MACHINE LEVEL REPRESENTATION 2

CAS CS 210 Computer Systems
Based on CMU Slides
CS:APP2e Ch 3.5-3.7

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 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

ARITHMETIC OPERATIONS

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

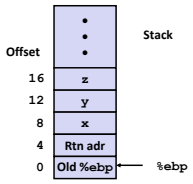
```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

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

    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

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

Set Up — Body — Finish

Understanding `arith`

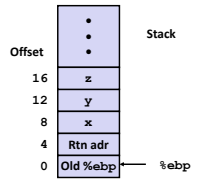
```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
movl 8(%ebp),%eax       # 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`

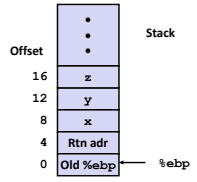
```
int arith
   (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
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`

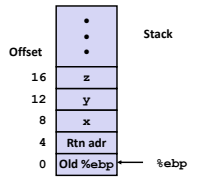
```
int arith
   (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
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           }
    andl $8185,%eax         } Body

    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              } 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                     }
```

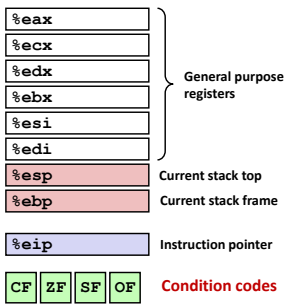
$2^{13} = 8192$, $2^{13} - 7 = 8185$

```
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
```

CONTROL: CONDITION CODES

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`)



General purpose registers

Current stack top

Current stack frame

Instruction pointer

Condition codes

Condition Codes (Implicit Setting)

- **Single bit registers**
  - `CF` Carry Flag (for unsigned)
  - `SF` Sign Flag (for signed)
  - `ZF` Zero Flag
  - `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     # 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!

Conditional Move (cmov*C*):

```
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

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                    } 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;
}
```

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

```
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
```

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

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

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
```

Registers:

%edx	x
%eax	result

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;
}
```

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;
}
```

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

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 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

FOR LOOPS

"For" Loop Example

```
int result;
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

General Form

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

Test	*Init*	*Update*	*Body*
`p != 0`	`result = 1`	`p = p >> 1`	`{ if (p & 0x1) result *= x; x = x*x; }`

"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:
```

For-Loop: Compilation #1

For Version

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

```
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

Goto Version

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

```
  result = 1;
  if (p == 0)
    goto done;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
  if (p != 0)
    goto loop;
done:
```

"For"→ "While" (Jump-to-Middle)

For Version

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

While Version

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

Goto Version

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

For-Loop: Compilation #2

For Version

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

```
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

Goto Version

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

```
  result = 1;
goto middle;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
middle:
  if (p != 0)
    goto loop;
done:
```

SWITCH STATEMENTS

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch Statement Example

- **Multiple case labels**
  - Here: 5, 6
- **Fall through cases**
  - Here: 2
- **Missing cases**
  - Here: 4

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

jtab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

• • •

Targn-1: Code Block n–1

Approximate Translation

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

Switch Statement Example (IA32)

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

Jump table

```
.section .rodata
   .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

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]
```

Indirect jump →

Assembly Setup Explanation

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at `.L62`
- **Jumping**

  Direct: `jmp .L61`
  - Jump target is denoted by label `.L61`

  Indirect: `jmp *.L62(,%edx,4)`
  - Start of jump table: `.L62`
  - Must scale by factor of 4 (labels have 32-bit = 4 Bytes on IA32)
  - Fetch target from effective Address `.L62 + edx*4`
    - Only for $0 \le x \le 6$

Jump table

```
.section .rodata
   .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

Jump Table

Jump table

```
.section .rodata
   .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

```
   switch(x) {
   case 1:      // .L56
       w = y*z;
       break;
   case 2:      // .L57
       w = y/z;
       /* Fall Through */
   case 3:      // .L58
       w += z;
       break;
   case 5:
   case 6:      // .L60
       w -= z;
       break;
   default:     // .L61
       w = 2;
   }
```

Code Blocks (Partial)

```
  switch(x) {
    . . .
  case 2:      // .L57
      w = y/z;
      /* Fall Through */
  case 3:      // .L58
      w += z;
      break;
    . . .
  default:     // .L61
      w = 2;
  }
```

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   movl  %ebx, %eax   # Return w
   popl  %ebx
   leave
   ret
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd                  # Div prep
   idivl %ecx            # y/z
   movl  %eax, %ebx # w = y/z
# Fall through
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   movl  %ebx, %eax # Return w
   popl  %ebx
   leave
   ret
```

IA32 Object Code

- **Setup**
  - Label **.L61** becomes address **0x8048630**
  - Label **.L62** becomes address **0x80488dc**

Assembly Code

```
switch_eg:
  . . .
   ja    .L61            # if > goto default
   jmp   *.L62(,%edx,4)  # goto JTab[x]
```

Disassembled Object Code

```
08048610 <switch_eg>:
 . . .
 8048622:  77 0c                  ja     8048630
 8048624:  ff 24 95 dc 88 04 08   jmp    *0x80488dc(,%edx,4)
```

IA32 Object Code (cont.)

- **Jump Table**
  - Doesn't show up in disassembled code
  - Can inspect using GDB

```
gdb asm-cntl
(gdb) x/7xw 0x80488dc
```

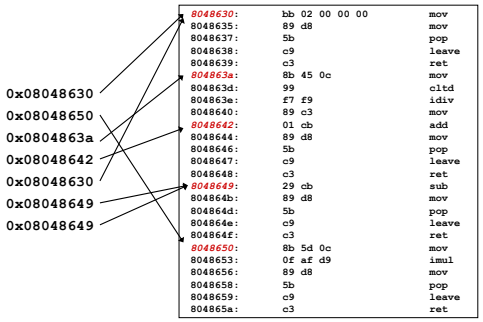
    - Examine 7 hexadecimal format "words" (4-bytes each)
    - Use command "`help x`" to get format documentation

```
0x80488dc:
  0x08048630
  0x08048650
  0x0804863a
  0x08048642
  0x08048630
  0x08048649
  0x08048649
```

Disassembled Targets

```
8048630:    bb 02 00 00 00          mov    $0x2,%ebx
8048635:    89 d8                   mov    %ebx,%eax
8048637:    5b                      pop    %ebx
8048638:    c9                      leave
8048639:    c3                      ret
804863a:    8b 45 0c                mov    0xc(%ebp),%eax
804863d:    99                      cltd
804863e:    f7 f9                   idiv   %ecx
8048640:    89 c3                   mov    %eax,%ebx
8048642:    01 cb                   add    %ecx,%ebx
8048644:    89 d8                   mov    %ebx,%eax
8048646:    5b                      pop    %ebx
8048647:    c9                      leave
8048648:    c3                      ret
8048649:    29 cb                   sub    %ecx,%ebx
804864b:    89 d8                   mov    %ebx,%eax
804864d:    5b                      pop    %ebx
804864e:    c9                      leave
804864f:    c3                      ret
8048650:    8b 5d 0c                mov    0xc(%ebp),%ebx
8048653:    0f af d9                imul   %ecx,%ebx
8048656:    89 d8                   mov    %ebx,%eax
8048658:    5b                      pop    %ebx
8048659:    c9                      leave
804865a:    c3                      ret
```

Matching Disassembled Targets

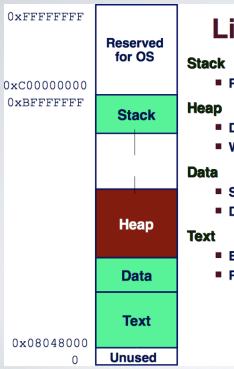



```
8048630:    bb 02 00 00 00          mov
8048635:    89 d8                   mov
8048637:    5b                      pop
8048638:    c9                      leave
8048639:    c3                      ret
804863a:    8b 45 0c                mov
804863d:    99                      cltd
804863e:    f7 f9                   idiv
8048640:    89 c3                   mov
8048642:    01 cb                   add
8048644:    89 d8                   mov
8048646:    5b                      pop
8048647:    c9                      leave
8048648:    c3                      ret
8048649:    29 cb                   sub
804864b:    89 d8                   mov
804864d:    5b                      pop
804864e:    c9                      leave
804864f:    c3                      ret
8048650:    8b 5d 0c                mov
8048653:    0f af d9                imul
8048656:    89 d8                   mov
8048658:    5b                      pop
8048659:    c9                      leave
804865a:    c3                      ret
```

Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch
- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler
  - Must generate assembly code to implement more complex control
- **Standard Techniques**
  - IA32 loops converted to do-while form
  - x86-64 loops use jump-to-middle
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (not shown)
- **Conditions in CISC**
  - CISC machines generally have condition code registers

PROCEDURES



Linux Memory Layout

Stack
- Runtime user stack

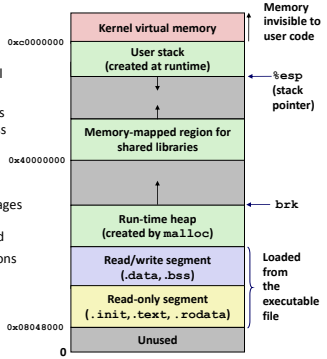
Heap
- Dynamically allocated storage
- When call `malloc`, `calloc`, `new` at run time

Data
- Statically allocated data
- Data declared in code

Text
- Executable machine instructions
- Read-only

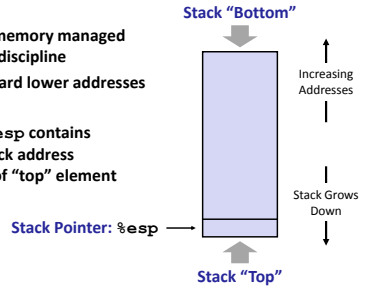
Simplifying Linking and Loading

- **Linking**
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address
- **Loading**
  - `execve()` allocates virtual pages for .text and .data sections = creates PTEs marked as invalid
  - The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system



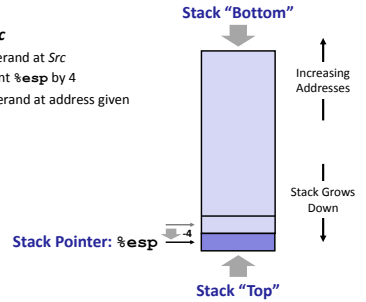
IA32 Stack

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



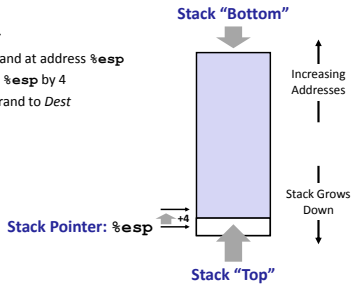
IA32 Stack: Push

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



IA32 Stack: Pop

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



Procedure Control Flow

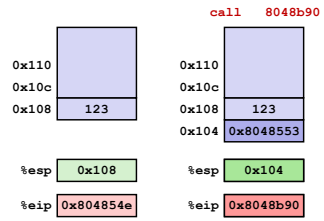
- **Use stack to support procedure call and return**
- **Procedure call:** `call` ***label***
  - Push return address on stack
  - Jump to ***label***
- **Return address:**
  - Address of instruction beyond `call`
  - Example from disassembly

```
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
```

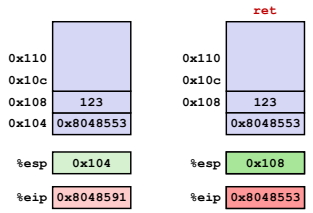


`call 8048b90`

`%eip`: program counter

Procedure Return Example

```
8048591:    c3                  ret
```



%eip: program counter

Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Pascal, Java
  - Code must be "*Reentrant*"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- **Stack discipline**
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- **Stack allocated in *Frames***
  - state for single procedure instantiation

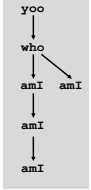
Call Chain Example

```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

```
who(...)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

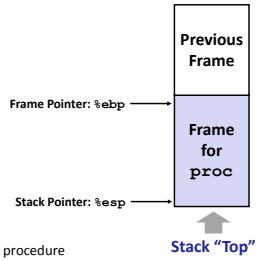
Example Call Chain



Procedure `amI` is recursive

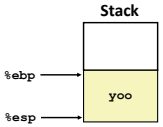
Stack Frames

- **Contents**
  - Local variables
  - Return information
  - Temporary space
- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code



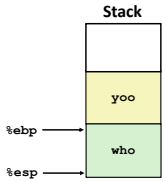
Example

```
yoo (...)
{
  •
  •
  who();
  •
  •
}
```

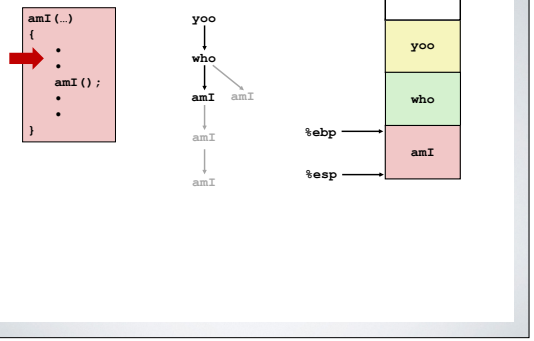


Example

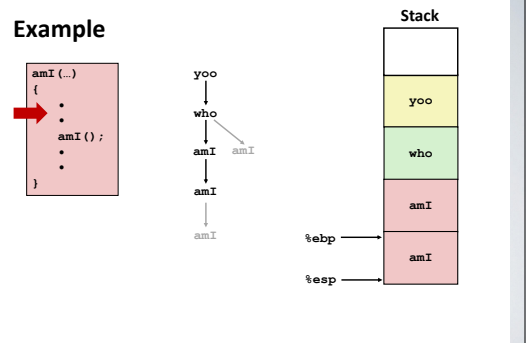
```
who (...)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```



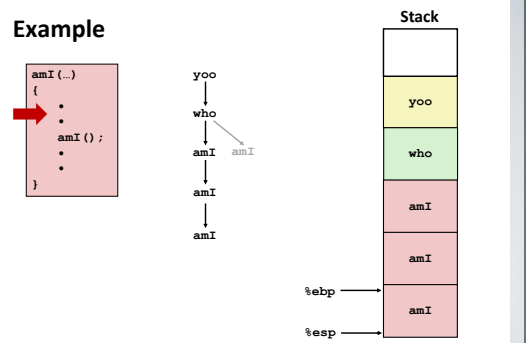
Example



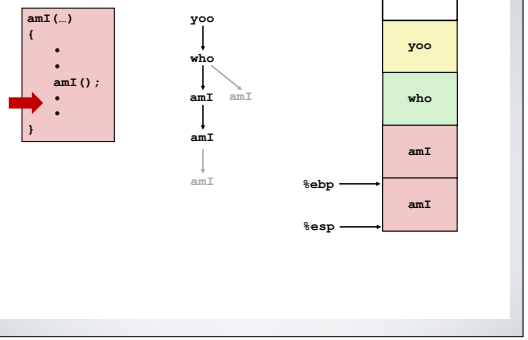
Example



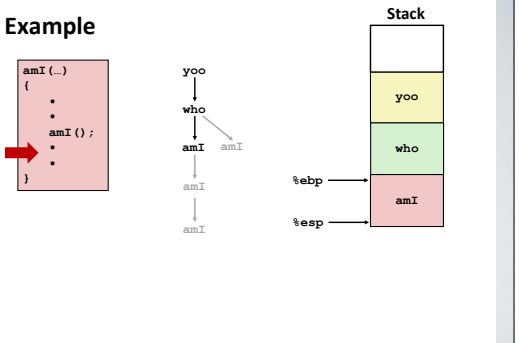
Example



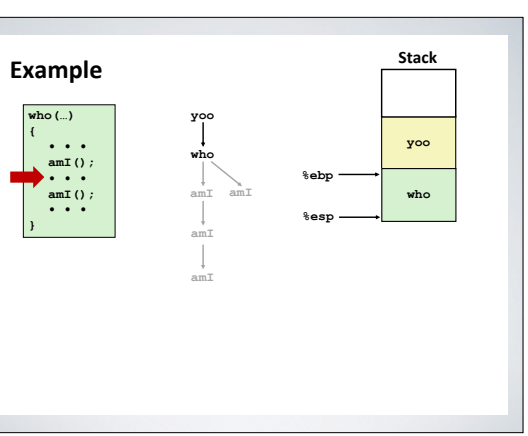
Example



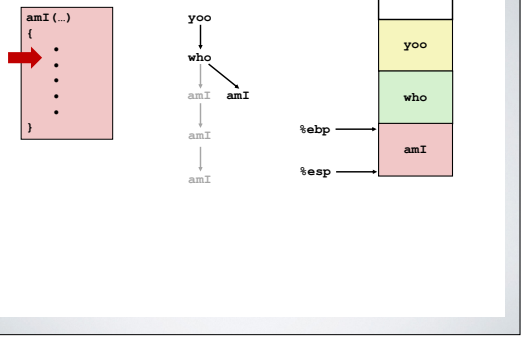
Example



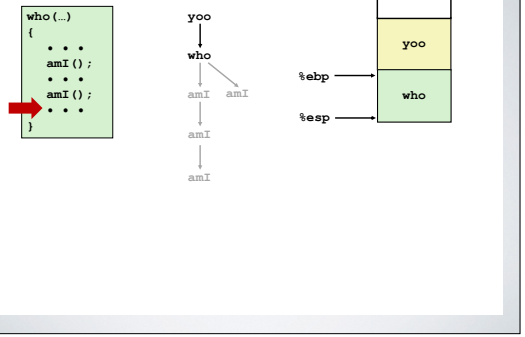
Example



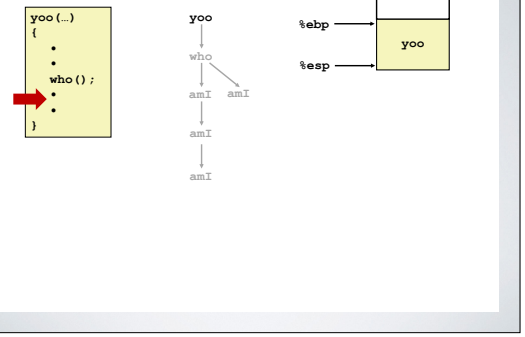
Example



Example

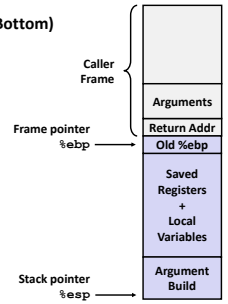


Example



IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:" Parameters for function about to call
  - 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



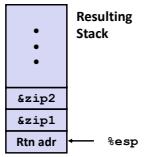
Revisiting `swap`

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

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

Calling `swap` from `call_swap`

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

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



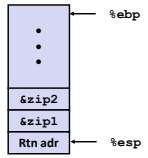
Revisiting `swap`

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

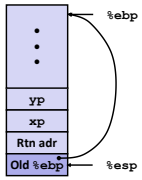
```
swap:
    pushl %ebp              } Set Up
    movl %esp,%ebp          }
    pushl %ebx              }

    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                     }
```

swap Setup #1

Entering Stack



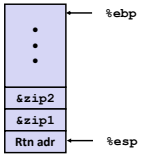
Resulting Stack

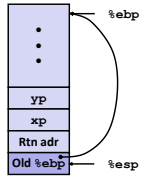



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #1

Entering Stack

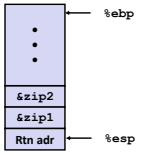




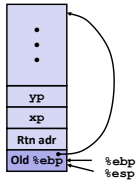

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #1

Entering Stack



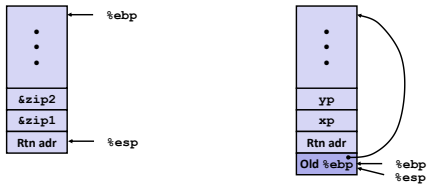
Resulting Stack




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

swap Setup #1

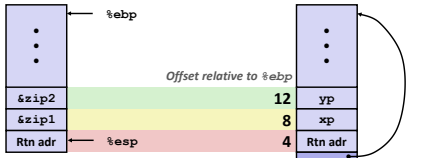
Entering Stack




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

swap Setup #1

Entering Stack

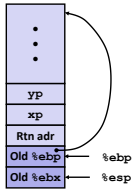


Resulting Stack

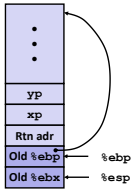

```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

swap Finish #1

swap's Stack



Resulting Stack

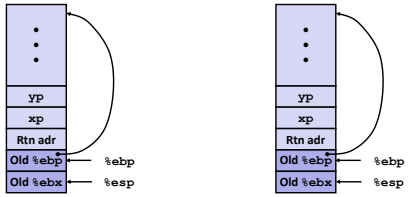



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation: Saved and restored register %ebx

swap Finish #2

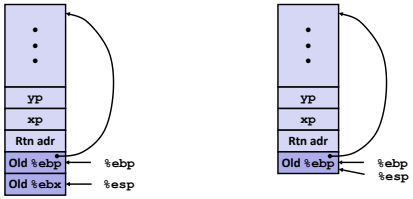
swap's Stack




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

swap Finish #2

swap's Stack

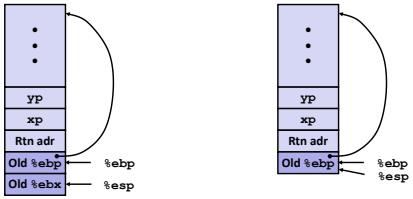


Resulting Stack


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

swap Finish #2

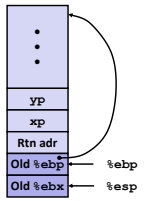
swap's Stack



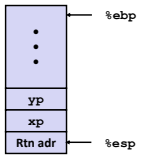

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #3

swap's Stack



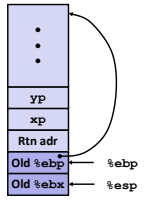
Resulting Stack

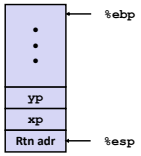



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4

swap's Stack

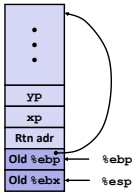




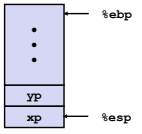

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4

swap's Stack



Resulting Stack




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

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

Disassembled `swap`

```
080483a4 <swap>:
 80483a4:   55           push   %ebp
 80483a5:   89 e5        mov    %esp,%ebp
 80483a7:   53           push   %ebx
 80483a8:   8b 55 08     mov    0x8(%ebp),%edx
 80483ab:   8b 4d 0c     mov    0xc(%ebp),%ecx
 80483ae:   8b 1a        mov    (%edx),%ebx
 80483b0:   8b 01        mov    (%ecx),%eax
 80483b2:   89 02        mov    %eax,(%edx)
 80483b4:   89 19        mov    %ebx,(%ecx)
 80483b6:   5b           pop    %ebx
 80483b7:   c9           leave
 80483b8:   c3           ret
```

Calling Code

```
 8048409:   e8 96 ff ff ff   call 80483a4 <swap>
 804840e:   8b 45 f8         mov  0xfffffff8(%ebp),%eax
```

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
```

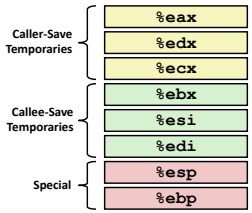
  - Contents of register `%edx` overwritten by `who`

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

- `%eax, %edx, %ecx`
  - Caller saves prior to call if values are used later
- `%eax`
  - also used to return integer value
- `%ebx, %esi, %edi`
  - Callee saves if wants to use them
- `%esp, %ebp`
  - special



Recursive Factorial

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

- **Registers**
  - `%eax` used without first saving
  - `%ebx` used, but saved at beginning & restore at end

```
.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
```

Pointer Code

Recursive Procedure

```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

Top-Level Call

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

- **Pass pointer to update location**

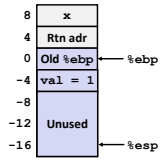
Creating & Initializing Pointer

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

- Variable `val` must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as `-4(%ebp)`
- Push on stack as second argument

Initial part of `sfact`

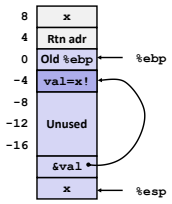
```
_sfact:
   pushl %ebp          # Save %ebp
   movl %esp,%ebp      # Set %ebp
   subl $16,%esp       # Add 16 bytes
   movl 8(%ebp),%edx   # edx = x
   movl $1,-4(%ebp)    # val = 1
```



Passing Pointer

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Stack at time of call

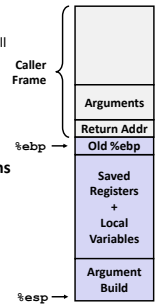


Calling `s_helper` from `sfact`

```
   leal -4(%ebp),%eax  # Compute &val
   pushl %eax          # Push on stack
   pushl %edx          # Push x
   call s_helper       # call
   movl -4(%ebp),%eax  # Return val
   • • •               # Finish
```

IA 32 Procedure Summary

- **The Stack Makes Recursion Work**
  - Private storage for each *instance* of procedure call
    - Instantiations don't clobber each other
    - Addressing of locals + arguments can be relative to stack positions
  - Managed by stack discipline
    - Procedures return in inverse order of calls
- **IA32 Procedures Combination of Instructions + Conventions**
  - Call / Ret instructions
  - Register usage conventions
    - Caller / Callee save
    - `%ebp` and `%esp`
  - Stack frame organization conventions



EXTRA

X86-64

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 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

Body

Finish

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

x86-64 Switch Implementation

- **Same general idea, adapted to 64-bit code**
- **Table entries 64 bits (pointers)**
- **Cases use revised code**

```
    switch(x) {
    case 1:       // .L50
        w = y*z;
        break;
      . . .
    }
```

```
.L50: // Case 1:
  movq   %rsi, %r8  # w = y
  imulq  %rdx, %r8  # w *= z
  movq   %r8, %rax  # Return w
  ret
```

Jump Table

```
.section .rodata
   .align 8
.L62:
  .quad   .L55  # x = 0
  .quad   .L50  # x = 1
  .quad   .L51  # x = 2
  .quad   .L52  # x = 3
  .quad   .L55  # x = 4
  .quad   .L54  # x = 5
  .quad   .L54  # x = 6
```

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!

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

x86-64 Object Code

- **Setup**
  - Label **.L61** becomes address **0x0000000000400716**
  - Label **.L62** becomes address **0x0000000000400990**

Assembly Code

```
switch_eg:
  . . .
  ja    .L55            # if > goto default
  jmp   *.L56(,%rdi,8)  # goto JTab[x]
```

Disassembled Object Code

```
0000000000400700 <switch_eg>:
 . . .
  40070d:  77 07                   ja     400716
  40070f:  ff 24 fd 90 09 40 00    jmpq   *0x400990(,%rdi,8)
```

x86-64 Object Code (cont.)

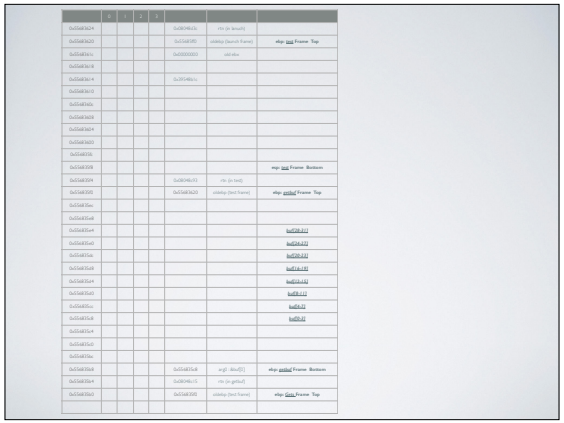
- **Jump Table**
  - Can inspect using GDB

    gdb asm-cntl

    (gdb) x/7xg 0x400990
    - Examine 7 hexadecimal format "giant words" (8-bytes each)
    - Use command "**help x**" to get format documentation

    0x400990:

    0x0000000000400716
    0x0000000000400739
    0x0000000000400720
    0x000000000040072b
    0x0000000000400716
    0x0000000000400732
    0x0000000000400732



	0	1	2	3			
0x55683624					0x08048d3c	rtn (in lanuch)	
0x55683620					0x55685ff0	oldebp (launch frame)	**ebp: _test_ Frame Top**
0x5568361c					0x00000000	old ebx	
0x55683618							
0x55683614					0x39548b1c		
0x55683610							
0x5568360c							
0x55683608							
0x55683604							
0x55683600							
0x556835fc							
0x556835f8							**ebp: _test_ Frame Bottom**
0x556835f4					0x08048c93	rtn (in test)	
0x556835f0					0x55683620	oldebp (test frame)	**ebp: _getbuf_ Frame Top**
0x556835ec							
0x556835e8							
0x556835e4							_buf[28-31]_
0x556835e0							_buf[24-27]_
0x556835dc							_buf[20-23]_
0x556835d8							_buf[16-19]_
0x556835d4							_buf[12-15]_
0x556835d0							_buf[8-11]_
0x556835cc							_buf[4-7]_
0x556835c8							_buf[0-3]_
0x556835c4							
0x556835c0							
0x556835bc							
0x556835b8					0x556835c8	arg0 : &buf[0]	**ebp: _getbuf_ Frame Bottom**
0x556835b4					0x08048c15	rtn (in getbuf)	
0x556835b0					0x556835f0	oldebp (test frame)	**ebp: _Gets_ Frame Top**