Sewanee: The University of the South

CS 370: Computer Organization

Machine Representation of Programs I

Instructor:
Professor Stephen P. Carl

Sewanee: The University of the South

Intel x86 Processors

- **Totally dominate *desktop/laptop* computer markets**
  (Distinction is important)
- **Evolutionary design**
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time passed
- **Complex instruction set computer (CISC)**
  - Many different instructions with many different formats
    - But, generally just a small subset is used in most programs
  - While it is considered hard to match performance of Reduced Instruction Set Computers (RISC), Intel has done just that (in terms of speed)!

2015 State of the Art

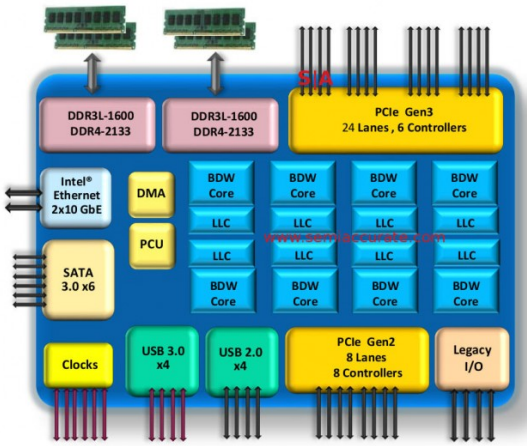
Core i7 Broadwell 2015

Desktop Model

- 4 cores
- Integrated graphics
- 3.3 to 3.8 GHz
- 65 Watts

Server Model

- 8 cores
- Integrated I/O
- 2 to 2.6 GHz
- 45 Watts (important!)



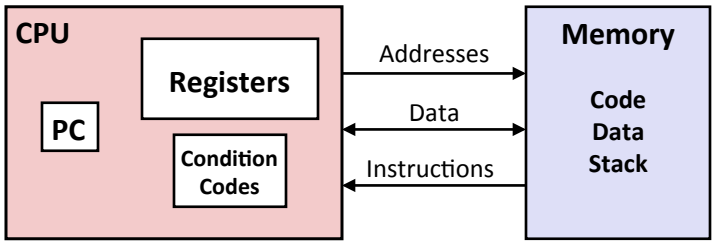
Definitions

- **Architecture: (also ISA: instruction set architecture)**
  The parts of a processor design that one needs to understand to write assembly code.

  (E.g., instruction set, registers)
- **Microarchitecture: Implementation of the architecture.**

  (E.g., cache size, core clock frequency)
- **Code Forms:**
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code
- **Example ISAs: ARM, x86, IA32, x86-64**

Assembly/Machine Code View



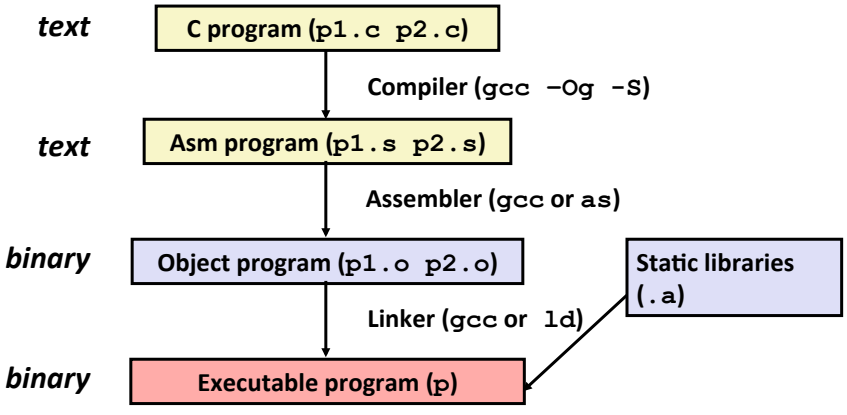
Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical 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 optimizations (`-Og`)
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated IA32 Assembly

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

```
gcc -O -S
code.c
```

Produces file `code.s`

Some compilers use single instruction "`leave`"

Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);

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

Generated x86-64 Assembly

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

Obtained with command

```
gcc –Og –S sum.c
```

Produces file `sum.s`

Warning: Will get very different results on non-Ubuntu machines (Mac OS-X, etc.) due to different versions of gcc and different compiler settings.

Sewanee: The University of the South

Assembly Characteristics: Data Types

- **"Integer" data of 1, 2, 4 or 8 bytes**
  - Data values
  - Addresses (untyped pointers)
- **Floating point data of 4, 8, or 10/12 bytes**
- **Code: Byte sequences encoding series of instructions**
- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

Sewanee: The University of the South

Assembly Characteristics: Operations

- **Arithmetic functions on register or memory data**
- **Transfer data between memory and register**
  - Load data from memory into register
  - Store data from register into memory
- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

Sewanee: The University of the South

Object Code

Code for `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

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

Assembler

- Translates `.s` into `.o`
- Encodes each instruction in binary
- Nearly-complete image of executable code
- Still need linkage to code in different files

Linker

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

Sewanee: The University of the South

Machine Instruction Example

```
*dest = t;
```

↓

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

↓

```
0x40059e:  48 89 03
```

C Code

- Store value **`t`** into address given by **`dest`**

Assembly

- Move 8-byte value to memory
  - *Quad words* in x86-64 parlance
- Operands:
  - **`t`**: Register **`%rax`**
  - **`dest`**: Register **`%rbx`**
  - **`*dest`**: Memory **`M[%rbx]`**

Object Code

- 3-byte instruction
- Stored at address **`0x40059e`**

Disassembling Object Code

Disassembled

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

Disassembler

`objdump -d p`

- 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

Reading Byte-Reversed Listings

Example Fragment:

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add $0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl $0x0,0x28(%ebx)

Deciphering Numbers

- Value: 0x12ab
- Pad to 32 bits: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00

Alternate Disassembly

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

Using gdb Debugger:

```
gdb p
disassemble sum
x/14xb sum
```

(Examine the 14 bytes starting at `sum`)

What Can be Disassembled?

Anything that can be interpreted as executable code:

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by Microsoft End User License Agreement

Disassembler examines bytes and reconstructs assembly source

Sewanee: The University of the South

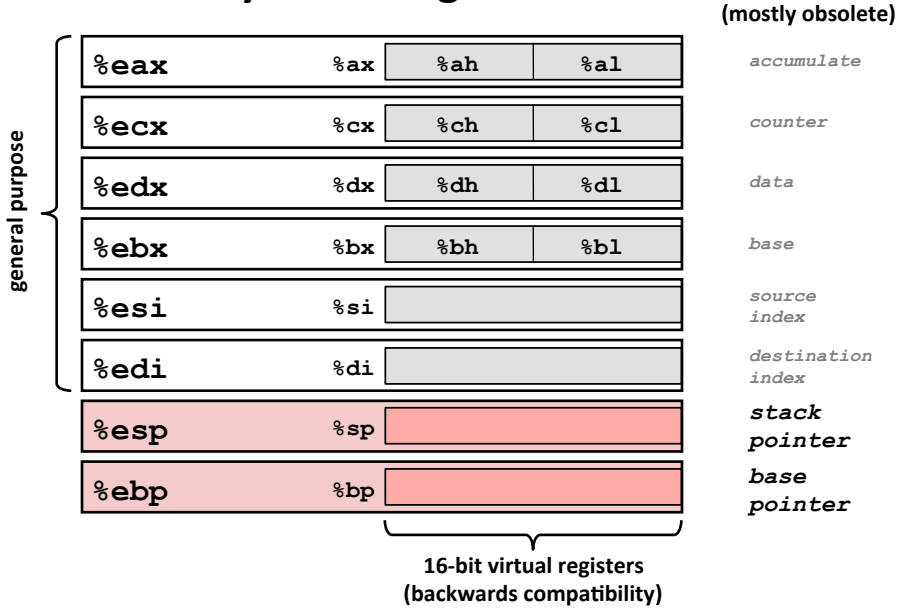
x86-64 Integer Registers

%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)

Sewanee: The University of the South

Some History: IA32 Registers



Moving Data

Moving Data instruction:

`movq` ***Source*, *Dest*:**

Operand Types:

Immediate: Constant integer data

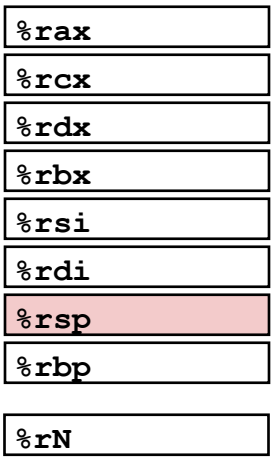
- Example: `$0x400, $-533`
- Like C constant, but prefixed with `'$'`
- Encoded with 1, 2, or 4 bytes

Register: One of 16 integer registers

- Example: `%rax, %r13`
- **But** `%rsp` reserved for special use
- Others have special uses for particular instructions

Memory: 8 consecutive bytes of memory at address given by register

- Simplest example: `(%rax)`
- Various other "address modes"



`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

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

```
movq (%rcx),%rax
```

Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```


Example of Simple Addressing Modes

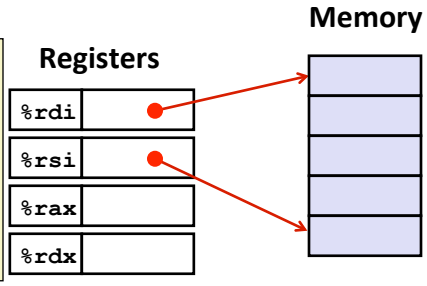
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

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

Sewanee: The University of the South

Understanding Swap()

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

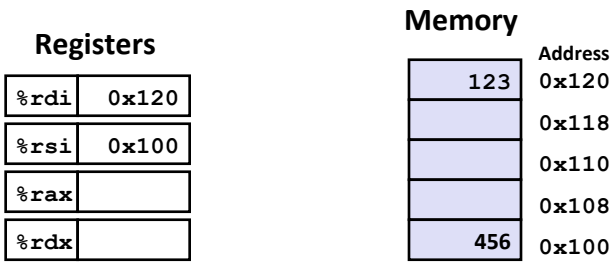


Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

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

Sewanee: The University of the South

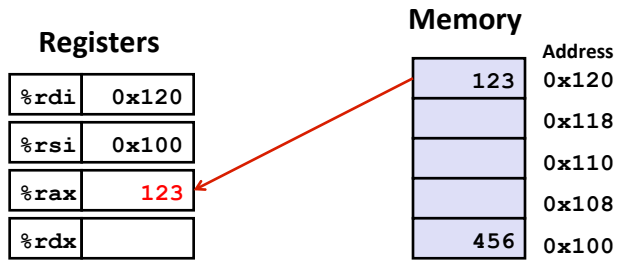
Understanding Swap()




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

Sewanee: The University of the South

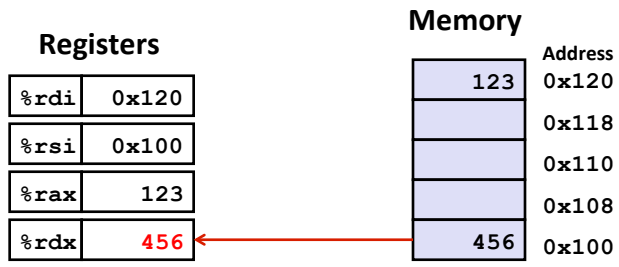
Understanding Swap()




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

Sewanee: The University of the South

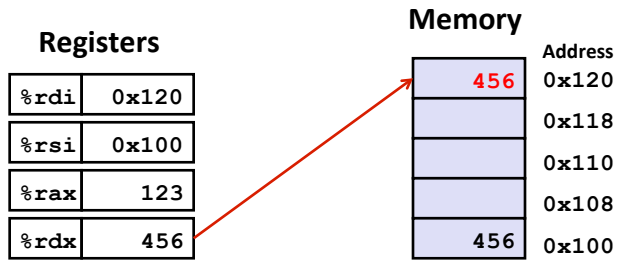
Understanding Swap()




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

Sewanee: The University of the South

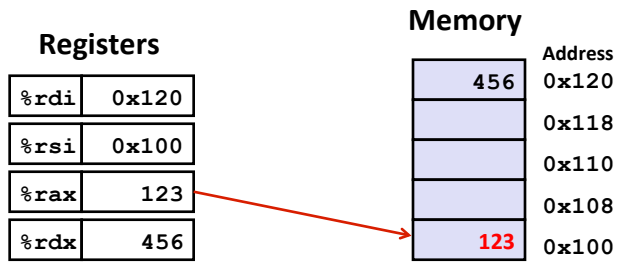
Understanding Swap()




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

Sewanee: The University of the South

Understanding Swap()




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

Sewanee: The University of the South

Complete Memory Addressing Modes

Most General Form of addressing:

Operand Format	Operand Value
$D(r_b, r_i, s)$	$Mem[Reg[r_b]+s*Reg[r_i]+ D]$

- D: Constant "displacement" 1, 2, or 4 bytes
- r_b: Base register: Any of 16 integer registers
- r_i: Index register: Any, except for **`%rsp`**
- *s*: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Special Cases

(r_b, r_i)	**$Mem[Reg[r_b]+Reg[r_i]]$**
$D(r_b, r_i)$	**$Mem[Reg[r_b]+Reg[r_i]+D]$**
(r_b, r_i, s)	**$Mem[Reg[r_b]+s*Reg[r_i]]$**

Sewanee: The University of the South Carnegie Mellon

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Sewanee: The University Carnegie Mellon

Address Computation Instruction

`leaq` Src, Dst

- Src is address mode expression
- Set Dst 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

Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Sewanee: The University Carnegie Mellon

Some Arithmetic Operations

Two Operand Instructions:

Format		Computation
`addq`	Src,Dest	Dest = Dest + Src
`subq`	Src,Dest	Dest = Dest - Src
`imulq`	Src,Dest	Dest = Dest * Src
`salq`	Src,Dest	Dest = Dest << Src Also called shlq
`sarq`	Src,Dest	Dest = Dest >> Src Arithmetic Shift
`shrq`	Src,Dest	Dest = Dest >> Src Logical Shift
`xorq`	Src,Dest	Dest = Dest ^ Src
`andq`	Src,Dest	Dest = Dest & Src
`orq`	Src,Dest	Dest = Dest | Src

Watch out for argument order!

No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

- **One Operand Instructions**

`incq`	Dest	Dest = Dest + 1
`decq`	Dest	Dest = Dest - 1
`negq`	Dest	Dest = - Dest
`notq`	Dest	Dest = ~Dest

- **See book for more instructions**

Arithmetic Expression Example

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

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

Sewanee: The University of the South Carnegie Mellon

Understanding the Example

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

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

Sewanee: The University of the South

Machine Programming I: Summary

- **History of Intel processors and architectures**
  - Evolutionary design leads to many quirks and artifacts
- **C, assembly, machine code**
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- **Assembly Basics: Registers, operands, move**
  - The x86-64 move instructions cover wide range of data movement forms
- **Arithmetic**
  - C compiler generates instruction combinations to carry out computation