+ Machine Level Programming: x86-64 History

Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
 - Backwards compatible up until 8086, introduced in 1978
 - Added more features as time goes on
- Complex instruction set computer (CISC)
 - Many different instructions with many different formats
 - But, only small subset encountered with typical programs
 - Hard to match performance of Reduced Instruction Set Computers (RISC)
 - But, Intel has done just that!
 - In terms of speed. Less so for low power.

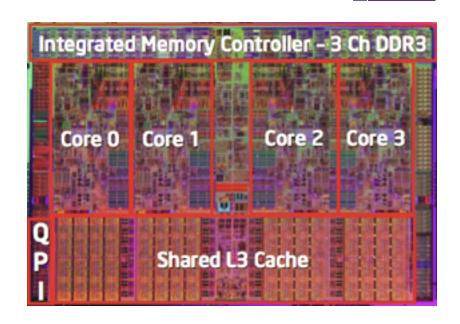
+ Intel x86 Evolution: Milestones

Name	Date	Transistors	MHz
 8086 First 16-bit 1MB address 	•	29K Basis for IBM PC & DOS	5-10
		275K referred to as IA32 pable of running Unix	16-33
Pentium 4EFirst 64-bit	2004 Intel x86 process	125M sor, referred to as x86-64	2800-3800
Core 2First multi-c	2006 core Intel process	291M sor	1060-3500
Core i7Four cores	2008	731M	1700-3900

Intel x86 Processors, cont.



386	1985	0.3M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M

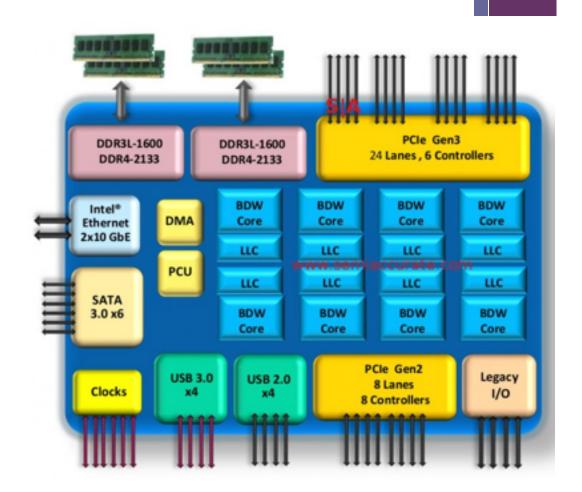


Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores

+ 2015 State of the Art

- Core i7 Broadwell 2015
- Desktop Model
 - 4 cores
 - Integrated graphics
 - 3.3-3.8 GHz
 - 65W
- Server Model
 - 8 cores
 - Integrated I/O
 - 2-2.6 GHz
 - 45W



x86 Clones: Advanced Micro Devices (AMD)

Historically

- AMD has followed just behind Intel
- A little bit slower, a lot cheaper

Then

- Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

Recent Years

- Intel got its act together
 - Leads the world in semiconductor technology
- AMD has fallen behind
 - Relies on external semiconductor manufacturer

Intel's 64-Bit History

2001: Intel Attempts Radical Shift from IA32 to IA64

- Totally different architecture (Itanium)
- Executes IA32 code only as legacy
- Performance disappointing
- 2003: AMD Steps in with Evolutionary Solution
 - x86-64 (now called "AMD64")
- Intel Felt Obligated to Focus on IA64
 - Hard to admit mistake or that AMD is better
- 2004: Intel Announces EM64T extension to IA32
 - Extended Memory 64-bit Technology
 - Almost identical to x86-64!
- All but low-end x86 processors support x86-64
 - But, lots of code still runs in 32-bit mode

Our Coverage

x86-64

- The standard
- Emitted by commands like...
 - \$ gcc hello.c

Book

- Book covers x86-64
- This is why the latest edition is critical.
- Prior to this it was 32-bit
- Rare case where new edition of textbook is actually worth it!

C, Assembly & Machine code

Definitions

- Architecture: (also ISA: instruction set architecture): The parts of a processor design that one needs to understand or write assembly/ machine code.
 - Examples: instruction set specification, registers.
 - Target of compiler

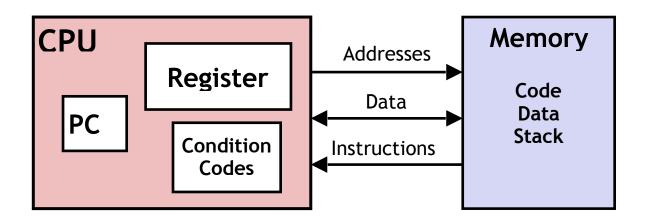
Code Forms:

- Machine Code: The byte-level programs that a processor executes
- Assembly Code: A text representation of machine code

Example ISAs:

- Intel: x86, IA32, Itanium, x86-64
- ARM: Used in almost all mobile phones

Assembly/Machine Code View



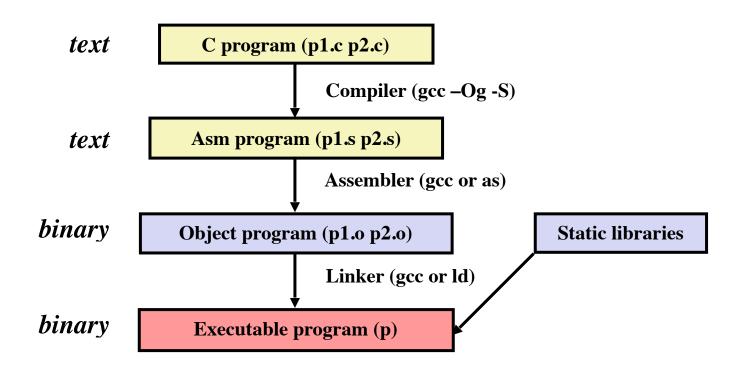
Programmer-Visible State

- Program counter
 - Address of next instruction
 - Called "RIP" (x86-64)
- Register file
 - Heavily used program data
- Condition codes
 - Status info on most recent operation
 - Used for conditional branching

- Memory
 - Byte addressable array
 - Code and user data
 - Stack to support procedures

Turning C into Object Code

- Code in files **p1.c p2.c**
- Compile with command: gcc –Og p1.c p2.c -o p
 - Use basic optimizations (-Og)
 - Put resulting binary in file p



Compiling Into Assembly

```
long plus(long x, long y) {
  return x + y;
}

void sumstore(long x, long y, long *dest) {
   long t = plus(x, y);
   *dest = t;
}
```

```
C source file sum.c
```

```
sumstore:
   pushq %rbx
   movq %rdx, %rbx
   call plus
   movq %rax, (%rbx)
   popq %rbx
   ret
```

Generated x86-64 Assembly

- Generated using command: gcc Og S sum.c
 - -Og tells gcc "do very little optimization".
- Produces file sum.s
- **Note:** Will get very different results different machines due to different versions of gcc and different compiler settings.
- **Note:** For now we ignore all instructions that begin with a dot (.)

Assembly Characteristics: Data Types

Integers

- 1, 2, 4, or 8 bytes
- Bit values (unsigned or not, doesn't matter!)
- Addresses (void pointers)

Floating point numbers

Floating point data of 4 or 8 bytes

Code

Byte sequences encoding series of instructions

Data structures

 No aggregate types such as arrays or structures, just contiguously allocated bytes in memory

Assembly Characteristics: Operations

- Perform arithmetic functions on register or memory data
- Transfer data between memory and register
 - Load data from memory into register
 - Store register data into memory
- Transfer control
 - Unconditional jumps to/from procedures
 - Conditional branches and loops
- Very limited in what can be done in one instruction does only one thing: move data, single simple arithmetic operation, memory dereference.

From Assembly to Object Code



sumstore

0×0400595 : 0x530x480x890xd30xe8 0xf20xff 0xff 0xff 0x480x89 0×03 0x5b0xc3

Total of 14 bytes. Each instruction 1, 3, or 5 bytes. Starts at address 0x0400595

Assembler

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

Linker

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

Machine Instruction Example



```
*dest = t;
```

C Code

• Store value **t** where designated by **dest**

```
movq %rax, (%rbx)
```

Assembly

Move 8-byte value to memory

• Operands:

t: Register %rax

dest: Register %rbx

*dest: Memory M[%rbx]

0x40059e: 48 89 03

Object Code

• 3-byte instruction

Stored at address 0x40059e

Disassembling Object Code

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

```
0000000000400595 <sumstore>:
  400595:
           53
                                 push
                                        %rbx
  400596:
           48 89 d3
                                        %rdx,%rbx
                                 mov
           e8 f2 ff ff ff
  400599:
                                        400590 <plus>
                                 callq
  40059e:
           48 89 03
                                        %rax,(%rbx)
                                 mov
  4005a1:
           5b
                                        %rbx
                                 pop
  4005a2:
           c3
                                 retq
```

+ ,

Alternate Disassember

- Within gdb debugger disassemble sumstore
- Disassemble procedure x/14xb sumstore
 - Examine the 14 bytes starting at sumstore

```
Dump of assembler code for function sumstore:
 0 \times 00000000000400595 <+0>:
                             push
                                     %rbx
 0 \times 000000000000400596 <+1>:
                                    %rdx,%rbx
                             mov
 0x0000000000400599 <+4>:
                             callq 0x400590 <plus>
 0x000000000040059e <+9>:
                                     %rax,(%rbx)
                             mov
 0x00000000004005a1 <+12>:
                                     %rbx
                             pop
 0x00000000004005a2 <+13>:
                              retq
```

What Can be Disassembled?

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source
- Might be illegal

```
% objdump -d WINWORD.EXE
WINWORD.EXE: file format pei-i386
No symbols in "WINWORD.EXE".
Disassembly of section .text:
30001000 <.text>:
30001000: 55
                               %ebp
                      push
30001001: 8b ec
                               %esp,%ebp
                        mov
30001003: 6a ff
                               $0xffffffff
                        push
30001005: 68 90 10 00 30 push
                               $0x30001090
3000100a: 68 91 dc 4c 30 push
                               $0x304cdc91
```

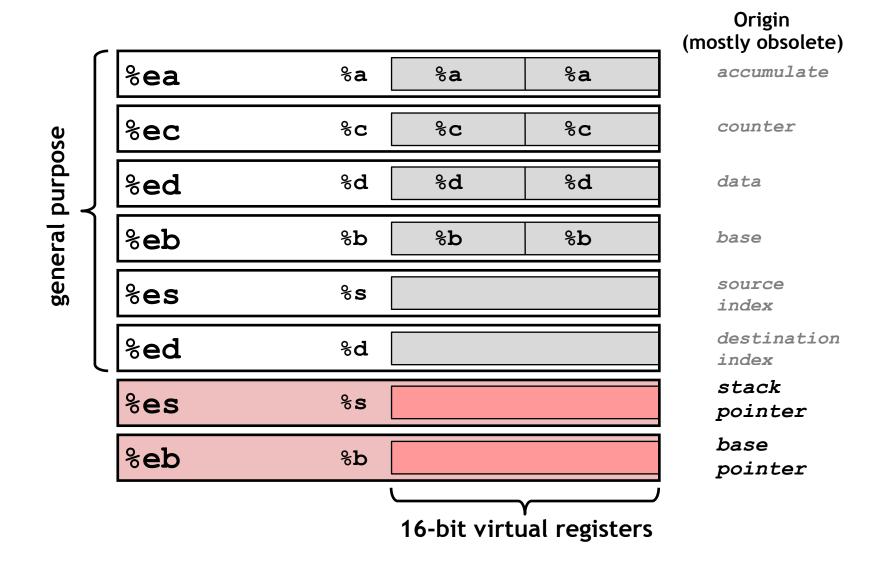
+ Assembly Basics: Registers, Operands, Move

*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

Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Some History: IA32 Registers



+ Mox

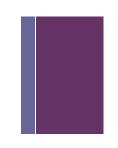
Moving Data

- Moving data: movq src, dest
- Operand Types
 - Immediate: Constant integer data
 - Example: \$0x400, \$-533
 - Like C constant, but prefixed with '\$'
 - Encoded with 1, 2, or 4 bytes
 - **Register**: One of 16 integer registers
 - Example: %rax, %r13
 - But %rsp reserved for special use
 - Others have special uses for particular instructions
 - **Memory**: 8 bytes at address in register
 - Used parens like a dereference (%rax)

%rax	
%rcx	
%rdx	
%rbx	
%rsi	
%rdi	
%rsp	
%rbp	

%rN

movq Operand Combinations



```
Source Dest Src, Dest
             C Analog
```

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

Normal

(R)

- Mem[Reg[R]]
- Register R specifies memory address
- Aha! Pointer dereferencing in C

Displacement

D(R)

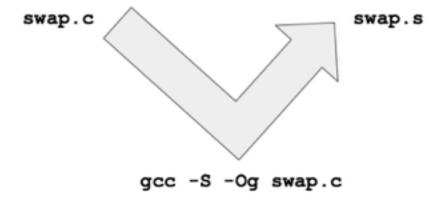
- Mem[Reg[R]+D]
- Register R specifies start of memory region
- Constant displacement D specifies offset

• Note: the normal mode is a special case of displacement mode in which D = 0

Example of Simple Addressing Modes

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

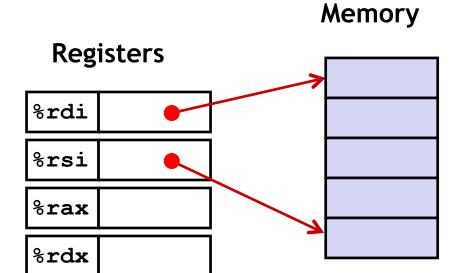
```
swap:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```



Understanding swap()

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

Register	Value
%rdi	хр
%rsi	УÞ
%rax	t0
%rdx	t1



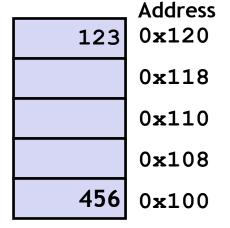
```
swap:
  movq (%rdi), %rax # t0 = *xp
  movq (%rsi), %rdx # t1 = *yp
  movq %rdx, (%rdi) # *xp = t1
  movq %rax, (%rsi) # *yp = t0
  ret
```

Understanding swap() con't

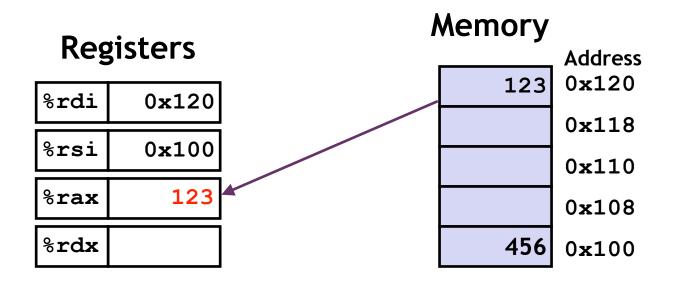


%rdi	0x120
%rsi	0x100
%rax	
%rdx	

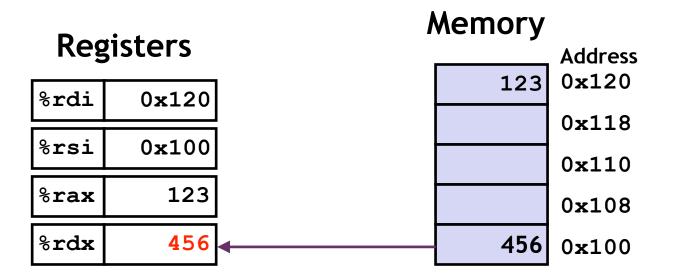
Memory



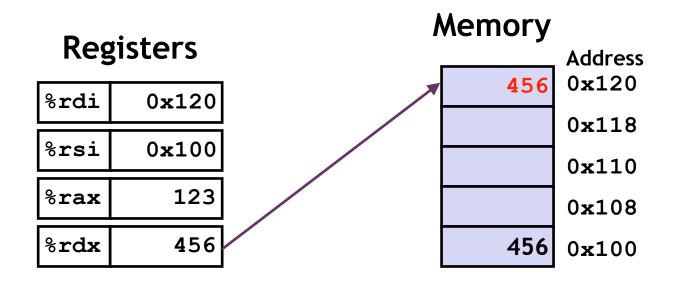
```
swap:
  movq (%rdi), %rax # t0 = *xp
  movq (%rsi), %rdx # t1 = *yp
  movq %rdx, (%rdi) # *xp = t1
  movq %rax, (%rsi) # *yp = t0
  ret
```



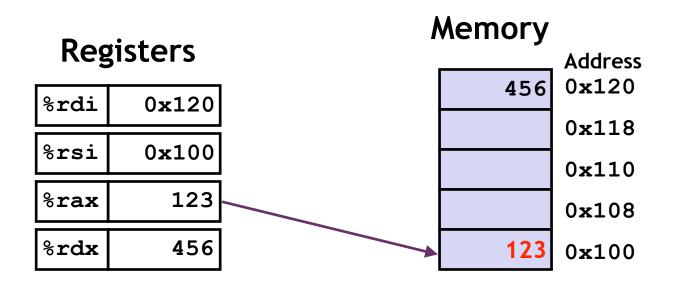
```
movq (%rdi), %rax # t0 = *xp
movq (%rsi), %rdx # t1 = *yp
movq %rdx, (%rdi) # *xp = t1
movq %rax, (%rsi) # *yp = t0
ret
```



```
swap:
  movq (%rdi), %rax # t0 = *xp
  movq (%rsi), %rdx # t1 = *yp
  movq %rdx, (%rdi) # *xp = t1
  movq %rax, (%rsi) # *yp = t0
  ret
```



```
swap:
  movq (%rdi), %rax # t0 = *xp
  movq (%rsi), %rdx # t1 = *yp
  movq %rdx, (%rdi) # *xp = t1
  movq %rax, (%rsi) # *yp = t0
  ret
```



```
swap:
  movq (%rdi), %rax # t0 = *xp
  movq (%rsi), %rdx # t1 = *yp
  movq %rdx, (%rdi) # *xp = t1
  movq %rax, (%rsi) # *yp = t0
  ret
```

Simple Memory Addressing Modes

Normal

(R)

- Mem[Reg[R]]
- Register R specifies memory address
- Aha! Pointer dereferencing in C

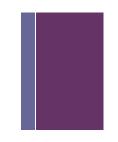
Displacement

D(R)

- Mem[Reg[R]+D]
- Register R specifies start of memory region
- Constant displacement D specifies offset

• Note: the normal mode is a special case of displacement mode in which D = 0

Complete Memory Addressing Modes



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 16 integer registers
- Ri: Index register: Any, except for %rsp
- S: Scale: 1, 2, 4, or 8 (why these numbers?)

Complete Memory Addressing Modes



General form

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for %rsp
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)
- Special cases: you can omit certain arguments if not needed.

(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

%rdx	0xf000
%rcx	0x0100

Base register

Index register

Expression	Address Computation	Address
0x8 (%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400