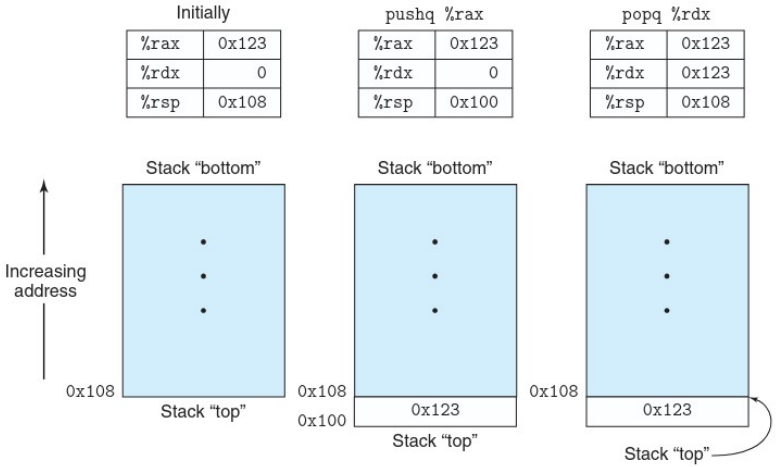
Run-Time Stack (ch 3.4.4)

- pushX
- popX

Instruction	Effect	Description
pushq S	R[%rsp] ← R[%rsp] − 8; M[R[%rsp]] ← S	Push quad word
popq D	D ← M[R[%rsp]]; R[%rsp] ← R[%rsp] + 8	Pop quad word

Figure 3.8 **Push and pop instructions.**

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8



Registers (X86_64)

- You get 16 registers

Type	Form	Operand value	Name
Immediate	$*Imm*	*Imm*	Immediate
Register	r_a	R[r_a]	Register
Memory	*Imm*	M[*Imm*]	Absolute
Memory	(r_a)	M[R[r_a]]	Indirect
Memory	*Imm*(r_b)	M[*Imm* + R[r_b]]	Base + displacement
Memory	(r_b,r_i)	M[R[r_b] + R[r_i]]	Indexed
Memory	*Imm*(r_b,r_i)	M[*Imm* + R[r_b] + R[r_i]]	Indexed
Memory	(,r_i,s)	M[R[r_i] · s]	Scaled indexed
Memory	*Imm*(,r_i,s)	M[*Imm* + R[r_i] · s]	Scaled indexed
Memory	(r_b,r_i,s)	M[R[r_b] + R[r_i] · s]	Scaled indexed
Memory	*Imm*(r_b,r_i,s)	M[*Imm* + R[r_b] + R[r_i] · s]	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

ASM Maths

- inc/dec ++/--
- add/sub/imul
- xor/or/and
- AR shift
- Logical shift
- Variants for all sizes

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow \&S$	Load effective address
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
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
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_{\mathrm{A}} k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_{\mathrm{L}} k$	Logical right shift

ASM Maths: imult/div

- Requires more regs to avoid overflow
- Int128 types!

Instruction		Effect	Description
`imulq`	*S*	R[%rdx]:R[%rax] ← *S* × R[%rax]	Signed full multiply
`mulq`	*S*	R[%rdx]:R[%rax] ← *S* × R[%rax]	Unsigned full multiply
`cqto`		R[%rdx]:R[%rax] ← SignExtend(R[%rax])	Convert to oct word
`idivq`	*S*	R[%rdx] ← R[%rdx]:R[%rax] mod *S*; R[%rax] ← R[%rdx]:R[%rax] ÷ *S*	Signed divide
`divq`	*S*	R[%rdx] ← R[%rdx]:R[%rax] mod *S*; R[%rax] ← R[%rdx]:R[%rax] ÷ *S*	Unsigned divide

Figure 3.12 Special arithmetic operations. These operations provide full 128-bit multiplication and division, for both signed and unsigned numbers. The pair of registers %rdx and %rax are viewed as forming a single 128-bit oct word.

ASM Control Flow

- Condition Codes
- Carry Flag
- Zero Flag
- Sign Flag
- Overflow

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

ASM Control Flow

- CMP and TEST
- Results?

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

ASM Control Flow

- SET

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 <=)

ASM Control Flow

- JMP
- C “goto”

```
  movq $0,%rax              Set %rax to 0
  jmp .L1                   Goto .L1
  movq (%rax),%rdx          Null pointer dereference (skipped)
.L1:
  popq %rdx                 Jump target
```

ASM Control Flow

- JMP operations

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 <=)

ASM Control Flow

- Putting it all together.

```
long lt_cnt = 0;
long ge_cnt = 0;

long absdiff_se(long x, long y)
{
    long result;
    if (x < y) {
        lt_cnt++;
        result = y - x;
    }
    else {
        ge_cnt++;
        result = x - y;
    }
    return result;
}
```

```
1  long gotodiff_se(long x, long y)
2  {
3      long result;
4      if (x >= y)
5          goto x_ge_y;
6      lt_cnt++;
7      result =  y - x;
8      return result;
9   x_ge_y:
10     ge_cnt++;
11     result = x - y;
12     return result;
13 }
```

ASM Control Flow

- C Conversion

(b) Equivalent goto version

```
1   long gotodiff_se(long x, long y)
2   {
3       long result;
4       if (x >= y)
5           goto x_ge_y;
6       lt_cnt++;
7       result =  y - x;
8       return result;
9    x_ge_y:
10      ge_cnt++;
11      result = x - y;
12      return result;
13  }
```

```
    long absdiff_se(long x, long y)
    x in %rdi, y in %rsi
1   absdiff_se:
2     cmpq    %rsi, %rdi          Compare x:y
3     jge     .L2                 If >= goto x_ge_y
4     addq    $1, lt_cnt(%rip)    lt_cnt++
5     movq    %rsi, %rax
6     subq    %rdi, %rax          result = y - x
7     ret                         Return
8   .L2:                        x_ge_y:
9     addq    $1, ge_cnt(%rip)    ge_cnt++
10    movq    %rdi, %rax
11    subq    %rsi, %rax          result = x - y
12    ret                         Return
```

ASM Control Flow

- Branches and Moves

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

```
1   long cmovdiff(long x, long y)
2   {
3       long rval = y-x;
4       long eval = x-y;
5       long ntest = x >= y;
6       /* Line below requires
7          single instruction: */
8       if (ntest) rval = eval;
9       return rval;
10  }
```

ASM Control Flow

- C Conversion

```
1   long cmovdiff(long x, long y)
2   {
3       long rval = y-x;
4       long eval = x-y;
5       long ntest = x >= y;
6       /* Line below requires
7          single instruction: */
8       if (ntest) rval = eval;
9       return rval;
10  }
```

```
1   absdiff:
2     movq    %rsi, %rax
3     subq    %rdi, %rax        rval = y-x
4     movq    %rdi, %rdx
5     subq    %rsi, %rdx        eval = x-y
6     cmpq    %rsi, %rdi        Compare x:y
7     cmovge  %rdx, %rax        If >=, rval = eval
8     ret                       Return tval
```

long absdiff(long x, long y)
x in %rdi, y in %rsi

ASM Control Flow

- CMOV instructions

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 <=)

Figure 3.18 The conditional move instructions. These instructions copy the source value *S* to its destination *R* when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

ASM Control Flow

- Loops

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

```
long fact_do_goto(long n)
{
    long result = 1;
 loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

ASM Control Flow

- C Conversion

```
long fact_do_goto(long n)
{
    long result = 1;
 loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

```
long fact_do(long n)
n in %rdi
fact_do:
  movl    $1, %eax        Set result = 1
.L2:                    loop:
  imulq   %rdi, %rax      Compute result *= n
  subq    $1, %rdi        Decrement n
  cmpq    $1, %rdi        Compare n:1
  jg      .L2             If >, goto loop
  rep; ret                Return
```

ASM Control Flow

- Loops

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

```
long fact_while_jm_goto(long n)
{
    long result = 1;
    goto test;
 loop:
    result *= n;
    n = n-1;
 test:
    if (n > 1)
        goto loop;
    return result;
}
```

ASM Control Flow

- C Conversion

```
long fact_while_jm_goto(long n)
{
    long result = 1;
    goto test;
 loop:
    result *= n;
    n = n-1;
 test:
    if (n > 1)
        goto loop;
    return result;
}
```

```
  long fact_while(long n)
  n in %rdi
fact_while:
  movl    $1, %eax        Set result = 1
  jmp     .L5             Goto test
.L6:                    loop:
  imulq   %rdi, %rax      Compute result *= n
  subq    $1, %rdi        Decrement n
.L5:                    test:
  cmpq    $1, %rdi        Compare n:1
  jg      .L6             If >, goto loop
  rep; ret                Return
```

ASM Control Flow

- Case/Switch

```
void switch_eg(long x, long n,
               long *dest)
{
    long val = x;

    switch (n) {

    case 100:
        val *= 13;
        break;

    case 102:
        val += 10;
        /* Fall through */

    case 103:
        val += 11;
        break;

    case 104:
    case 106:
        val *= val;
        break;

    default:
        val = 0;
    }
    *dest = val;
}
```

```
1  void switch_eg_impl(long x, long n,
2                      long *dest)
3  {
4      /* Table of code pointers */
5      static void *jt[7] = {
6          &&loc_A, &&loc_def, &&loc_B,
7          &&loc_C, &&loc_D, &&loc_def,
8          &&loc_D
9      };
10     unsigned long index = n - 100;
11     long val;
12
13     if (index > 6)
14         goto loc_def;
15     /* Multiway branch */
16     goto *jt[index];
17
18  loc_A:     /* Case 100 */
19     val = x * 13;
20     goto done;
21  loc_B:     /* Case 102 */
22     x = x + 10;
23     /* Fall through */
24  loc_C:     /* Case 103 */
25     val = x + 11;
26     goto done;
27  loc_D:     /* Cases 104, 106 */
28     val = x * x;
29     goto done;
30  loc_def:   /* Default case */
31     val = 0;
32  done:
33     *dest = val;
34 }
```

ASM Control Flow

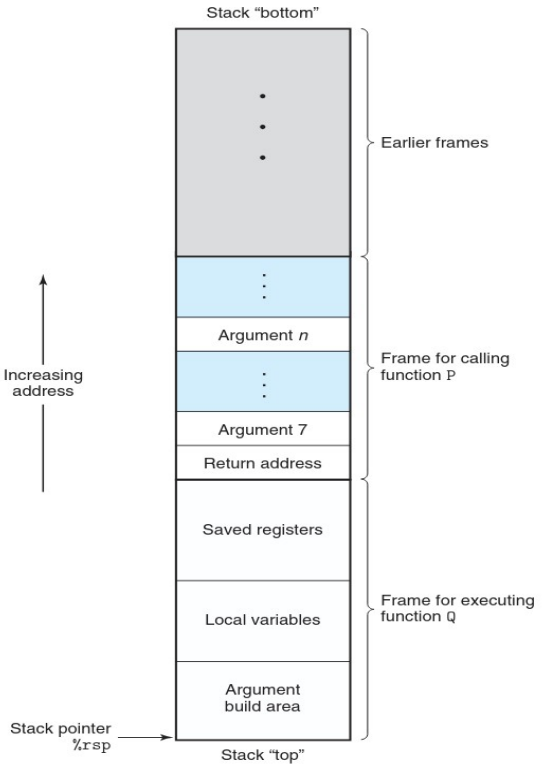
- C Conversion

```
   void switch_eg(long x, long n, long *dest)
   x in %rdi, n in %rsi, dest in %rdx
1  switch_eg:
2    subq    $100, %rsi              Compute index = n-100
3    cmpq    $6, %rsi                Compare index:6
4    ja      .L8                     If >, goto loc_def
5    jmp     *.L4(,%rsi,8)           Goto *jg[index]
6  .L3:                            loc_A:
7    leaq    (%rdi,%rdi,2), %rax     3*x
8    leaq    (%rdi,%rax,4), %rdi     val = 13*x
9    jmp     .L2                     Goto done
10 .L5:                            loc_B:
11   addq    $10, %rdi               x = x + 10
12 .L6:                            loc_C:
13   addq    $11, %rdi               val = x + 11
14   jmp     .L2                     Goto done
15 .L7:                            loc_D:
16   imulq   %rdi, %rdi              val = x * x
17   jmp     .L2                     Goto done
18 .L8:                            loc_def:
19   movl    $0, %edi                val = 0
20 .L2:                            done:
21   movq    %rdi, (%rdx)            *dest = val
22   ret                             Return
```

ASM Procedures

- Functions?
- More STACKS

Instruction		Description
`call`	*Label*	Procedure call
`call`	**Operand*	Procedure call
`ret`		Return from call



ASM Procedures

- Functions?

```
void proc(long  a1, long  *a1p,
          int   a2, int   *a2p,
          short a3, short *a3p,
          char  a4, char  *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

```
1  proc:
2    movq    16(%rsp), %rax    Fetch a4p   (64 bits)
3    addq    %rdi, (%rsi)      *a1p += a1  (64 bits)
4    addl    %edx, (%rcx)      *a2p += a2  (32 bits)
5    addw    %r8w, (%r9)       *a3p += a3  (16 bits)
6    movl    8(%rsp), %edx     Fetch a4    ( 8 bits)
7    addb    %dl, (%rax)       *a4p += a4  ( 8 bits)
8    ret                       Return
```

ASM Procedures

- Calling a function

```
long call_proc()
{
    long  x1 = 1; int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

```
    long call_proc()
1   call_proc:
      Set up arguments to proc
2     subq    $32, %rsp          Allocate 32-byte stack frame
3     movq    $1, 24(%rsp)       Store 1 in &x1
4     movl    $2, 20(%rsp)       Store 2 in &x2
5     movw    $3, 18(%rsp)       Store 3 in &x3
6     movb    $4, 17(%rsp)       Store 4 in &x4
7     leaq    17(%rsp), %rax     Create &x4
8     movq    %rax, 8(%rsp)      Store &x4 as argument 8
9     movl    $4, (%rsp)         Store 4 as argument 7
10    leaq    18(%rsp), %r9      Pass &x3 as argument 6
11    movl    $3, %r8d           Pass 3 as argument 5
12    leaq    20(%rsp), %rcx     Pass &x2 as argument 4
13    movl    $2, %edx           Pass 2 as argument 3
14    leaq    24(%rsp), %rsi     Pass &x1 as argument 2
15    movl    $1, %edi           Pass 1 as argument 1
      Call proc
16    call    proc
      Retrieve changes to memory
17    movslq  20(%rsp), %rdx     Get x2 and convert to long
18    addq    24(%rsp), %rdx     Compute x1+x2
19    movswl  18(%rsp), %eax     Get x3 and convert to int
20    movsbl  17(%rsp), %ecx     Get x4 and convert to int
21    subl    %ecx, %eax         Compute x3-x4
22    cltq                       Convert to long
23    imulq   %rdx, %rax         Compute (x1+x2) * (x3-x4)
24    addq    $32, %rsp          Deallocate stack frame
25    ret                        Return
```

ASM Procedures

- Recursive Calls

```
long rfact(long n)
{
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n-1);
    return result;
}
```

```
long rfact(long n)
n in %rdi
rfact:
    pushq   %rbx                Save %rbx
    movq    %rdi, %rbx          Store n in callee-saved register
    movl    $1, %eax            Set return value = 1
    cmpq    $1, %rdi            Compare n:1
    jle     .L35                If <=, goto done
    leaq    -1(%rdi), %rdi      Compute n-1
    call    rfact               Call rfact(n-1)
    imulq   %rbx, %rax          Multiply result by n
.L35:                         done:
    popq    %rbx                Restore %rbx
    ret                         Return
```

ASM Procedures

- Arrays(Size of LONG?)

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

```
long sum_element(long i, long j)
i in %rdi, j in %rsi
sum_element:
    leaq    0(,%rdi,8), %rdx
    subq    %rdi, %rdx
    addq    %rsi, %rdx
    leaq    (%rsi,%rsi,4), %rax
    addq    %rax, %rdi
    movq    Q(,%rdi,8), %rax
    addq    P(,%rdx,8), %rax
    ret
```

ASM Floats

- Even more regs

Instruction	Source	Destination	Description
vmovss	M_{32}	X	Move single precision
vmovss	X	M_{32}	Move single precision
vmovsd	M_{64}	X	Move double precision
vmovsd	X	M_{64}	Move double precision
vmovaps	X	X	Move aligned, packed single precision
vmovapd	X	X	Move aligned, packed double precision

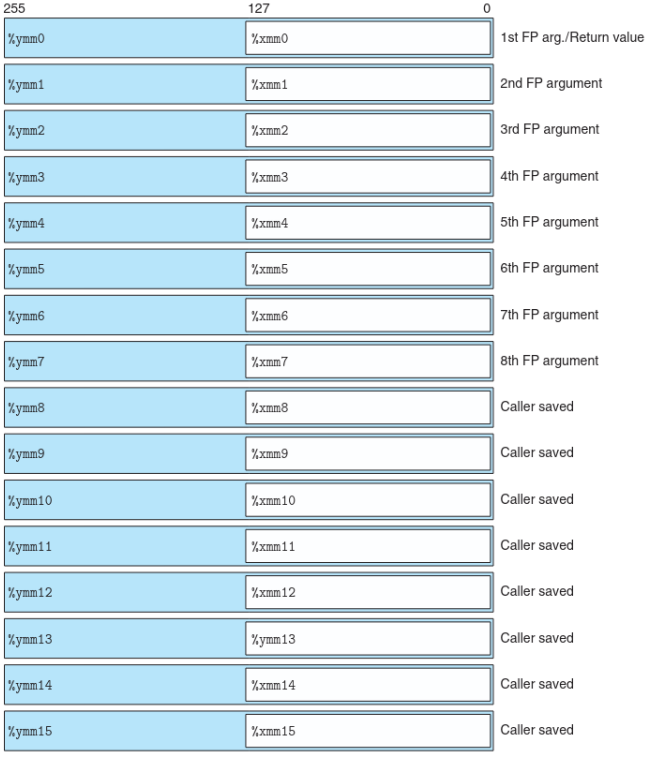


Figure 3.45 **Media registers.** These registers are used to hold floating-point data. Each YMM register holds 32 bytes. The low-order 16 bytes can be accessed as an XMM register.

ASM Floats

- Conversions

Instruction	Source	Destination	Description
`vcvttss2si`	X/M_{32}	R_{32}	Convert with truncation single precision to integer
`vcvttsd2si`	X/M_{64}	R_{32}	Convert with truncation double precision to integer
`vcvttss2siq`	X/M_{32}	R_{64}	Convert with truncation single precision to quad word integer
`vcvttsd2siq`	X/M_{64}	R_{64}	Convert with truncation double precision to quad word integer

Figure 3.47 Two-operand floating-point conversion operations. These convert floating-point data to integers. (X: XMM register (e.g., `%xmm3`); R_{32}: 32-bit general-purpose register (e.g., `%eax`); R_{64}: 64-bit general-purpose register (e.g., `%rax`); M_{32}: 32-bit memory range; M_{64}: 64-bit memory range)

ASM Floats

- Conversions

Instruction	Source 1	Source 2	Destination	Description
`vcvtsi2ss`	M_{32}/R_{32}	X	X	Convert integer to single precision
`vcvtsi2sd`	M_{32}/R_{32}	X	X	Convert integer to double precision
`vcvtsi2ssq`	M_{64}/R_{64}	X	X	Convert quad word integer to single precision
`vcvtsi2sdq`	M_{64}/R_{64}	X	X	Convert quad word integer to double precision

Figure 3.48 Three-operand floating-point conversion operations. These instructions convert from the data type of the first source to the data type of the destination. The second source value has no effect on the low-order bytes of the result. (X: XMM register (e.g., `%xmm3`); M_{32}: 32-bit memory range; M_{64}: 64-bit memory range)

ASM Floats

- C to ASM

```
double fcvt(int i, float *fp, double *dp, long *lp)
{
    float f = *fp; double d = *dp; long l = *lp;
    *lp = (long)    d;
    *fp = (float)   i;
    *dp = (double)  l;
    return (double) f;
}
```

```
    double fcvt(int i, float *fp, double *dp, long *lp)
    i in %edi, fp in %rsi, dp in %rdx, lp in %rcx
1   fcvt:
2     vmovss  (%rsi), %xmm0                    Get f = *fp
3     movq    (%rcx), %rax                     Get l = *lp
4     vcvttsd2siq     (%rdx), %r8              Get d = *dp and convert to long
5     movq    %r8, (%rcx)                      Store at lp
6     vcvtsi2ss       %edi, %xmm1, %xmm1       Convert i to float
7     vmovss  %xmm1, (%rsi)                    Store at fp
8     vcvtsi2sdq      %rax, %xmm1, %xmm1       Convert l to double
9     vmovsd  %xmm1, (%rdx)                    Store at dp
      The following two instructions convert f to double
10    vunpcklps       %xmm0, %xmm0, %xmm0
11    vcvtps2pd       %xmm0, %xmm0
12    ret                                      Return f
```

ASM Floats

- Even more math

Single	Double	Effect	Description
`vaddss`	`vaddsd`	$D \leftarrow S_2 + S_1$	Floating-point add
`vsubss`	`vsubsd`	$D \leftarrow S_2 - S_1$	Floating-point subtract
`vmulss`	`vmulsd`	$D \leftarrow S_2 \times S_1$	Floating-point multiply
`vdivss`	`vdivsd`	$D \leftarrow S_2/S_1$	Floating-point divide
`vmaxss`	`vmaxsd`	$D \leftarrow \max(S_2, S_1)$	Floating-point maximum
`vminss`	`vminsd`	$D \leftarrow \min(S_2, S_1)$	Floating-point minimum
`sqrtss`	`sqrtsd`	$D \leftarrow \sqrt{S_1}$	Floating-point square root

ASM Floats

- Comparison and Control

Instruction		Based on	Description
`ucomiss`	S_1, S_2	$S_2 - S_1$	Compare single precision
`ucomisd`	S_1, S_2	$S_2 - S_1$	Compare double precision

Ordering $S_2:S_1$	CF	ZF	PF
Unordered	1	1	1
$S_2 < S_1$	1	0	0
$S_2 = S_1$	0	1	0
$S_2 > S_1$	0	0	0

ASM Floats

- C to ASM Control

```c
typedef enum {NEG, ZERO, POS, OTHER} range_t;

range_t find_range(float x)
{
    int result;
    if (x < 0)
        result = NEG;
    else if (x == 0)
        result = ZERO;
    else if (x > 0)
        result = POS;
    else
        result = OTHER;
    return result;
}
```

```
    range_t find_range(float x)
    x in %xmm0
1   find_range:
2     vxorps  %xmm1, %xmm1, %xmm1          Set %xmm1 = 0
3     vucomiss        %xmm0, %xmm1         Compare 0:x
4     ja      .L5                          If >, goto neg
5     vucomiss        %xmm1, %xmm0         Compare x:0
6     jp      .L8                          If NaN, goto posornan
7     movl    $1, %eax                     result = ZERO
8     je      .L3                          If =, goto done
9   .L8:                                 posornan:
10    vucomiss        .LC0(%rip), %xmm0    Compare x:0
11    setbe   %al                          Set result = NaN ? 1 : 0
12    movzbl  %al, %eax                    Zero-extend
13    addl    $2, %eax                     result += 2 (POS for > 0, OTHER for NaN)
14    ret                                  Return
15  .L5:                                 neg:
16    movl    $0, %eax                     result = NEG
17  .L3:                                 done:
18    rep; ret                           Return
```