Computer System Organization (CS2.201)

Lecture # 08-12

Instruction Set Architecture / Assembly Language Programming

Avinash Sharma

Center for Visual Information Technology (CVIT),

IIIT Hyderabad

Slide content acknowledgment: Dr. Suresh Purini & other public sources

Arithmetic and Logical Operations

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow \&S$	Load effective address

Compiler can use it for compact arithmetic operations:

```
leal 7(%edx,%edx,4), %eax
```

Arithmetic and Logical Operations

Unary Operations

Instruction		Effect	Description
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement

The single operand D can be either a register or a memory location.

- E.g., **incl (%esp)** causes the 4-byte element on the top of the stack to be incremented.

Arithmetic and Logical Operations

Binary Operations

Instruction		Effect	Description
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \& S$	And

subl %eax,%edx decrements register %edx by the value in %eax.

Arithmetic and Logical Operations

Shift Operations

Instruction		Effect	Description
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Operation	Values	
Argument `x`	[01100011]	[10010101]
`x << 4`	[0011*0000*]	[0101*0000*]
`x >> 4` (logical)	[*0000*0110]	[*0000*1001]
`x >> 4` (arithmetic)	[*0000*0110]	[*1111*1001]

Arithmetic and Logical Operations

(a) C code

```
1   int arith(int x,
2             int y,
3             int z)
4   {
5       int t1 = x+y;
6       int t2 = z*48;
7       int t3 = t1 & 0xFFFF;
8       int t4 = t2 * t3;
9       return t4;
10  }
```

(b) Assembly code

```
    x at %ebp+8, y at %ebp+12, z at %ebp+16
1       movl    16(%ebp), %eax          z
2       leal    (%eax,%eax,2), %eax     z*3
3       sall    $4, %eax                t2 = z*48
4       movl    12(%ebp), %edx          y
5       addl    8(%ebp), %edx           t1 = x+y
6       andl    $65535, %edx            t3 = t1&0xFFFF
7       imull   %edx, %eax              Return t4 = t2*t3
```

Special Arithmetic Operations

Instruction		Effect	Description
`imull`	S	R[%edx]:R[%eax] $\leftarrow S \times$ R[%eax]	Signed full multiply
`mull`	S	R[%edx]:R[%eax] $\leftarrow S \times$ R[%eax]	Unsigned full multiply

```
                x at %ebp+8, y at %ebp+12
1               movl    12(%ebp), %eax    Put y in %eax
2               imull   8(%ebp)           Multiply by x
3               movl    %eax, (%esp)      Store low-order 32 bits
4               movl    %edx, 4(%esp)     Store high-order 32 bits
```

Special Arithmetic Operations

Instruction		Effect	Description
`cltd`		R[%edx]:R[%eax] ← SignExtend(R[%eax])	Convert to quad word
`idivl`	*S*	R[%edx] ← R[%edx]:R[%eax] mod *S*; R[%eax] ← R[%edx]:R[%eax] ÷ *S*	Signed divide
`divl`	*S*	R[%edx] ← R[%edx]:R[%eax] mod *S*; R[%eax] ← R[%edx]:R[%eax] ÷ *S*	Unsigned divide

Special Arithmetic Operations

```
      x at %ebp+8, y at %ebp+12
1       movl    8(%ebp), %edx     Put x in %edx
2       movl    %edx, %eax        Copy x to %eax
3       sarl    $31, %edx         Sign extend x in %edx
4       idivl   12(%ebp)          Divide by y
5       movl    %eax, 4(%esp)     Store x / y
6       movl    %edx, (%esp)      Store x % y
```

Condition Codes

- CPU maintains a set of single-bit condition code registers describing attributes of the most recent arithmetic or logical operation.
- These registers (listed below) can then be tested to perform conditional branches.

`CF`: Carry Flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

`ZF`: Zero Flag. The most recent operation yielded zero.

`SF`: Sign Flag. The most recent operation yielded a negative value.

`OF`: Overflow Flag. The most recent operation caused a two's-complement overflow—either negative or positive.

Condition Codes

`t=a+b,`

`CF:`	`(unsigned) t < (unsigned) a`	Unsigned overflow
`ZF:`	`(t == 0)`	Zero
`SF:`	`(t < 0)`	Negative
`OF:`	`(a < 0 == b < 0) && (t < 0 != a < 0)`	Signed overflow

Condition Codes

- All unary and binary arithmetic & logical operations (except leaq) set the single bit condition codes.
- For logical operations, the carry and overflow flags are set to zero.
- Fore shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to zero.
- INC and DEC instruction set the overflow and zero flags but leave the carry flag unchanged.

Condition Codes

- Additionally, two instruction classes (CMP and TEST) set the condition codes.
- CMP behave similar to SUB without altering the destination register.
- TEST behave similar to AND without altering the destination register.

Instruction		Based on	Description
CMP	S_2, S_1	$S_1 - S_2$	Compare
`cmpb`		Compare byte	
`cmpw`		Compare word	
`cmpl`		Compare double word	
TEST	S_2, S_1	S_1 & S_2	Test
`testb`		Test byte	
`testw`		Test word	
`testl`		Test double word	

Condition Codes

- Three common ways to use the condition codes:
  1. Set a single byte to 0 or 1 depending on some combination of condition codes (SET instructions)
  2. Conditionally jump some other part of program (Jump instructions)
  3. Conditionally move data (CMOVE instructions)

Accessing the Condition Codes

- Instruction suffixes refer to combination of condition codes and not the operand types.
- Operand *D* refers to a single byte register or single byte memory location.

Instruction		Synonym	Effect	Set condition
`sete`	*D*	`setz`	*D* ← `ZF`	Equal / zero
`setne`	*D*	`setnz`	*D* ← `~ZF`	Not equal / not zero
`sets`	*D*		*D* ← `SF`	Negative
`setns`	*D*		*D* ← `~SF`	Nonnegative
`setg`	*D*	`setnle`	*D* ← `~(SF ^ OF) & ~ZF`	Greater (signed >)
`setge`	*D*	`setnl`	*D* ← `~(SF ^ OF)`	Greater or equal (signed >=)
`setl`	*D*	`setnge`	*D* ← `SF ^ OF`	Less (signed <)
`setle`	*D*	`setng`	*D* ← `(SF ^ OF) | ZF`	Less or equal (signed <=)
`seta`	*D*	`setnbe`	*D* ← `~CF & ~ZF`	Above (unsigned >)
`setae`	*D*	`setnb`	*D* ← `~CF`	Above or equal (unsigned >=)
`setb`	*D*	`setnae`	*D* ← `CF`	Below (unsigned <)
`setbe`	*D*	`setna`	*D* ← `CF | ZF`	Below or equal (unsigned <=)

Condition Codes

a < *b*

```
    a is in %edx, b is in %eax
1     cmpl    %eax, %edx     Compare a:b
2     setl    %al            Set low order byte of %eax to 0 or 1
3     movzbl  %al, %eax      Set remaining bytes of %eax to 0
```

`setl`	*D*	`setnge`	*D* ← `SF ^ OF`	Less (signed <)

Jump Instruction

Jump instruction can be either direct (to a label) or indirect (to a value stored in register , e.g., jmp *%eax)

```
1       movl $0,%eax              Set %eax to 0
2       jmp .L1                   Goto .L1
3       movl (%eax),%edx          Null pointer dereference
4     .L1:
5       popl %edx
```

Condition Codes & Jump Instructions

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	`*`*Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

Jump Instruction Encoding

- The two common encodings used by assemblers are:
  - First encoding known as *PC relative* encode the difference between the address of the target instruction and the address of the instruction immediately following the jump.
  - These offsets can be encoded using 1, 2, or 4 bytes.
  - Second encoding method is to give an “absolute” address, using 4 bytes to directly specify the target.

Jump Instruction

```
1     jle     .L2              if <=, goto dest2
2   .L5:                     dest1:
3     movl    %edx, %eax
4     sarl    %eax
5     subl    %eax, %edx
6     leal    (%edx,%edx,2), %edx
7     testl   %edx, %edx
8     jg      .L5              if >, goto dest1
9   .L2:                     dest2:
10    movl    %edx, %eax
```

Disassembled version of the ".o" format

```
1      8:   7e 0d              jle    17 <silly+0x17>   Target = dest2
2      a:   89 d0              mov    %edx,%eax         dest1:
3      c:   d1 f8              sar    %eax
4      e:   29 c2              sub    %eax,%edx
5     10:   8d 14 52           lea    (%edx,%edx,2),%edx
6     13:   85 d2              test   %edx,%edx
7     15:   7f f3              jg     a <silly+0xa>     Target = dest1
8     17:   89 d0              mov    %edx,%eax         dest2:
```

Jump Instruction

```
1     jle     .L2              if <=, goto dest2
2   .L5:                     dest1:
3     movl    %edx, %eax
4     sarl    %eax
5     subl    %eax, %edx
6     leal    (%edx,%edx,2), %edx
7     testl   %edx, %edx
8     jg      .L5              if >, goto dest1
9   .L2:                     dest2:
10    movl    %edx, %eax
```

Disassembled version of the program after linking

```
1     804839c:  7e 0d                 jle    80483ab <silly+0x17>
2     804839e:  89 d0                 mov    %edx,%eax
3     80483a0:  d1 f8                 sar    %eax
4     80483a2:  29 c2                 sub    %eax,%edx
5     80483a4:  8d 14 52              lea    (%edx,%edx,2),%edx
6     80483a7:  85 d2                 test   %edx,%edx
7     80483a9:  7f f3                 jg     804839e <silly+0xa>
8     80483ab:  89 d0                 mov    %edx,%eax
```

Translating Conditional Branches

(a) Original C code

```
1   int absdiff(int x, int y) {
2       if (x < y)
3           return y - x;
4       else
5           return x - y;
6   }
```

(b) Equivalent goto version

```
1   int gotodiff(int x, int y) {
2       int result;
3       if (x >= y)
4           goto x_ge_y;
5       result =  y - x;
6       goto done;
7    x_ge_y:
8       result = x - y;
9    done:
10      return result;
11  }
```

Translating Conditional Branches

(c) Generated assembly code

```
    x at %ebp+8, y at %ebp+12
 1      movl    8(%ebp), %edx     Get x
 2      movl    12(%ebp), %eax    Get y
 3      cmpl    %eax, %edx        Compare x:y
 4      jge     .L2               if >= goto x_ge_y
 5      subl    %edx, %eax        Compute result = y-x
 6      jmp     .L3               Goto done
 7    .L2:                      x_ge_y:
 8      subl    %eax, %edx        Compute result = x-y
 9      movl    %edx, %eax        Set result as return value
10    .L3:                      done: Begin completion code
```

Translating Conditional Branches

```
if (test-expr)
    then-statement
else
    else-statement
```

```
    t = test-expr;
    if (!t)
        goto false;
    then-statement
    goto done;
false:
    else-statement
done:
```

Conditional Branches with Conditional Move

(a) Original C code

```
1   int absdiff(int x, int y) {
2       return x < y ? y-x : x-y;
3   }
```

(b) Implementation using conditional assignment

```
1   int cmovdiff(int x, int y) {
2       int tval = y-x;
3       int rval = x-y;
4       int test = x < y;
5       /* Line below requires
6          single instruction: */
7       if (test) rval = tval;
8       return rval;
9   }
```

Conditional Branches with Conditional Move

(a) Original C code

```
1   int absdiff(int x, int y) {
2       return x < y ? y-x : x-y;
3   }
```

(b) Implementation using conditional assignment

```
1   int cmovdiff(int x, int y) {
2       int tval = y-x;
3       int rval = x-y;
4       int test = x < y;
5       /* Line below requires
6          single instruction: */
7       if (test) rval = tval;
8       return rval;
9   }
```

Conditional Branches with Conditional Move

(c) Generated assembly code

```
    x at %ebp+8, y at %ebp+12
1       movl    8(%ebp), %ecx     Get x
2       movl    12(%ebp), %edx    Get y
3       movl    %edx, %ebx        Copy y
4       subl    %ecx, %ebx        Compute y-x
5       movl    %ecx, %eax        Copy x
6       subl    %edx, %eax        Compute x-y and set as return value
7       cmpl    %edx, %ecx        Compare x:y
8       cmovl   %ebx, %eax        If <, replace return value with y-x
```

Conditional Branches with Conditional Move

Instruction		Synonym	Move condition	Description
`cmove`	*S*, *R*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *R*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *R*		`SF`	Negative
`cmovns`	*S*, *R*		`~SF`	Nonnegative
`cmovg`	*S*, *R*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *R*	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	*S*, *R*	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	*S*, *R*	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	*S*, *R*	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	*S*, *R*	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	*S*, *R*	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	*S*, *R*	`cmovna`	`CF | ZF`	below or equal (unsigned <=)

When to use Conditional Move

- Improved Performance for Pipelined Execution ?

- Causes Error Condition ?

```
long cread(long *xp) {
  return (xp ? *xp : 0);
}
```

```
1.  cread:
2.        movl (%edx), (%eax)
3.        testl  %edx, %edx
4.        movl $0, %edx
5.        cmove %edx, %eax
```

Conditional Branches: Do-While Loop

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

```
    do
        body-statement
    while (test-expr);
```

```
    loop:
        body-statement
        t = test-expr;
        if (t)
            goto loop;
```

```
    Argument: n at %ebp+8
    Registers: n in %edx, result in %eax
1       movl    8(%ebp), %edx     Get n
2       movl    $1, %eax          Set result = 1
3     .L2:                      loop:
4       imull   %edx, %eax        Compute result *= n
5       subl    $1, %edx          Decrement n
6       cmpl    $1, %edx          Compare n:1
7       jg      .L2               If >, goto loop
    Return result
```

Conditional Branches: While Loop

```
while (test-expr)
    body-statement
```

```
if (!test-expr)
    goto done;
do
    body-statement
    while (test-expr);
done:
```

```
    t = test-expr;
    if (!t)
        goto done;
loop:
    body-statement
    t = test-expr;
    if (t)
        goto loop;
done:
```

Conditional Branches: While Loop

```
1   int fact_while(int n)
2   {
3       int result = 1;
4       while (n > 1) {
5           result *= n;
6           n = n-1;
7       }
8       return result;
9   }
```

```
1   int fact_while_goto(int n)
2   {
3       int result = 1;
4       if (n <= 1)
5           goto done;
6    loop:
7       result *= n;
8       n = n-1;
9       if (n > 1)
10          goto loop;
11   done:
12      return result;
13  }
```

```
    Argument: n at %ebp+8
    Registers: n in %edx, result in %eax
1       movl    8(%ebp), %edx     Get n
2       movl    $1, %eax          Set result = 1
3       cmpl    $1, %edx          Compare n:1
4       jle     .L7               If <=, goto done
5    .L10:                      loop:
6       imull   %edx, %eax        Compute result *= n
7       subl    $1, %edx          Decrement n
8       cmpl    $1, %edx          Compare n:1
9       jg      .L10              If >, goto loop
10   .L7:                       done:
      Return result
```

Conditional Branches: For Loop

```
for (init-expr; test-expr; update-expr)
    body-statement
```

```
    init-expr;
    while (test-expr) {
        body-statement
        update-expr;
    }
```

```
    init-expr;
    if (!test-expr)
        goto done;
    do {
        body-statement
        update-expr;
    } while (test-expr);
done:
```

```
        init-expr;
        t = test-expr;
        if (!t)
            goto done;
loop:
        body-statement
        update-expr;
        t = test-expr;
        if (t)
            goto loop;
done:
```

Conditional Branches: For Loop

```
1   int fact_for(int n)
2   {
3       int i;
4       int result = 1;
5       for (i = 2; i <= n; i++)
6           result *= i;
7       return result;
8   }
```

```
    Argument: n at %ebp+8
    Registers: n in %ecx, i in %edx, result in %eax
1       movl    8(%ebp), %ecx     Get n
2       movl    $2, %edx          Set i to 2          (init)
3       movl    $1, %eax          Set result to 1
4       cmpl    $1, %ecx          Compare n:1         (!test)
5       jle     .L14              If <=, goto done
6   .L17:                       loop:
7       imull   %edx, %eax        Compute result *= i (body)
8       addl    $1, %edx          Increment i         (update)
9       cmpl    %edx, %ecx        Compare n:i         (test)
10      jge     .L17              If >=, goto loop
11  .L14:                       done:
```

Conditional Branches: Switch Statement

```
1   int switch_eg(int x, int n) {
2       int result = x;
3
4       switch (n) {
5
6       case 100:
7           result *= 13;
8           break;
9
10      case 102:
11          result += 10;
12          /* Fall through */
13
14      case 103:
15          result += 11;
16          break;
17
18      case 104:
19      case 106:
20          result *= result;
21          break;
22
23      default:
24          result = 0;
25      }
26
27      return result;
28  }
```

```
1   int switch_eg_impl(int x, int n) {
2       /* Table of code pointers */
3       static void *jt[7] = {
4           &&loc_A, &&loc_def, &&loc_B,
5           &&loc_C, &&loc_D, &&loc_def,
6           &&loc_D
7       };
8
9       unsigned index = n - 100;
10      int result;
11
12      if (index > 6)
13          goto loc_def;
14
15      /* Multiway branch */
16      goto *jt[index];
17
18   loc_def:  /* Default case*/
19      result = 0;
20      goto done;
21
```

```
22    loc_C:    /* Case 103 */
23       result = x;
24       goto rest;
25
26    loc_A:    /* Case 100 */
27       result = x * 13;
28       goto done;
29
30    loc_B:    /* Case 102 */
31       result = x + 10;
32       /* Fall through */
33
34    rest:     /* Finish case 103 */
35       result += 11;
36       goto done;
37
38    loc_D:    /* Cases 104, 106 */
39       result = x * x;
40       /* Fall through */
41
42    done:
43       return result;
44    }
```

Conditional Branches: Switch Statement

```
    x at %ebp+8, n at %ebp+12
1     movl    8(%ebp), %edx          Get x
2     movl    12(%ebp), %eax         Get n
    Set up jump table access
3     subl    $100, %eax             Compute index = n-100
4     cmpl    $6, %eax               Compare index:6
5     ja      .L2                    If >, goto loc_def
6     jmp     *.L7(,%eax,4)          Goto *jt[index]
    Default case
7   .L2:                           loc_def:
8     movl    $0, %eax               result = 0;
9     jmp     .L8                    Goto done
    Case 103
10  .L5:                           loc_C:
11    movl    %edx, %eax             result = x;
12    jmp     .L9                    Goto rest
    Case 100
13  .L3:                           loc_A:
14    leal    (%edx,%edx,2), %eax      result = x*3;
15    leal    (%edx,%eax,4), %eax      result = x+4*result
16    jmp     .L8                      Goto done
    Case 102
17  .L4:                           loc_B:
18    leal    10(%edx), %eax           result = x+10
    Fall through
19  .L9:                           rest:
20    addl    $11, %eax                result += 11;
21    jmp     .L8                      Goto done
    Cases 104, 106
22  .L6:                           loc_D
23    movl    %edx, %eax               result = x
24    imull   %edx, %eax               result *= x
    Fall through
25  .L8:                           done:
    Return result
```

Conditional Branches: Switch Statement

```
1       .section        .rodata
2       .align 4        Align address to multiple of 4
3   .L7:
4       .long   .L3     Case 100: loc_A
5       .long   .L2     Case 101: loc_def
6       .long   .L4     Case 102: loc_B
7       .long   .L5     Case 103: loc_C
```

Procedures

- A procedure call involves control from one part of a program to another.
- It also involves passing both data (in the form of procedure parameters and return values).
- In addition, it must allocate space for the local variables of the procedure on entry and deallocate them on exit.
- The passing of data and the allocation and deallocation of local variables is handled by manipulating the program stack.

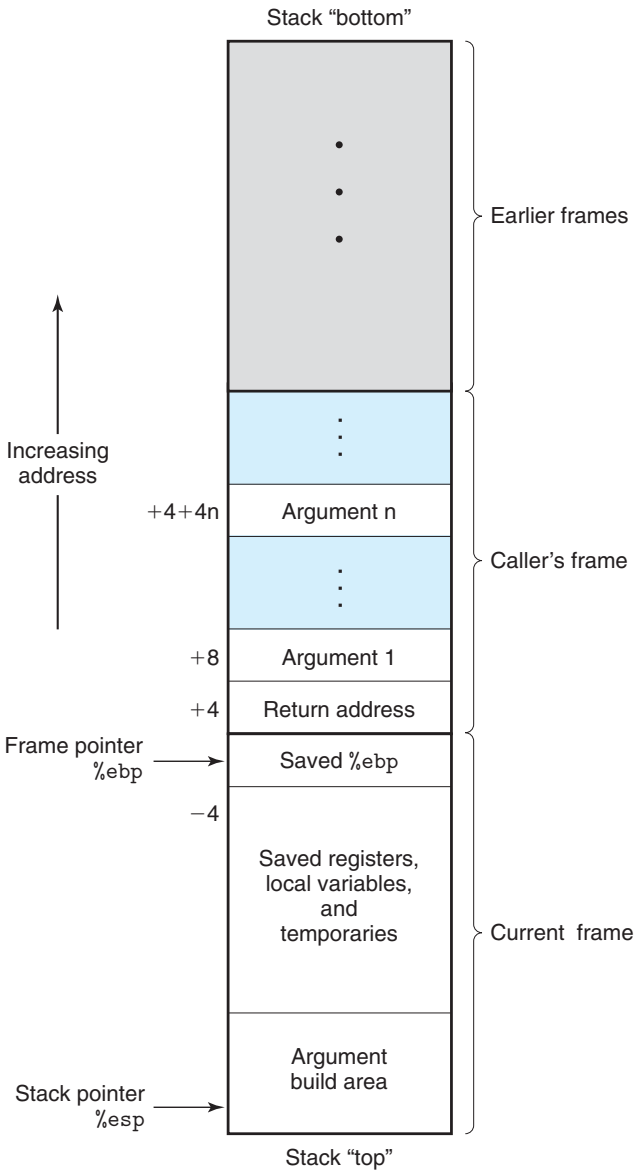
Procedures: Stack Frame

- The *stack frame*, also known as *activation record* is the collection of all data on the stack associated with one subprogram call.

- The stack frame generally includes the following components:
  - The return address
  - Argument variables passed on the stack
  - Local variables
  - Saved copies of any registers modified by the subprogram that need to be restored .

Procedures: Stack Frame

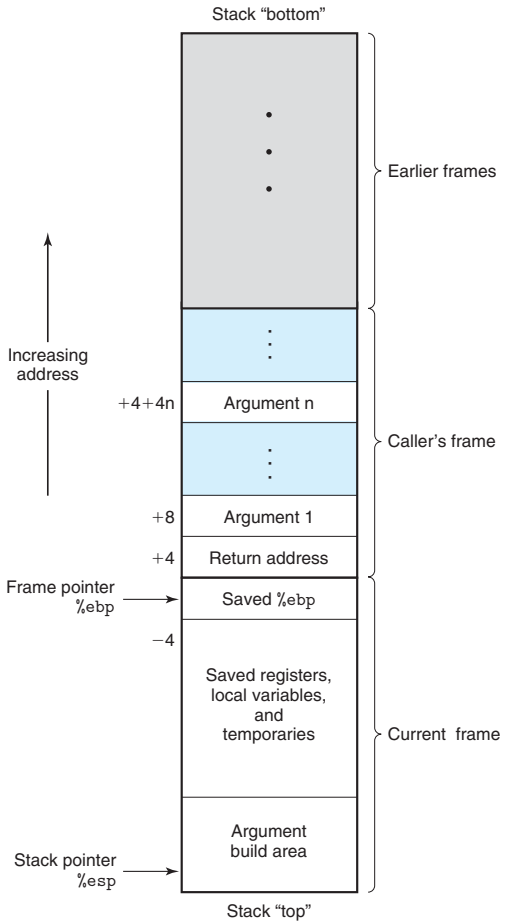
- Procedure Q uses the stack for any local variables that cannot be stored in registers. This can occur for the following reasons:
  - There are not enough registers to hold all of the local data.
  - Some of the local variables are arrays or structures and hence must be accessed by array or structure references.
  - The address operator '&' is applied to a local variable, and hence we must be able to generate an address for it.
- In addition, Q uses the stack frame for storing arguments to any procedures it calls.



Procedures

- Procedure P (the *caller*) calls procedure Q (the *callee*). The arguments to Q are contained within the stack frame for P.
- When P calls Q, the *return address* within P where the program should resume execution when it returns from Q is pushed onto the stack, forming the end of P's stack frame.
- The stack frame for Q starts with the saved value of the frame pointer (a copy of register %ebp), followed by copies of any other saved register values.

Procedures: Stack Frame x86_64



- First 6 arguments are passed by dedicated registers, named, %rdi, %rsi, %rdx, %rcx, %r8 and %r9 (for 64 bit operands). For 32, 16 and 8 bit operands, respective sub-registers are used.
- Extra arguments are added to stack frame of caller function

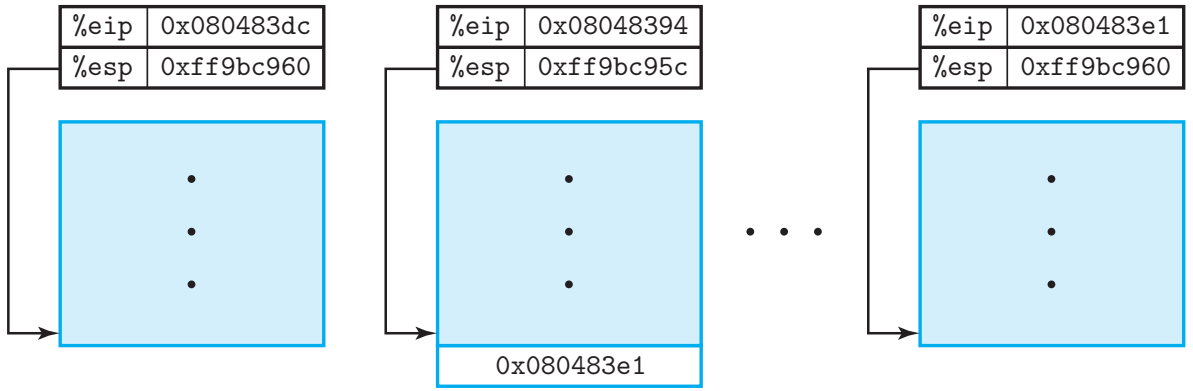
Procedures: Transferring Control

- The effect of a ***call*** instruction is to push a return address on the stack and jump to the start of the called procedure.
- The ***ret*** instruction pops an address off the stack and jumps to this location.

Instruction		Description
`call`	*Label*	Procedure call
`call`	`*`*Operand*	Procedure call
`leave`		Prepare stack for return
`ret`		Return from call

Procedures: Transferring Control

```
    Beginning of function sum
1   08048394 <sum>:
2    8048394:   55                      push   %ebp
    . . .
    Return from function sum
3    80483a4:   c3                      ret
    . . .
    Call to sum from main
4    80483dc:   e8 b3 ff ff ff          call   8048394 <sum>
5    80483e1:   83 c4 14                add    $0x14,%esp
```



Procedures: Register Usage Conventions

- Registers %eax, %edx, and %ecx are classified as *caller-save* registers. When procedure Q is called by P, it can overwrite these registers without destroying any data required by P.

- On the other hand, registers %ebx, %esi, and %edi are classified as *callee-save* registers. This means that Q must save the values of any of these registers on the stack before overwriting them, and restore them before returning, because P (or some higher-level procedure) may need these values for its future computations.

- In addition, registers %ebp and %esp must be maintained according to the conventions described here.

Procedures: Register Usage Conventions

```
1    int P(int x)
2    {
3        int y = x*x;
4        int z = Q(y);
5        return y + z;
6    }
```

Procedures: Register Usage Conventions

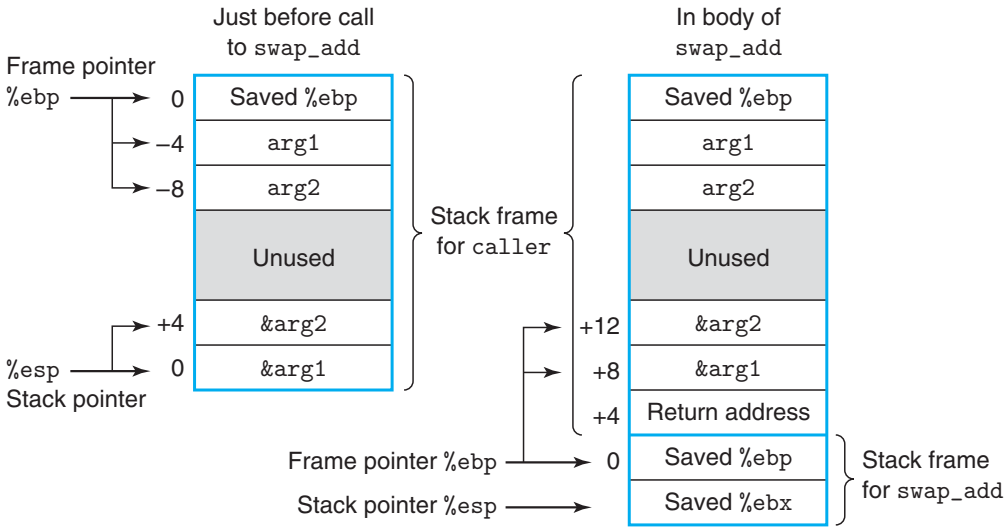
- Procedure P computes y before calling Q, but it must also ensure that the value of y is available after Q returns.
- It can :
  1. store the value of y in its own stack frame before calling Q; when Q returns, procedure P can then retrieve the value of y from the stack. In other words, P, the *caller,* saves the value.

  2. store the value of y in a callee-save register. If Q, or any procedure called by Q, wants to use this register, it must save the register value in its stack frame and restore the value before it returns (in other words, the *callee* saves the value). When Q returns to P, the value of y will be in the callee-save register, either because the register was never altered or because it was saved and restored.

Procedures: Register Usage Conventions

```
1   int swap_add(int *xp, int *yp)
2   {
3       int x = *xp;
4       int y = *yp;
5
6       *xp = y;
7       *yp = x;
8       return x + y;
9   }
10
11  int caller()
12  {
13      int arg1 = 534;
14      int arg2 = 1057;
15      int sum = swap_add(&arg1, &arg2);
16      int diff = arg1 - arg2;
17
18      return sum * diff;
19  }
```

Procedures: Register Usage Conventions



Procedures: Register Usage Conventions

```
1    caller:
2      pushl   %ebp              Save old %ebp
3      movl    %esp, %ebp        Set %ebp as frame pointer
4      subl    $24, %esp         Allocate 24 bytes on stack
5      movl    $534, -4(%ebp)    Set arg1 to 534
6      movl    $1057, -8(%ebp)   Set arg2 to 1057
7      leal    -8(%ebp), %eax    Compute &arg2
8      movl    %eax, 4(%esp)     Store on stack
9      leal    -4(%ebp), %eax    Compute &arg1
10     movl    %eax, (%esp)      Store on stack
11     call    swap_add          Call the swap_add function
```

```
int caller()
{
    int arg1 = 534;
    int arg2 = 1057;
    int sum = swap_add(&arg1, &arg2);
    int diff = arg1 - arg2;

    return sum * diff;
}
```

Procedures: Register Usage Conventions

```
1   swap_add:
2     pushl   %ebp              Save old %ebp
3     movl    %esp, %ebp        Set %ebp as frame pointer
4     pushl   %ebx              Save %ebx
5     movl    8(%ebp), %edx       Get xp
6     movl    12(%ebp), %ecx      Get yp
7     movl    (%edx), %ebx        Get x
8     movl    (%ecx), %eax        Get y
9     movl    %eax, (%edx)        Store y at xp
10    movl    %ebx, (%ecx)        Store x at yp
11    addl    %ebx, %eax          Return value = x+y
12    popl    %ebx              Restore %ebx
13    popl    %ebp              Restore %ebp
14    ret                       Return
```

```
int swap_add(int *xp, int *yp)
{
    int x = *xp;
    int y = *yp;

    *xp = y;
    *yp = x;
    return x + y;
}
```

Procedures: Register Usage Conventions

```
1   caller:
2     pushl   %ebp              Save old %ebp
3     movl    %esp, %ebp        Set %ebp as frame pointer
4     subl    $24, %esp         Allocate 24 bytes on stack
5     movl    $534, -4(%ebp)    Set arg1 to 534
6     movl    $1057, -8(%ebp)   Set arg2 to 1057
7     leal    -8(%ebp), %eax    Compute &arg2
8     movl    %eax, 4(%esp)     Store on stack
9     leal    -4(%ebp), %eax    Compute &arg1
10    movl    %eax, (%esp)      Store on stack
11    call    swap_add          Call the swap_add function
12    movl    -4(%ebp), %edx
13    subl    -8(%ebp), %edx
14    imull   %edx, %eax
15    leave
16    ret
```

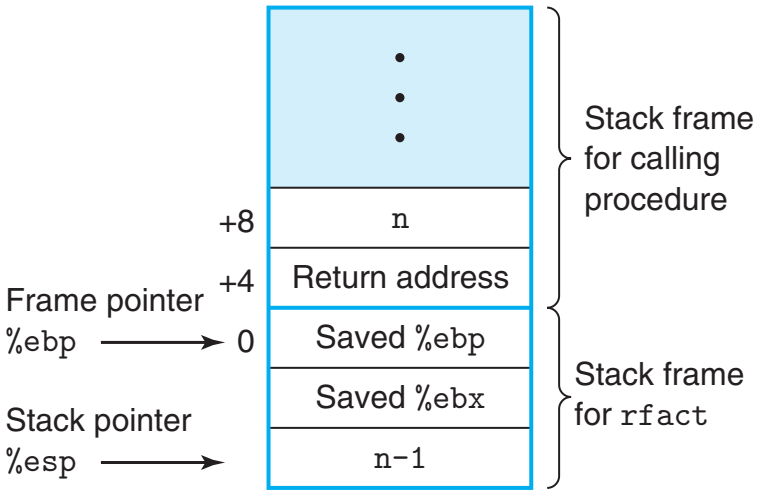
```
int caller()
{
    int arg1 = 534;
    int arg2 = 1057;
    int sum = swap_add(&arg1, &arg2);
    int diff = arg1 - arg2;

    return sum * diff;
}
```

Procedures: Recursion

```
1   int rfact(int n)
2   {
3       int result;
4       if (n <= 1)
5           result = 1;
6       else
7           result = n * rfact(n-1);
8       return result;
9   }
```

Procedures: Recursion



Procedures: Recursion

```
    Argument: n at %ebp+8
    Registers: n in %ebx, result in %eax
 1   rfact:
 2     pushl   %ebp              Save old %ebp
 3     movl    %esp, %ebp        Set %ebp as frame pointer
 4     pushl   %ebx              Save callee save register %ebx
 5     subl    $4, %esp          Allocate 4 bytes on stack
 6     movl    8(%ebp), %ebx     Get n
 7     movl    $1, %eax          result = 1
 8     cmpl    $1, %ebx          Compare n:1
 9     jle     .L53              If <=, goto done
10     leal    -1(%ebx), %eax    Compute n-1
11     movl    %eax, (%esp)      Store at top of stack
12     call    rfact             Call rfact(n-1)
13     imull   %ebx, %eax        Compute result = return value * n
14   .L53:                     done:
15     addl    $4, %esp          Deallocate 4 bytes from stack
16     popl    %ebx              Restore %ebx
17     popl    %ebp              Restore %ebp
18     ret                       Return result
```