Machine-Level Programming I: Basics

Kai Shen

1

Why do I care for machine code?

- Chances are, you'll never write programs in machine code
  - Compilers are much better & more patient than you are
- But: understanding machine code is useful and important
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems do weird things (save/restore process state)
  - Access special hardware features
    - Processor model-specific registers
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Creating / fighting malware
    - Machine/assembly code is the language of choice!

2

But if you are really into the hardware

- This course only teaches the interaction with machine hardware, not designing/implementing the hardware
- Read chapter 4
- A guest lecture by Prof. Dwarkhdas
- Go to take the Computer Architecture course in ECE

3

Machine Programming I: Basics

- History of x86 processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64

4

Intel x86 Processors

- Totally dominate laptop/desktop/server market
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!

5

Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
8086	1978	29K	5-10
386	1985	275K	16-33
Pentium 4F	2004	125M	2800-3800
Core i7	2008	731M	2667-3333

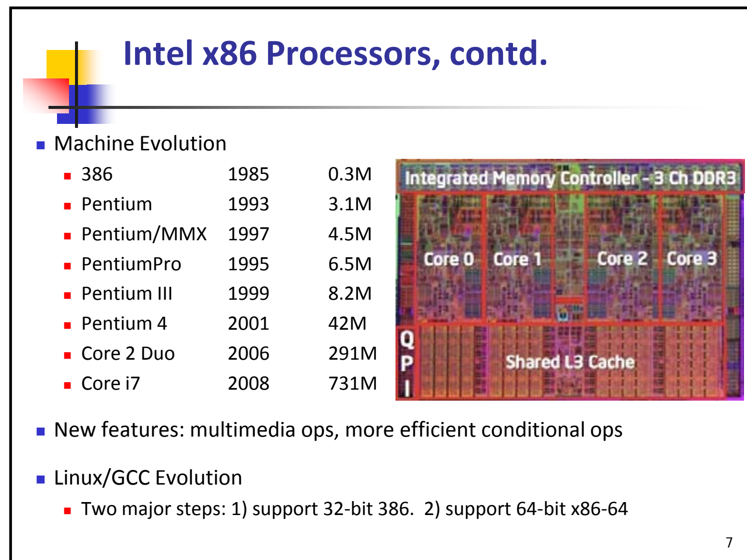
- 8086
  - First 16-bit processor. Basis for IBM PC & DOS
  - 1MB address space
- 386
  - First 32 bit processor , referred to as IA32
  - Capable of running Unix
- Pentium 4F
  - First 64-bit processor, referred to as x86-64
- Core i7
  - Latest, multicore
- Backward compatible

6

Intel x86 Processors, contd.

- Machine Evolution

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



- New features: multimedia ops, more efficient conditional ops
- Linux/GCC Evolution
  - Two major steps: 1) support 32-bit 386. 2) support 64-bit x86-64

7

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

8

Intel's 64-Bit

- Intel attempted radical shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- AMD stepped 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

9

Machine Programming I: Basics

- History of x86 processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64

10

Definitions

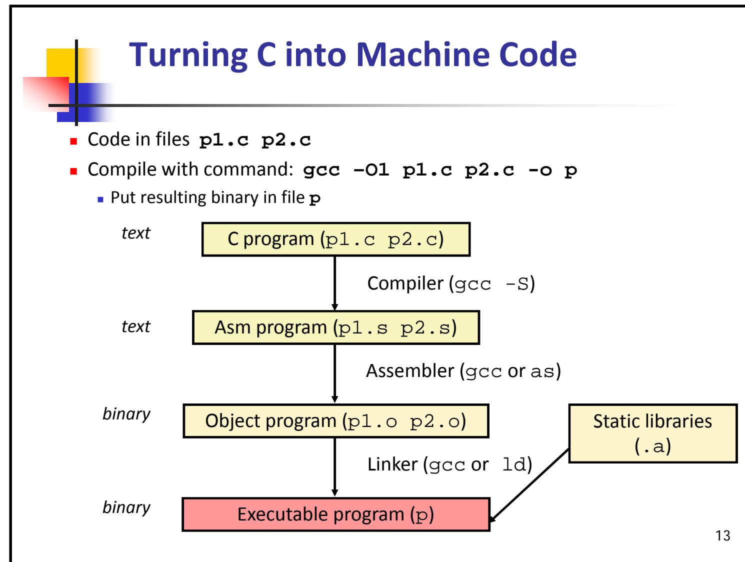
- Architecture: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  - Including instruction set specification, registers.
  - Example ISA: IA32, x86-64
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.

11

Machine Programmer's View

- 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

12



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
   popl %ebp
   ret
```

Obtain with command

```
gcc -O1 -S code.c
```

Produces file `code.s`

14

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, or 4 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

15

Assembly Characteristics: 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

16

Object Code

Code for **sum**

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

- Total of 11 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`

- 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

17

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]
```

```
0x80483ca:  03 45 08
```

- C code
  - Add two signed integers
- Assembly
  - Add 2 4-byte integers
    - "Long" words in GCC parlance
    - 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 **`0x80483ca`**

18

Disassembling Object Code

Disassembled

```
080483c4 <sum>:
 80483c4:  55         push   %ebp
 80483c5:  89 e5      mov    %esp,%ebp
 80483c7:  8b 45 0c   mov    0xc(%ebp),%eax
 80483ca:  03 45 08   add    0x8(%ebp),%eax
 80483cd:  5d         pop    %ebp
 80483ce:  c3         ret
```

- Disassembler

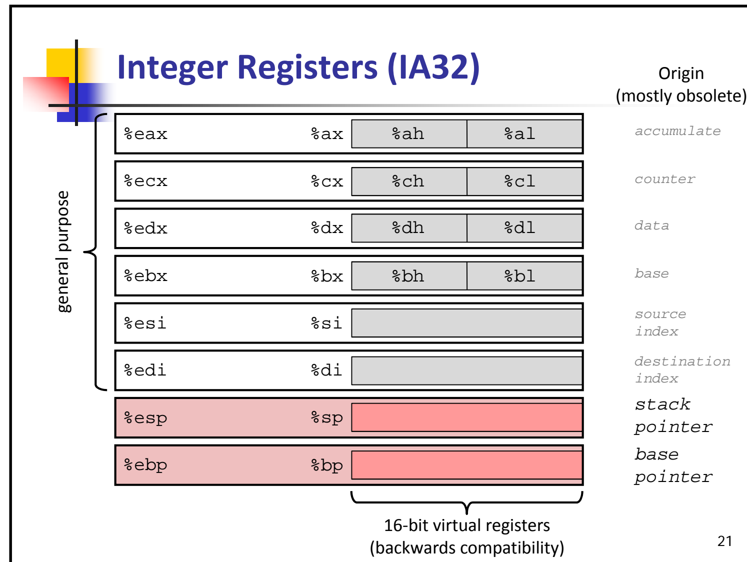
  `objdump -d p`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

19

Machine Programming I: Basics

- History of x86 processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64

20



Moving Data: IA32

- Moving Data
  `movl` ***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 8 integer registers
    - Example: **`%eax`**, **`%edx`**
    - But **`%esp`** and **`%ebp`** reserved for special use
    - Others may have special uses for particular instructions
  - ***Memory:*** 4 consecutive bytes of memory at address given by register
    - Simplest example: **`(%eax)`**
    - Various other "address modes"

%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp

22

`movl` Operand Combinations

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

Cannot do memory-memory transfer with a single instruction

23

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address

```
movl (%ecx),%eax
```

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

```
movl 8(%ebp),%edx
```

24

Memory Addressing Example

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

```
swap:
   pushl %ebp              } Set
   movl  %esp,%ebp         } Up
   pushl %ebx

   movl  8(%ebp), %edx     }
   movl  12(%ebp), %ecx    }
   movl  (%edx), %ebx      } Body
   movl  (%ecx), %eax      }
   movl  %eax, (%edx)      }
   movl  %ebx, (%ecx)      }

   popl  %ebx              }
   popl  %ebp              } Finish
   ret                     }
```

25

Understanding Swap

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

Stack (in memory)

Offset	
	• • •
12	yp
8	xp
4	Rtn adr
0	Old %ebp ← %ebp
-4	Old %ebx ← %esp

Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1

```
movl  8(%ebp), %edx    # edx = xp
movl  12(%ebp), %ecx   # ecx = yp
movl  (%edx), %ebx     # ebx = *xp (t0)
movl  (%ecx), %eax     # eax = *yp (t1)
movl  %eax, (%edx)     # *xp = t1
movl  %ebx, (%ecx)     # *yp = t0
```

26

Complete Memory Addressing Modes

- Most 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 8 integer registers
  - Ri: Index register: Any, except for **%esp**
    - Unlikely you'd use **%ebp**, either
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)
- Special Cases
  - (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]]

27

Address Computation Examples

%edx	0xf000
%ecx	0x0100

Expression	Address Computation	Address
0x8(%edx)		
(%edx,%ecx)		
(%edx,%ecx,4)		
0x80(,%edx,2)		

28

Address Computation Instruction

- `leal` *Src,Dest*
  - *Src* is address mode expression
  - Set *Dest* 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

```
int mul12(int x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leal (%eax,%eax,2), %eax  ;t <- x+x*2
sall $2, %eax             ;return t<<2
```

29

Machine Programming I: Basics

- History of x86 processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64

30

Data Representations: IA32 + x86-64

- Sizes of C Objects (in Bytes)

C Data Type	Generic 32-bit	Intel IA32	x86-64
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
char *	4	4	8
Or any other pointer			

31

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

- Extend existing registers. Add 8 new ones.
- Make `%ebp`/`%rbp` general purpose

32

New 64-bit Instructions

- Long word `l` (4 Bytes) ↔ Quad word `q` (8 Bytes)
- New instructions:
  - **movl → movq**
  - **addl → addq**
  - **sall → salq**
  - etc.

33

32-bit code for swap

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

```
swap:
   pushl %ebp                   } Set Up
   movl  %esp,%ebp
   pushl %ebx

   movl   8(%ebp), %edx         } Body
   movl   12(%ebp), %ecx
   movl   (%edx), %ebx
   movl   (%ecx), %eax
   movl   %eax, (%edx)
   movl   %ebx, (%ecx)

   popl   %ebx                  } Finish
   popl   %ebp
   ret
```

34

64-bit code for swap

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

```
swap:
                                } Set Up
   movl   (%rdi), %edx          } Body
   movl   (%rsi), %eax
   movl   %eax, (%rdi)
   movl   %edx, (%rsi)

   ret                          } Finish
```

- Operands passed in registers
  - First (**xp**) in **%rdi**, second (**yp**) in **%rsi**
- No stack operations required
- 64-bit pointers, 32-bit data

35

64-bit code for long int swap

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

```
swap_l:
                                } Set Up
   movq   (%rdi), %rdx          } Body
   movq   (%rsi), %rax
   movq   %rax, (%rdi)
   movq   %rdx, (%rsi)

   ret                          } Finish
```

- 64-bit data
  - Data held in registers **%rax** and **%rdx**
  - **movq** operation

36

Machine Programming I: Summary

- History of x86 processors and architectures
  - Evolutionary design, backward compatibility
- C, assembly, machine code
  - High-level language, machine-level code, human friendliness
- Assembly basics: registers, operands, move
  - Key is data location and semantics
- Intro to x86-64
  - More than 64-bits, more register space allows additional optimization

37

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

38