C Objects (in Bytes)

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

» Or any other pointer

x86-64 General Purpose Registers

%rax	%eax	%r8	%r8d
%rdx	%edx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rbx	%ebx	%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**

Swap in 32-bit Mode (Review)

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

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

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

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

- **Operands passed in registers**
  - 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)
   ret
```

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

Condition Codes

Single Bit Registers

CF	Carry Flag	SF	Sign Flag
ZF	Zero Flag	OF	Overflow Flag

Implicitly Set By Arithmetic Operations

addl *Src,Dest* addq *Src,Dest*

C analog: `t = a + b` (a = Src, b = Dest)

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

Not **set by `lea`, `inc`, or `dec` instructions**

Setting Condition Codes (cont.)

Explicit Setting by Compare Instruction

cmpl *Src2,Src1* 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`
- OF set if two's complement overflow
  - `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Setting Condition Codes (cont.)

Explicit Setting by Test instruction

testl *Src2,Src1*
testq *Src2,Src1*

- Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
- `testl b,a` like computing `a&b` without setting destination
- 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 combinations of condition codes
- One of 8 addressable byte registers
  - Embedded within first 4 integer 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
%edx	%dh	%dl
%ecx	%ch	%cl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

Body

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

Note inverted ordering!

Reading condition codes: x86-64

SetX Instructions

- Set single byte based on combinations of condition codes
  - Does not alter remaining 7 bytes

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

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

- x86-64 arguments
  - x in %rdi
  - y in %rsi

Body (same for both)

(32-bit instructions set high order 32 bits to 0)

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


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)

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

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  # v   = x
  movl   %esi, %edx  # ve  = y
  subl   %esi, %eax  # v  -= y
  subl   %edi, %edx  # ve -= x
  cmpl   %esi, %edi  # x:y
  cmovle %edx, %eax  # v=ve 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 & predictable control flow

General Form with Conditional Move

C Code

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

Conditional Move Version

```
val  = Then-Expr;
vale = Else-Expr;
val  = vale if !Test;
```

- Both values get computed
- Overwrite then-value with else-value if condition doesn't hold

Limitations of Conditional Move

```
val  = Then-Expr;
vale = Else-Expr;
val  = vale if !Test;
```

```
int xgty  = 0, xltey = 0;

int absdiff_se(
    int x, int y)
{
    int result;
    if (x > y) {
        xgty++;  result = x-y;
    } else {
        xltey++; result = y-x;
    }
    return result;
}
```

Don't use when:

- Then-Expr or Else-Expr has side effect
- Then-Expr or Else-Expr requires significant computation

"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            # 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
```


"While" Loop Translation

C Code

```
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:
```


"For"→ "While"→ "Do-While"

For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update ;
}
```

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

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


Switch Statements

Implementation Options

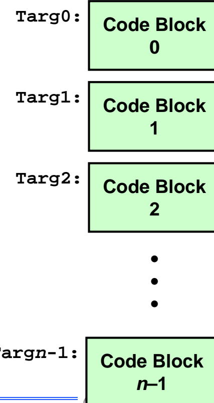
- **Series of conditionals**
  - Organize in tree structure
  - Logarithmic performance
- **Jump Table**
  - Lookup branch target
  - Constant time
  - Possible when cases are small integer constants
- ***GCC***
  - Picks one based on case structure

Jump Table Structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
  • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

Targn-1: Code Block n–1

Approx. Translation

```
target = JTab[x];
goto *target;
```


Switch Statement Example

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
   pushl %ebp            # Setup
   movl  %esp, %ebp      # Setup
   pushl %ebx            # Setup
   movl  $1, %ebx        # w = 1
   movl  8(%ebp), %edx   # edx = x
   movl  16(%ebp), %ecx  # ecx = z
   cmpl  $6, %edx        # x:6
   ja    .L61            # if > goto default
   jmp   *.L62(,%edx,4)  # goto JTab[x]
```


Procedure Calls

x86 (IA32):

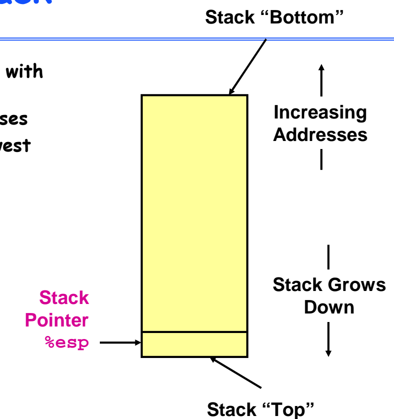
- stack discipline

X86-64:

- argument passing in registers

IA32 Stack

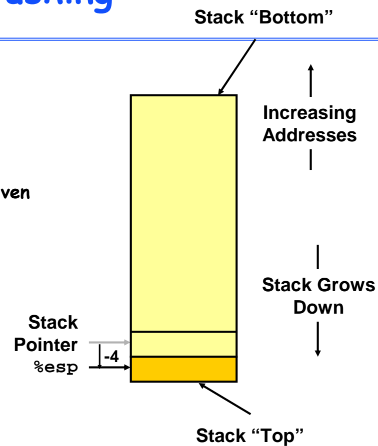
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of top element



IA32 Stack Pushing

Pushing

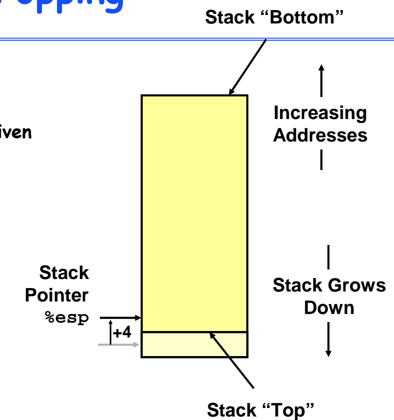
- `pushl` *Src*
- Fetch operand at *Src*
- Decrement `%esp` by 4
- Write operand at address given by `%esp`



IA32 Stack Popping

Popping

- `popl` *Dest*
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to *Dest*



Procedure Control Flow

- Use stack to support procedure call and return

Procedure call:

`call` *`label`* Push return address on stack; Jump to *`label`*

Return address value

- Address of instruction beyond `call`
- Example from disassembly

```
804854e:  e8 3d 06 00 00   call   8048b90 <main>
8048553:  50               pushl  %eax
```

  – Return address = 0x8048553

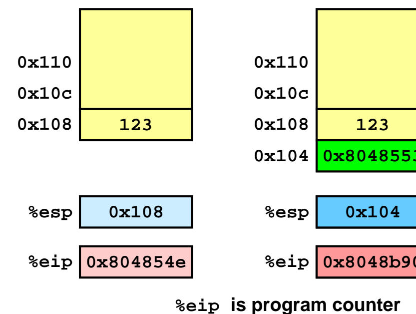
Procedure return:

- `ret` Pop address from stack; Jump to address

Procedure Call Example

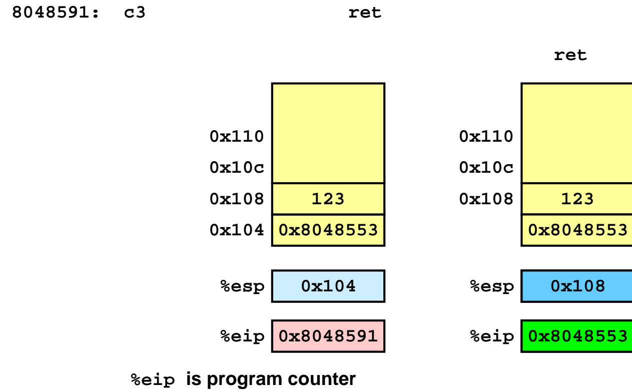
```
804854e:  e8 3d 06 00 00      call   8048b90 <main>
8048553:  50                  pushl  %eax
```

`call 8048b90`



`%eip` is program counter

Procedure Return Example



Stack-Based Languages

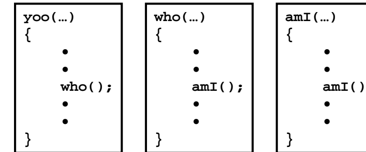
Languages that support recursion

- e.g., C, Pascal

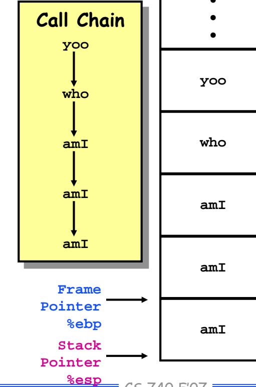
Stack Allocated in *Frames*

- state for procedure invocation
  - return point, arguments, locals

Code Example



Stack (grows down)



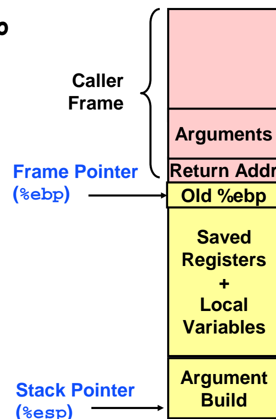
IA32/Linux Stack Frame

Current Stack Frame ("Top" to Bottom)

- Parameters for function about to call
  - "Argument build"
- Local variables
  - If can't keep in registers
- Saved register context
- Old frame pointer

Caller Stack Frame

- Return address
  - Pushed by `call` instruction
- Arguments for this call



Revisiting swap

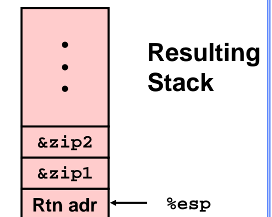
```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

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

Calling swap from call_swap

```
call_swap:
  • • •
  pushl $zip2   # Global Var
  pushl $zip1   # Global Var
  call swap
  • • •
```

Resulting Stack



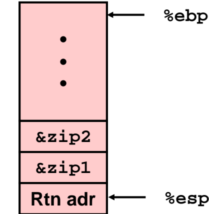
Revisiting swap

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

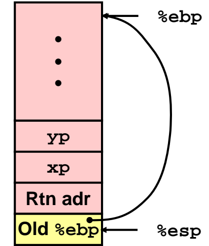
```
swap:
   pushl %ebp            } Set
   movl %esp,%ebp        } Up
   pushl %ebx

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

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


swap Setup #1

Entering Stack



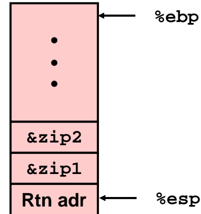
Resulting Stack



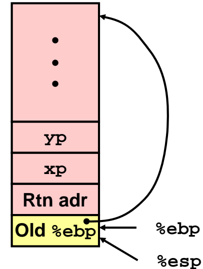

```
swap:
   pushl %ebp
   movl %esp,%ebp
   pushl %ebx
```


swap Setup #2

Entering Stack



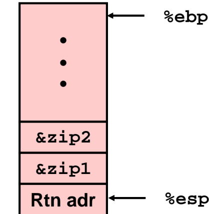
Resulting Stack



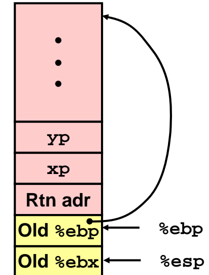

```
swap:
   pushl %ebp
   movl %esp,%ebp
   pushl %ebx
```


swap Setup #3

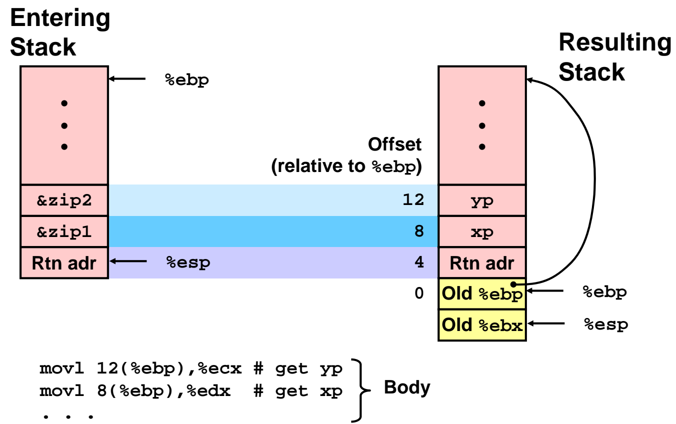
Entering Stack

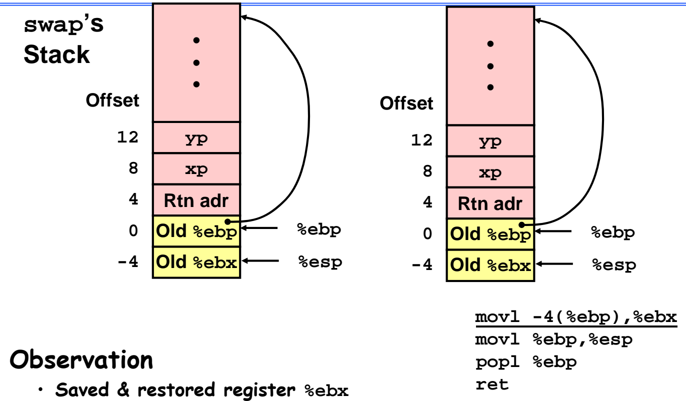


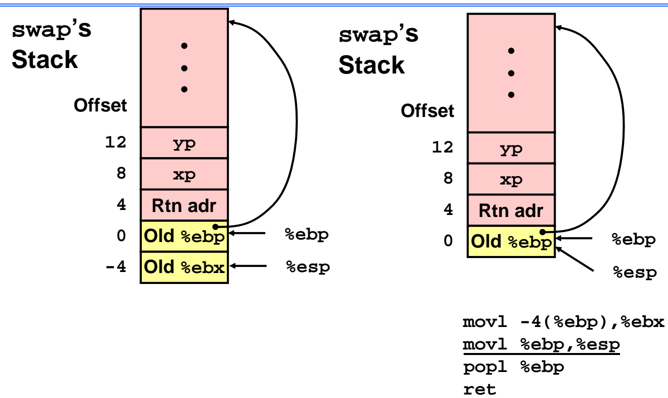
Resulting Stack

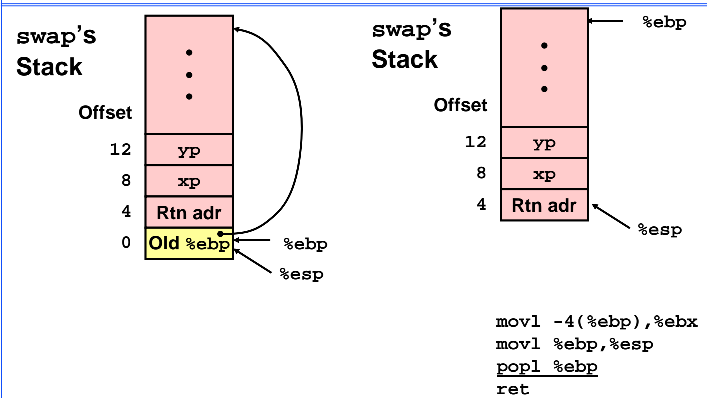



```
swap:
   pushl %ebp
   movl %esp,%ebp
   pushl %ebx
```

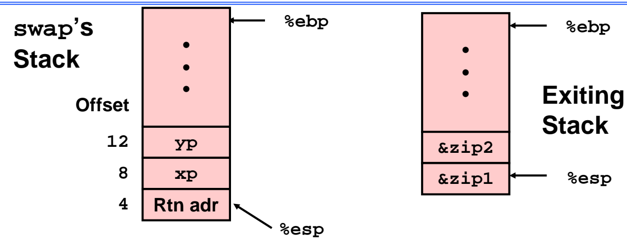

Effect of swap Setup
Entering Stack
Resulting Stack
%ebp
%esp
Offset (relative to %ebp)
&zip2
&zip1
Rtn adr
12
8
4
0
yp
xp
Old %ebp
Old %ebx
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx # get xp
. . .
Body
- 61 -
CS 740 F'07


swap Finish #1
swap's Stack
Offset
12
8
4
0
-4
yp
xp
Rtn adr
Old %ebp
Old %ebx
%ebp
%esp
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
Observation
Saved & restored register %ebx
- 62 -
CS 740 F'07


swap Finish #2
swap's Stack
Offset
12
8
4
0
-4
yp
xp
Rtn adr
Old %ebp
Old %ebx
%ebp
%esp
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
- 63 -
CS 740 F'07


swap Finish #3
swap's Stack
Offset
12
8
4
0
yp
xp
Rtn adr
Old %ebp
%ebp
%esp
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
- 64 -
CS 740 F'07

swap Finish #4




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

Observation

- Saved & restored register `%ebx`
- Didn't do so for `%eax`, `%ecx`, or `%edx`

Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the *caller*, `who` is the *callee*

Can Register be Used for Temporary Storage?

```
yoo:
   • • •
   movl $15213, %edx
   call who
   addl %edx, %eax
   • • •
   ret
```

```
who:
   • • •
   movl 8(%ebp), %edx
   addl $91125, %edx
   • • •
   ret
```

- Contents of register `%edx` overwritten by `who`

Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the *caller*, `who` is the *callee*

Can Register be Used for Temporary Storage?

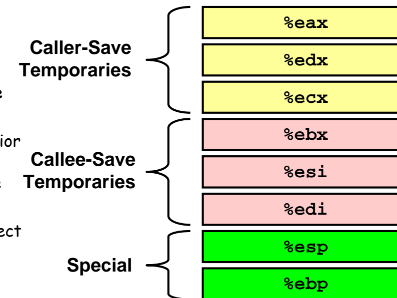
Conventions

- "Caller Save"
  - Caller saves temporary in its frame before calling
- "Callee Save"
  - Callee saves temporary in its frame before using

IA32/Linux Register Usage

Integer Registers

- Two have special uses
  - `%ebp`, `%esp`
- Three managed as callee-save
  - `%ebx`, `%esi`, `%edi`
  - Old values saved on stack prior to using
- Three managed as caller-save
  - `%eax`, `%edx`, `%ecx`
  - Do what you please, but expect any callee to do so, as well
- Register `%eax` also stores returned value



Recursive Factorial

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Registers

- %eax used without first saving
- %ebx used, but save at beginning & restore at end

x86-64 General Purpose 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

- Twice the number of registers
- Accessible as 8, 16, 32, or 64 bits

x86-64 Register Conventions

%rax	Return Value	%r8	Argument #5
%rbx	Callee Saved	%r9	Argument #6
%rcx	Argument #4	%r10	Callee Saved
%rdx	Argument #3	%r11	Used for linking
%rsi	Argument #2	%r12	C: Callee Saved
%rdi	Argument #1	%r13	Callee Saved
%rsp	Stack Pointer	%r14	Callee Saved
%rbp	Callee Saved	%r15	Callee Saved

x86-64 Registers

Arguments passed to functions via registers

- If more than 6 integral parameters, then pass rest on stack
- These registers can be used as caller-saved as well

All References to Stack Frame via Stack Pointer

- Eliminates need to update %ebp

Other Registers

- 6+1 callee saved
- 2 or 3 have special uses

Basic Data Types

Integral
- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

Floating Point
- Stored & operated on in floating point registers

Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12/16	long double

Array Accessing Example

Computation
- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at 4*%eax + %edx
- Use memory reference (%edx,%eax,4)

```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

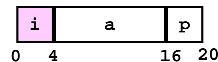
IA32 Memory Reference Code

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```


Structures

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout



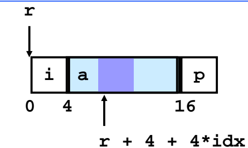

```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

IA32 Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```


Generating Pointer to Struct. Member

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



Generating Pointer to Array Element
- Offset of each structure member determined at compile time

```
int *
find_a
  (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

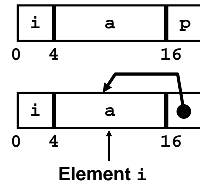
```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```

Structure Referencing (Cont.)

C Code

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```




```
 # %edx = r
movl (%edx),%ecx        # r->i
leal 0(,%ecx,4),%eax    # 4*(r->i)
leal 4(%edx,%eax),%eax  # r+4+4*(r->i)
movl %eax,16(%edx)      # Update r->p
```