The Hardware/Software Interface

CSE351 Winter 2013

Basics of Machine Programming

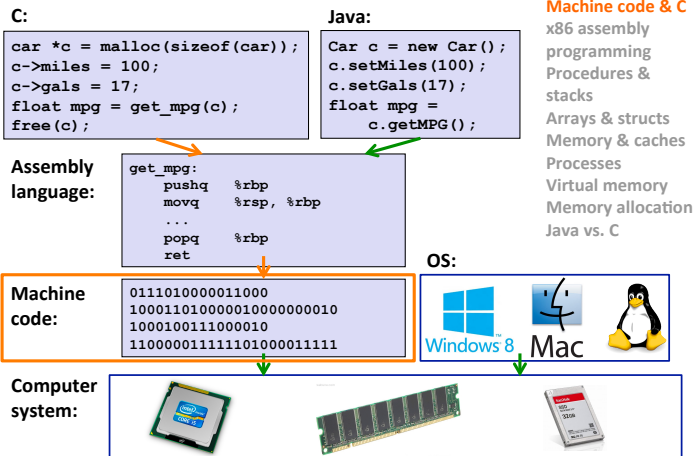
Themes of CSE 351

- **Interfaces and abstractions**
  - So far: some abstractions in C code
    - e.g. various data types: ints, floats, pointers, arrays
  - Today: what interface does the *hardware* present?

- **Representation**
  - So far: integers, floating point numbers, addresses
    - Understanding what's below the C abstractions makes you a better programmer
- **Translation**
  - Today: how do we get from C code to machine code? What machine code should you expect to be produced from your C code?
- **Control flow**

Roadmap

Data & addressing
Integers & floats
Machine code & C
x86 assembly programming
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq    %rbp
    movq     %rsp, %rbp
    ...
    popq     %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8   Mac
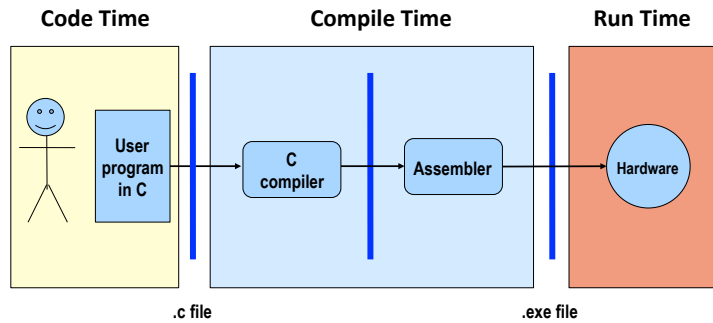
Computer system:

Today's Topics

- **What is an ISA (Instruction Set Architecture)?**
- **A brief history of Intel processors and architectures**
- **C, assembly, machine code**
- **x86 basics: registers**

Translation

Code Time	Compile Time	Run Time

.c file   .exe file

What makes programs run fast?

Translation Impacts Performance

- **The time required to execute a program depends on:**
  - *The program* (as written in C, for instance)
  - *The compiler*: what set of assembler instructions it translates the C program into
  - *The instruction set architecture* (ISA): what set of instructions it makes available to the compiler
  - *The hardware implementation*: how much time it takes to execute an instruction

- **There is a complex interaction among these**

Instruction Set Architectures

- **The ISA defines:**
  - The system's state (e.g. registers, memory, program counter)
  - The instructions the CPU can execute
  - The effect that each of these instructions will have on the system state

---
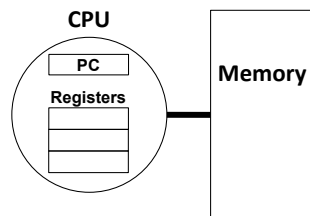
General ISA Design Decisions

- **Instructions**
  - What instructions are available? What do they do?
  - How are they encoded?

- **Registers**
  - How many registers are there?
  - How wide are they?

- **Memory**
  - How do you specify a memory location?

x86

- **Processors that implement the x86 ISA completely dominate the server, desktop and laptop markets**

- **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 Linux programs
  - (as opposed to Reduced Instruction Set Computers (RISC), which use simpler instructions)

Intel x86 Evolution: Milestones

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

- First 16-bit processor. Basis for IBM PC & DOS
- 1MB address space

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

- First 32 bit processor, referred to as IA32
- Added "flat addressing"
- Capable of running Unix
- 32-bit Linux/gcc targets i386 by default

Name	Date	Transistors	MHz
Pentium 4F	**2005**	**230M**	**2800-3800**

- First 64-bit Intel x86 processor, referred to as x86-64

Intel x86 Processors

- **Machine Evolution**

486	1989	1.9M
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

Intel Core i7

- **Added Features**
  - Instructions to support multimedia operations
    - Parallel operations on 1, 2, and 4-byte data
  - Instructions to enable more efficient conditional operations
  - More cores!
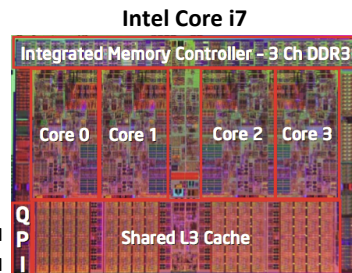
More information

- **References for Intel processor specifications:**
  - Intel's "automated relational knowledgebase":
    - http://ark.intel.com/
  - Wikipedia:
    - http://en.wikipedia.org/wiki/List_of_Intel_microprocessors

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 and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension of x86 to 64 bits

Intel's Transition to 64-Bit

- **Intel attempted radical shift from IA32 to IA64 (2001)**
  - Totally different architecture (Itanium) and ISA than x86
  - Executes IA32 code only as legacy
  - Performance disappointing
- **AMD stepped in with *evolutionary* solution (2003)**
  - x86-64 (also called "AMD64")
- **Intel felt obligated to focus on IA64**
  - Hard to admit mistake or that AMD is better
- **Intel announces "EM64T" extension to IA32 (2004)**
  - Extended Memory 64-bit Technology
  - Almost identical to AMD64!
- **Today: all but low-end x86 processors support x86-64**
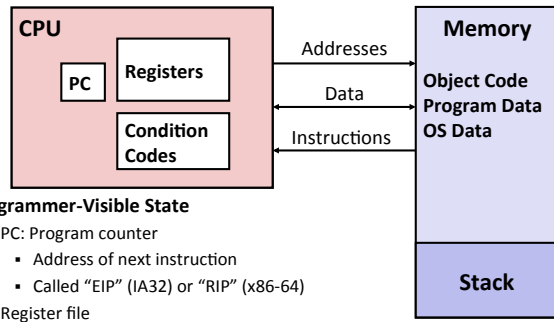 - But, lots of code out there is still just IA32

Our Coverage in 351

- **IA32**
  - The traditional x86

- **x86-64**
  - The emerging standard – all lab assignments use x86-64!

Definitions

- **Architecture: (also instruction set architecture or ISA) The parts of a processor design that one needs to understand to write assembly code**
  - "What is directly visible to software"
  - Includes: instruction set specification, registers, memory model
- **Microarchitecture: Implementation of the architecture**
  - Includes: CPU frequency, cache sizes, other implementation details

- **The ISA is an *abstraction* of the microarchitecture**

Assembly Programmer's View

CPU	
PC	Registers
	Condition Codes

Addresses
Data
Instructions

Memory

Object Code
Program Data
OS Data

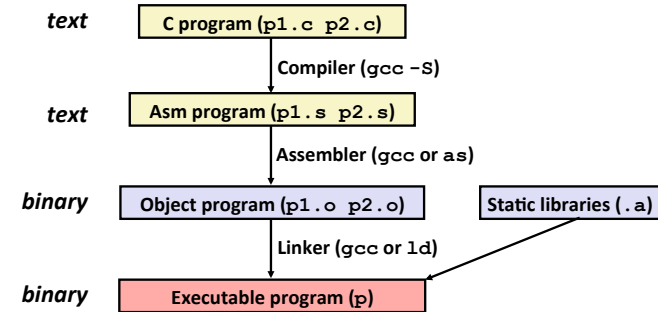**Stack**

- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called "EIP" (IA32) or "RIP" (x86-64)
  - 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, (some) OS data
  - Includes stack used to support procedures (we'll come back to that)

Turning C into Object Code

- **Code in files** `p1.c p2.c`
- **Compile with command:** `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

text → C program (`p1.c p2.c`)

↓ Compiler (`gcc -S`)

text → Asm program (`p1.s p2.s`)

↓ Assembler (`gcc or as`)

binary → Object program (`p1.o p2.o`)   Static libraries (`.a`)

↓ Linker (`gcc or ld`)

binary → Executable program (`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 -O1 -S code.c`

Produces file `code.s`

Three Basic Kinds of Instructions

- **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

Assembly Characteristics: Data Types

- **"Integer" data of 1, 2, 4 (IA32), or 8 (just in x86-64) bytes**
  - Data values
  - Addresses (untyped pointers)

- **Floating point data of 4, 8, or 10 bytes**

- **What about "aggregate" types such as arrays or structs?**
  - No aggregate types, just contiguously allocated bytes in memory

Object Code

Code for `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- **Total of 13 bytes**
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address 0x401040**
- **Not at all obvious where each instruction starts and ends**

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

- **Linker**
  - Resolves references between object files and (re)locates their data
  - 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

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression:

    x += y

More precisely:

    int eax;
    int *ebp;
    eax += ebp[2]

```
0x401046:    03 45 08
```

- **C Code:** add two signed integers
- **Assembly**
  - Add two 4-byte integers
    - "Long" words in GCC speak
    - Same instruction whether signed or unsigned
  - Operands:
    - **x:** Register **%eax**
    - **y:** Memory **M[%ebp+8]**
    - **t:** Register **%eax**
      - Return function value in **%eax**
- **Object Code**
  - 3-byte instruction
  - Stored at address **0x401046**

Disassembling Object Code

Disassembled

```
00401040 <_sum>:
   0:      55              push    %ebp
   1:      89 e5           mov     %esp,%ebp
   3:      8b 45 0c        mov     0xc(%ebp),%eax
   6:      03 45 08        add     0x8(%ebp),%eax
   9:      89 ec           mov     %ebp,%esp
   b:      5d              pop     %ebp
   c:      c3              ret
```

- **Disassembler**
  `objdump -d p`
  - Useful tool for examining object code (`man 1 objdump`)
  - Analyzes bit pattern of series of instructions (delineates instructions)
  - Produces near-exact rendition of assembly code
  - Can be run on either `p` (complete executable) or `p1.o`/`p2.o` file

Alternate Disassembly

Object	Disassembled

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Disassembled

```
0x401040 <sum>:       push    %ebp
0x401041 <sum+1>:     mov     %esp,%ebp
0x401043 <sum+3>:     mov     0xc(%ebp),%eax
0x401046 <sum+6>:     add     0x8(%ebp),%eax
0x401049 <sum+9>:     mov     %ebp,%esp
0x40104b <sum+11>:    pop     %ebp
0x40104c <sum+12>:    ret
```

- **Within gdb debugger**
  - `gdb p`
  - `disassemble sum`
  - (disassemble function)
  - `x/13b sum`
  - (examine the 13 bytes starting at `sum`)

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:  55                push    %ebp
30001001:  8b ec             mov     %esp,%ebp
30001003:  6a ff             push    $0xffffffff
30001005:  68 90 10 00 30    push    $0x30001090
3000100a:  68 91 dc 4c 30    push    $0x304cdc91
```
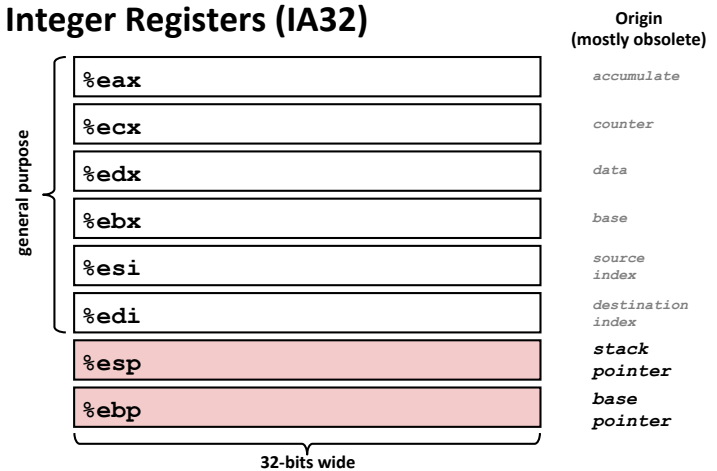
- **Anything that can be interpreted as executable code**
- **Disassembler examines bytes and reconstructs assembly source**

What Is A Register?

- **A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)**

- **Registers are at the heart of assembly programming**
  - They are a precious commodity in all architectures, but *especially* x86

Integer Registers (IA32)

Origin (mostly obsolete)

general purpose:
- `%eax` — *accumulate*
- `%ecx` — *counter*
- `%edx` — *data*
- `%ebx` — *base*
- `%esi` — *source index*
- `%edi` — *destination index*

- `%esp` — *stack pointer*
- `%ebp` — *base pointer*

32-bits wide

Integer Registers (IA32)

Origin (mostly obsolete)

%eax	**%ax**	**%ah**	**%al**

accumulate

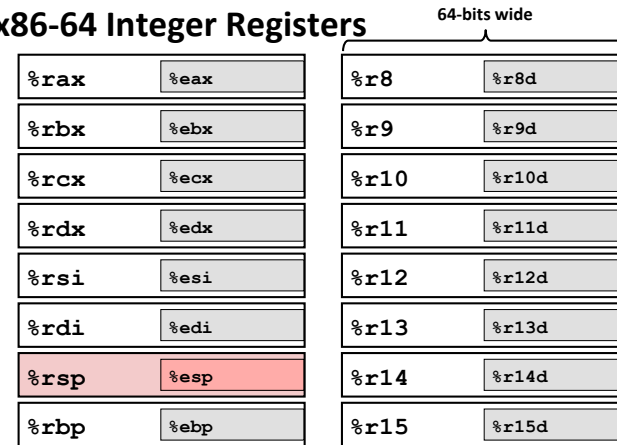|			
%ecx	**%cx**	**%ch**	**%cl**

counter

%edx	**%dx**	**%dh**	**%dl**

data

%ebx	**%bx**	**%bh**	**%bl**

base

%esi	**%si**	

source index

%edi	**%di**	

destination index

%esp	**%sp**	

stack pointer

%ebp	**%bp**	

base pointer

general purpose

16-bit virtual registers (backwards compatibility)

---

x86-64 Integer Registers

64-bits wide

%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**

- Extend existing registers, and add 8 new ones; *all* accessible as 8, 16, 32, 64 bits.

Summary: Machine Programming

- **What is an ISA (Instruction Set Architecture)?**
  - Defines the system's state and instructions that are available to the software
- **History of Intel processors and architectures**
  - Evolutionary design leads to many quirks and artifacts
- **C, assembly, machine code**
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- **x86 registers**
  - Very limited number
  - Not all general-purpose