Lab Review and Questions

- >> << | & ^ ?

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 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

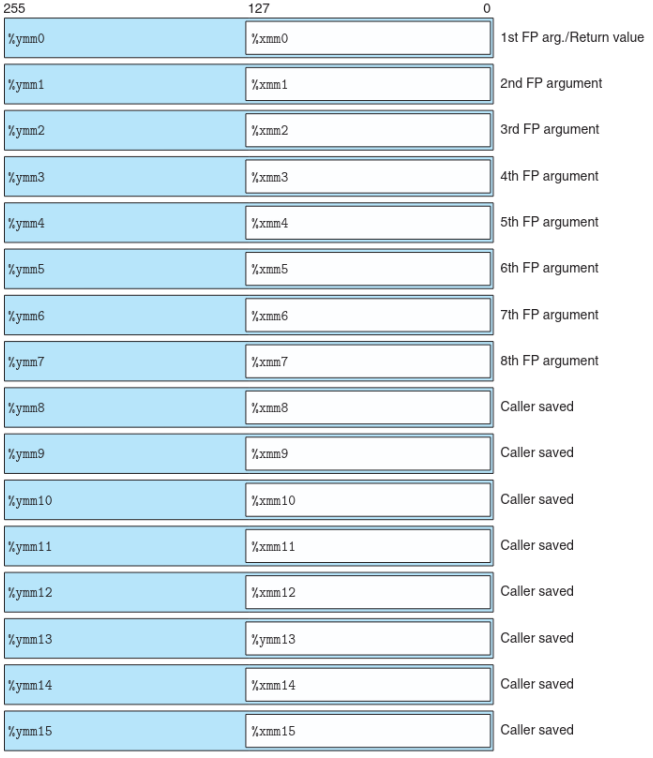


Figure 3.45 **Media registers.** These registers are used to hold floating-point data. Each YMM register holds 32 bytes. The low-order 16 bytes can be accessed as an XMM register.

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

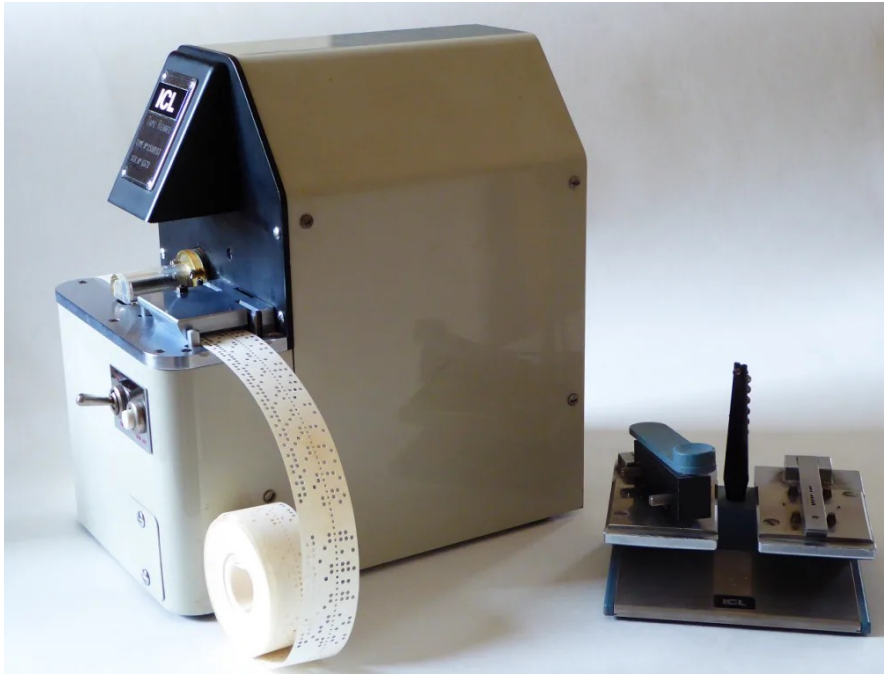
```c
typedef enum {NEG, ZERO, POS, OTHER} range_t;

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
```

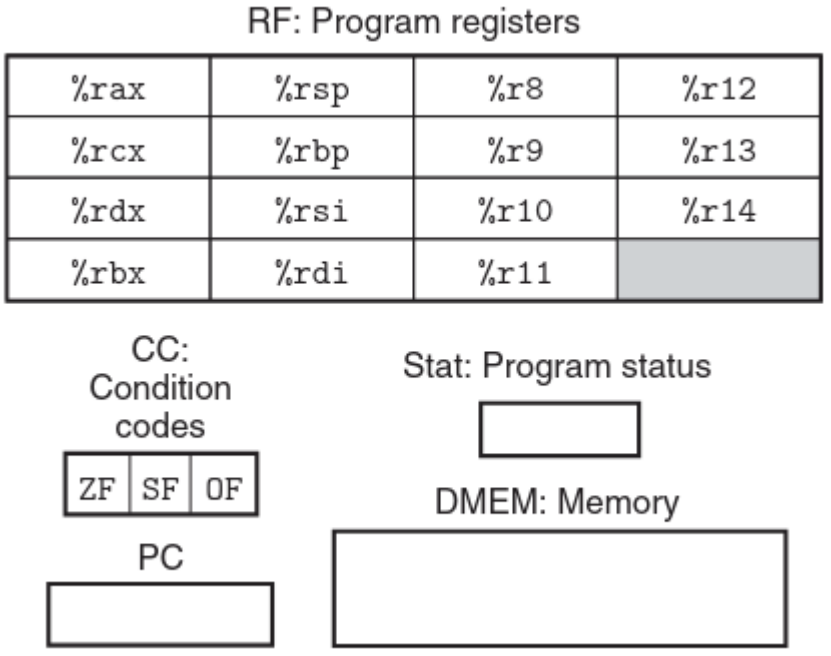
Processor Architecture

- Y86-64 is NOT real! ;-)
- Serialized
  - Tutorial arch for class



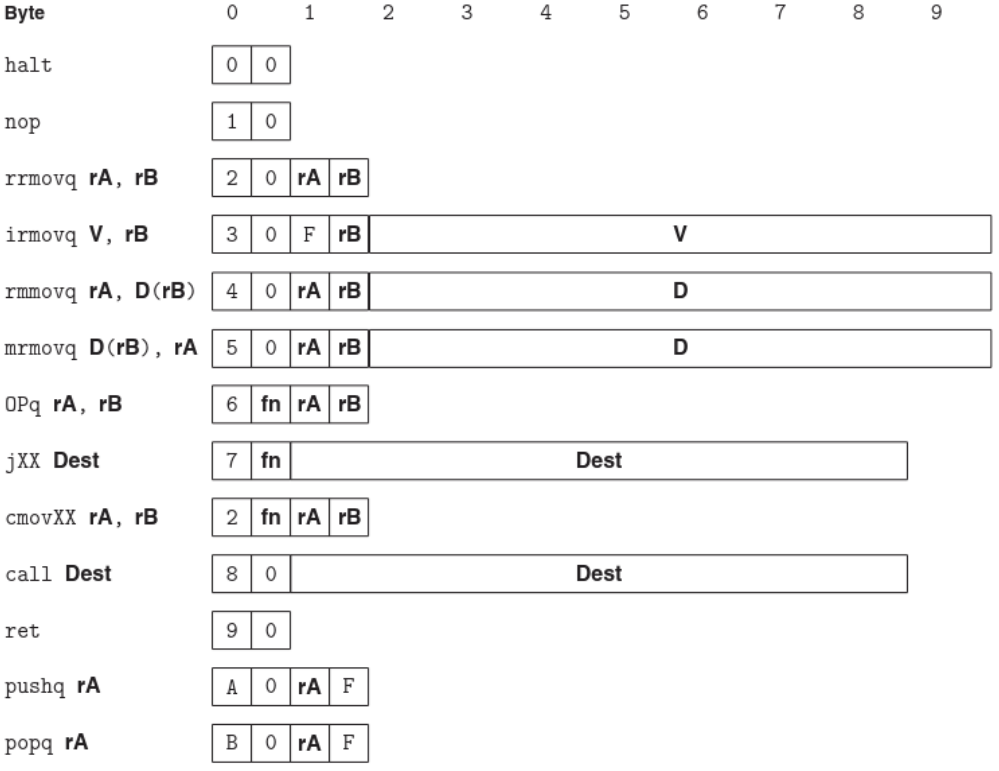
Processor Architecture

- Virtualized state of the system



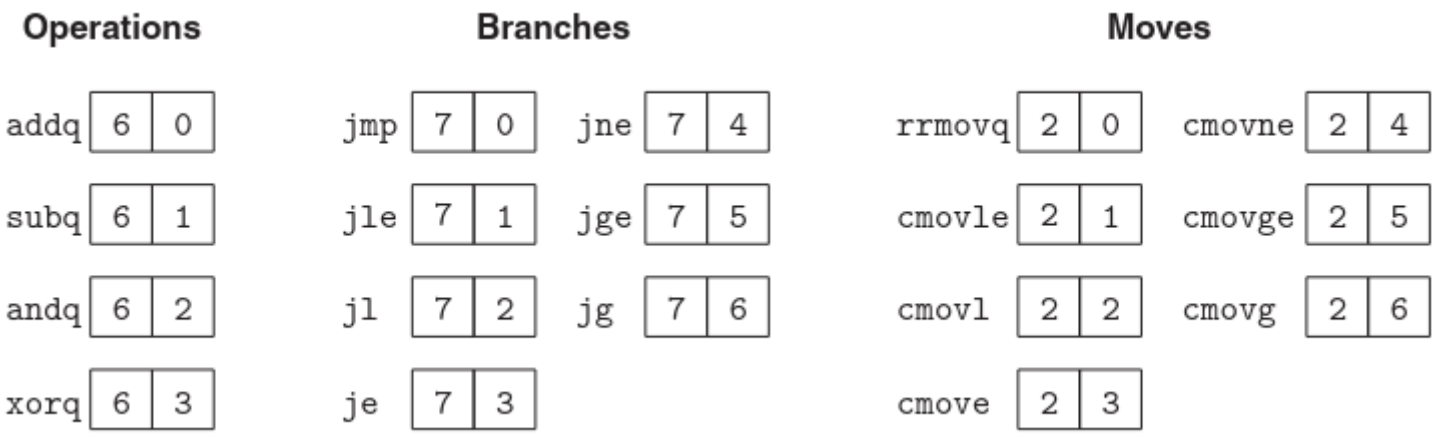
Processor Architecture

- Simplified Instruction Set



Processor Architecture

- Simplified Instruction Set



Processor Architecture

- Simplified Register Layout

Number	Register name	Number	Register name
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	No register

Y86-64 ASM

- Different Implementations

```
1   long sum(long *start, long count)
2   {
3       long sum = 0;
4       while (count) {
5           sum += *start;
6           start++;
7           count--;
8       }
9       return sum;
10  }
```

Y86-64 ASM

- Different Implementations

x86-64 code

```
    long sum(long *start, long count)
    start in %rdi, count in %rsi
1   sum:
2     movl    $0, %eax          sum = 0
3     jmp     .L2               Goto test
4   .L3:                      loop:
5     addq    (%rdi), %rax      Add *start to sum
6     addq    $8, %rdi          start++
7     subq    $1, %rsi          count--
8   .L2:                      test:
9     testq   %rsi, %rsi        Test sum
10    jne     .L3               If !=0, goto loop
11    rep; ret                  Return
```

```
1   long sum(long *start, long count)
2   {
3       long sum = 0;
4       while (count) {
5           sum += *start;
6           start++;
7           count--;
8       }
9       return sum;
10  }
```

Y86-64 ASM

- Different Implementations

Y86-64 code

```
    long sum(long *start, long count)
    start in %rdi, count in %rsi
1   sum:
2     irmovq $8,%r8          Constant 8
3     irmovq $1,%r9          Constant 1
4     xorq %rax,%rax         sum = 0
5     andq %rsi,%rsi         Set CC
6     jmp     test           Goto test
7   loop:
8     mrmovq (%rdi),%r10     Get *start
9     addq %r10,%rax         Add to sum
10    addq %r8,%rdi          start++
11    subq %r9,%rsi          count--.  Set CC
12  test:
13    jne    loop            Stop when 0
14    ret                    Return
```

x86-64 code

```
    long sum(long *start, long count)
    start in %rdi, count in %rsi
1   sum:
2     movl    $0, %eax         sum = 0
3     jmp     .L2              Goto test
4   .L3:                     loop:
5     addq    (%rdi), %rax     Add *start to sum
6     addq    $8, %rdi         start++
7     subq    $1, %rsi         count--
8   .L2:                     test:
9     testq   %rsi, %rsi       Test sum
10    jne     .L3              If !=0, goto loop
11    rep; ret                 Return
```