CS 105
"Tour of the Black Holes of Computing"

Machine-Level Programming I

Topics

- Assembly Programmer's Execution Model
- Accessing Information
  - Registers
  - Memory
- Arithmetic operations

Intel x86 (IA32/64) Processors

Totally Dominate Computer Market

Evolutionary Design

- Starting in 1978 with 8086 (really 1971 with 4004)
- Added more features as time went on
- Still support old features, although obsolete

Complex Instruction Set Computer (CISC)

- Many different instructions with many different formats
  - But only small subset encountered with Linux programs
- Hard to match performance of Reduced Instruction Set Computers (RISC)
- But Intel has done just that!
  - Well…in terms of speed; less so for low power

X86 Evolution: Milestones

Name	Date	Transistors	Frequency
4004	1971	2.3K	108 KHz

- 4-bit processor. First 1-chip microprocessor
- Didn't even have interrupts!

Name	Date	Transistors	Frequency
8008	1972	3.3K	200-800 KHz

- Like 4004, but with 8-bit ALU

Name	Date	Transistors	Frequency
8080	1974	6K	2 MHz

- Compatible at source level with 8008
- Processor in first "kit" computers
- Pricing caused it to beat similar processors with better programming models
  - Motorola 6800 (best of the bunch, IMO)
  - MOS Technologies (MOSTEK) 6502 (used in Apple II)

X86 Evolution: Milestones

Name	Date	Transistors	Frequency
8086	1978	29K	5-10 MHz

- 16-bit processor. Basis for IBM PC & DOS
- Limited to 1MB address space. DOS only gives you 640K

Name	Date	Transistors	Frequency
80286	1982	134K	4-12 MHz

- Added elaborate, but not very useful, addressing scheme
- Basis for IBM PC-AT and Windows

Name	Date	Transistors	Frequency
386	1985	275K	16-33 MHz

- Extended to 32 bits. Added "flat addressing"
- Capable of running Unix
- By default, Linux/gcc compiling for 32-bit x86 machines use no instructions introduced in later models

X86 Evolution: Milestones

Name	Date	Transistors	Frequency
486	1989	1.9M	16-150 MHz
Pentium P5	1993	3.1M	60-66 MHz
Pentium 4E	2004	125M	2.8-3.8 GHz

- First 64-bit Intel x86 processor

Core 2	2006	291M	1.0-3.5 GHz

- First multi-core Intel processor

Core i7	2008	731M	1.7-3.9 GHz
Ivy Bridge	2012	0.6-4.3B	3.2-4.0 GHz

- Transistor counts are going crazy here…
- …but max GHz has been stuck since 2004

X86 Evolution: Clones

Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Late 1990s
  - Recruited top circuit designers from Digital Equipment Corp.
  - Exploited fact that Intel distracted by Itanium
  - Became close competitors to Intel
- Developed own extension to 64 bits (called x86_64)
- Intel adopted in early 2000's after Itanium bombed
  - Has recovered lead in semiconductor technology
  - AMD has fallen behind again
  - But in recent years ARM has been rising due to smartphones

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.

Microarchitecture: Implementation of the architecture.

- Examples: cache sizes and core frequency.

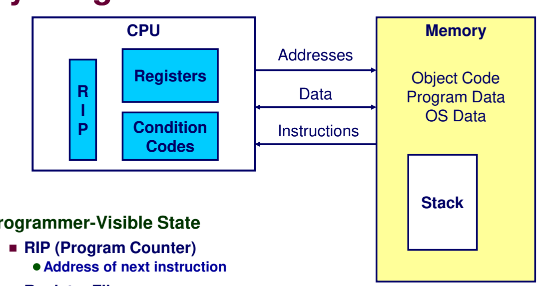
Code Forms:

- **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 smartphones

Assembly Programmer's View



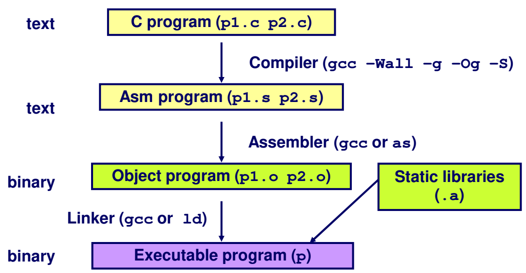
Programmer-Visible State

- RIP (Program Counter)
  - Address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

- Memory
  - Byte-addressable array
  - Code, user data, (most) OS data
  - Includes stack used to support procedures

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Wall -g -Og p1.c p2.c -o p`
  - Use basic, debugging-friendly optimizations (`-Og`)
  - Put resulting binary in file `p`



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
```

Obtain (on Wilkes) with command

`gcc -Og -g -S sum.c`

Produces file `sum.s`

(Note: we're removed a bunch of irrelevant *pseudo-ops* intended for the assembler)

Warning: May get very different results on other machines (Knuth, Mac OS-X, …) due to different versions of gcc and different compiler settings.

Assembly Characteristics

Minimal data types

- Integer data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating-point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Code is also just byte sequences encoding instructions

Primitive operations

- Perform arithmetic function 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

Object Code

Sample 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`
- 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
  - *Quad words* in x86-64 parlance
- 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

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 sum`

- Useful tool for examining object code
- Analyzes bit patterns of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either `a.out` (complete executable) or `.o` file

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
```

Within gdb Debugger

`gdb sum`

`disassemble sumstore`
- Disassembles procedure named `sumstore`

`x/14xb sumstore`
- Examines the 14 hex bytes starting at `sumstore`

`x/6i sumstore`
- Disassembles 6 insructions starting at `sumstore`

What Can be Disassembled?

```
% 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

Anything that can be interpreted as executable code

Disassembler examines bytes and reconstructs assembly source

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`

Moving Data

Moving Data

`movq` *Source*, *Dest*

Operand Types

- *Immediate:* Constant integer data
  - Example: `$0x400`, `$-533`
  - Like C constant, but prefixed with `'$'`
  - Encoded with 1, 2, 4, or 8 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"

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

`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 Addressing Modes

Direct A Mem[A]

- Memory address A is directly specified
- Mostly used for static and global variables

`movl 0x804acb8,%eax`

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`

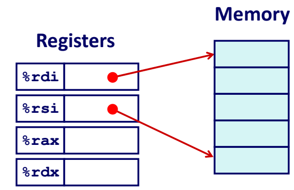
Example of Simple Addressing Modes

```
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
```


Understanding Swap()

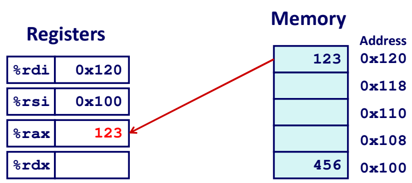
```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

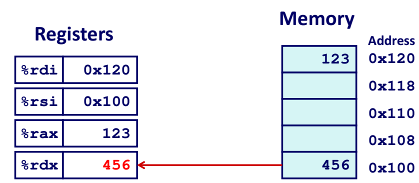
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```


Understanding Swap()



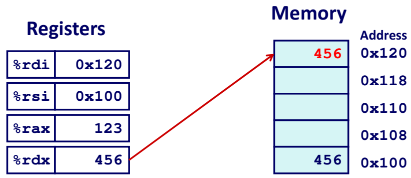

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```


Understanding Swap()



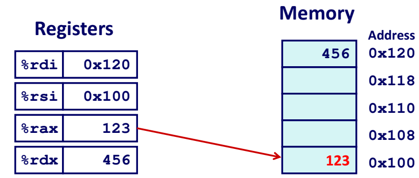

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()




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


Understanding Swap()




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


Simple Addressing Modes

Direct A Mem[A]

- Memory address A is directly specified
- Mostly used for static and global variables

`movl 0x804acb8,%eax`

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`

Complete Addressing Modes

Most General Form

D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]

- D: Constant "displacement" 1, 2, or 4 bytes (but not 8)
  - Can be small (offset) or large (address in first 4GB)
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8

Special Cases

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]	= 0(Rb,Ri,1)
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]	= D(Rb,Ri,1)
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]	= 0(Rb,Ri,S)
D	Mem[D]	= D(,,1)
(,Ri,S)	Mem[S*Reg[Ri]]	= 0(,Ri,S)

Address Computation Examples

%rdx	0xf000
%rcx	0x100

Expression	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

Address Computation Instruction

`leaq` *Src,Dest*

- *Src* is address mode expression
- Set *Dest* to address denoted by expression

Uses

- Computing address without doing memory reference
  - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form x + k*y
  - k = 1, 2, 4, or 8.

LEARN THIS INSTRUCTION!!!

- Used heavily by compiler
- Appears regularly on labs, quizzes, & exams

leaq vs. movq

Assume dest is %rax:

%rdi = 0xF000
%rsi = 0x8
Memory at 0xF000 = 0x12345
Memory at 0xF008 = 0x6789A
Memory at 0xF010 = 0xBCDEF

Src	leaq	movq
(%rdi)	0xF000	0x12345
8(%rdi)	0xF008	0x6789A
(%rdi,%rsi)	0xF008	0x6789A
(%rdi,%rsi,2)	0xF010	0xBCDEF
%rdi	*Illegal!*	0xF000

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*
`shrq`	*Src,Dest*	Dest = Dest >> Src	*Logical*
`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?)

Note: immediate source limited to 4 bytes (sigh)

Some Arithmetic Operations

One-Operand Instructions

`incq`	*Dest = Dest + 1*
`decq`	*Dest = Dest – 1*
`negq`	*Dest = – Dest*
`notq`	*Dest = ~Dest*

See textbook 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!

Understanding `arith`

```
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`