

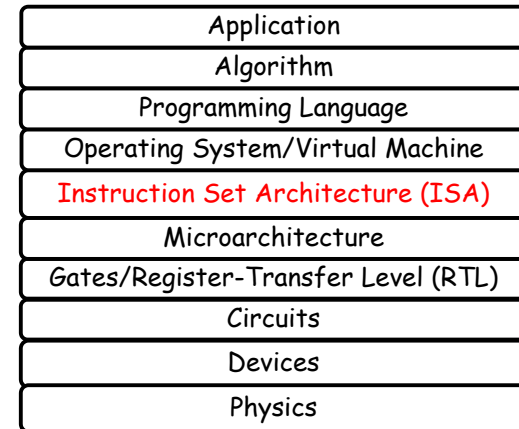
CS740

Instruction Set Architecture September 15, 2014

- Topics
 - ISA
 - x86
 - RISC & CISC

Instruction Set Architecture

- The ISA defines the functional contract between the software and the hardware



- 2 -

CS 740 F'14

Abstraction & Your Program

High-level language

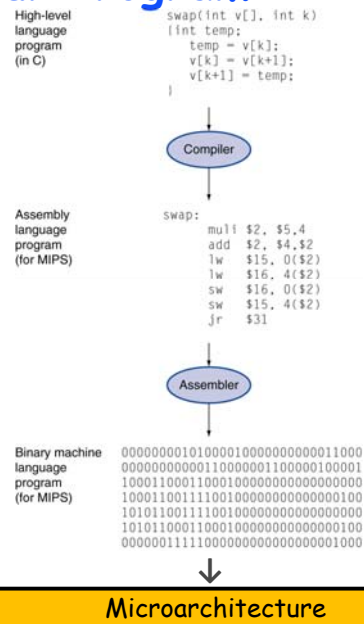
- Level of abstraction closer to problem domain
- Provides for productivity and portability

Assembly language

- Textual representation of instructions (ISA)

Hardware representation

- Binary representation of instructions (ISA)



- 3 -

Instruction Set Architecture

- The ISA defines the functional contract between the software and the hardware
- The ISA is an abstraction that hides details of the implementation from the software
- It is a functional abstraction of the processor
 - What operations can be preformed
 - How to name storage locations
 - The format (bit pattern) of the instructions
- It does NOT define
 - Timing of the operations
 - Power used by operations
 - How operations/storage are implemented

- 4 -

CS 740 F'14

ISA Goals

- Ease of Programming
- Ease of Implementation
- Good Performance
- Compatibility

Ease of Programming

- The ISA should make it easy to express programs and make it easy to create efficient programs.
- Who is creating the programs?
 - Early Days: Humans. Why?

Ease of Programming

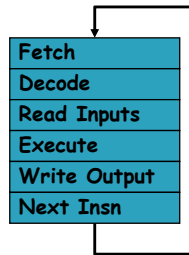
- The ISA should make it easy to express programs and make it easy to create efficient programs.
- Who is creating the programs?
 - Early Days: Humans.
 - No real compilers
 - Resources very limited
 - What does that mean for the ISA designer?
Probably want high-level operations

Ease of Programming

- The ISA should make it easy to express programs and make it easy to create efficient programs.
- Who is creating the programs?
 - Early Days: Humans.
 - Modern days (~1980 and beyond): Compilers
 - Today's optimizing compiler do a much better job than most humans could possibly do
 - Leads to change in type of instructions towards more fine-grained low-level instructions

Ease of Implementation

- ISA shouldn't get in the way of optimizing implementation
- Examples:
 - Variable length instructions
 - Varying instruction formats
 - Implied registers
 - Complex addressing modes
 - Precise interrupts
 - Appearance of atomic execution

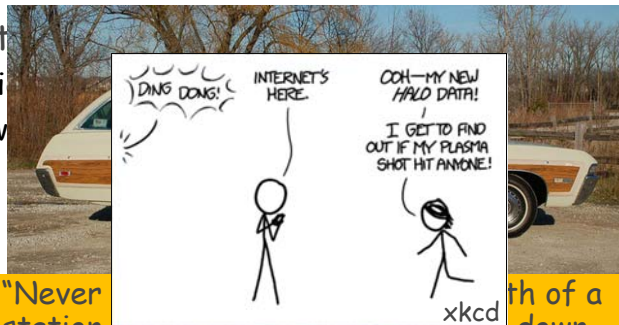


ISA & Performance

- First, let's define performance

Performance

- Response time:
 - AKA latency
 - How long does a task take?
- Throughput:
 - AKA bandwidth
 - How much work can you do per unit time?



"Never
station wagon full of tapes running down
the highway."

Tanenbaum, *Computer Networks*

Performance

- Response time:
 - AKA latency
 - How long does a task take?
- Throughput:
 - AKA bandwidth
 - How much work can you do per unit time?
- Let's examine response time
 - Elapsed time
Total time from start to finish including everything
 - CPU time
Only time spent on CPU

CPU Time

CPU Time = *CPU clock cycles* × *clock cycle time*

- CPU Clock Cycles
 - Number of clock cycles to execute program
 - Two components:
 - # of instructions &
 - cycles per instruction
- Clock Cycle Time
 - 1/Clock Frequency

$$\text{CPU Time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- 13 -

CS 740 F'14

CPU Time

$$\text{CPU Time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- Instr/program = instruction count (IC)
 - Determined by program, compiler, & **ISA**
 - This is the dynamic count of instructions executed
- Cycles/instr = cycles per instruction (CPI)
 - Determined by program, compiler, **ISA**, & μ arch
- Seconds/cycle = clock period = 1/freq
 - Determined by μ arch & technology

- 14 -

CS 740 F'14

CPI

$$\text{CPI} = \frac{\text{clock cycles}}{\text{instruction count}} = \sum_{cls=1}^n \text{CPI}_{cls} \times \frac{IC_{cls}}{IC}$$

- Different instruction classes take different numbers of cycles
- (In fact, even the same instruction can take a different number of cycles, E.g.?)
- When we say CPI, we really mean
Weighted CPI

- 15 -

CS 740 F'14

CPU Time

$$\text{CPU Time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- Improve performance by
 - Reducing instruction count
 - Reducing cycles taken by each instruction
 - Reducing clock period
- There is a tension between these

- 16 -

CS 740 F'14

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= 1 \times 2.0 \times 250\text{ps} = 1 \times 500\text{ps} \leftarrow \text{A is faster...} \\ \text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= 1 \times 1.2 \times 500\text{ps} = 1 \times 600\text{ps} \\ \frac{\text{CPU Time}_B}{\text{CPU Time}_A} &= \frac{1 \times 600\text{ps}}{1 \times 500\text{ps}} = 1.2 \leftarrow \text{...by this much}\end{aligned}$$

- 17 -

CS 740 F'14

ISA & Performance

$$\text{CPU Time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- CISC ISA:
 - Complex instructions, I.e., lots of work/instr
 - → fewer instructions/program
 - But, → more CPI & longer clock period
 - (However, modern μ arch gets around this)
- RISC ISA:
 - Simple instructions, I.e., less work/instr
 - → more instructions/program
 - But, → fewer CPI & shorter clock period
 - Heavy reliance on compiler to do the right thing

- 18 -

CS 740 F'14

Other measures of "performance"

- Performance is not just CPU time
- Or, even elapsed time
- E.g., ?

- 19 -

CS 740 F'14

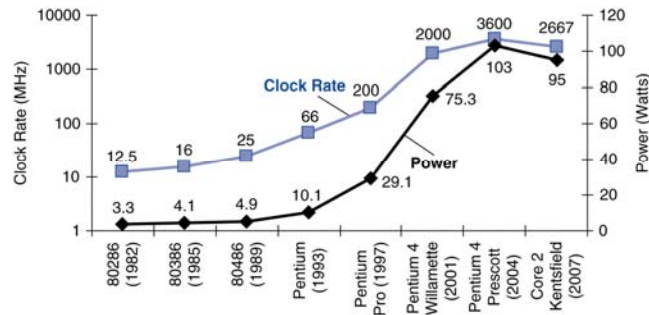
Other measures of "performance"

- Performance is not just CPU time
- Or, even elapsed time
- Power

- 20 -

CS 740 F'14

CMOS & POWER



In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30

5V → 1V

×1000

- 21 -

CS 740 F'14

Compatibility

- ISA separates interface from implementation
- Thus, many different implementations possible
 - IBM 360 first to do this and introduce 7 different machines all with same ISA
 - Intel from 4004 → core i7 → ?
 - ARM ISA
- Protects software investment
- Important to decide what should be exposed and what should be kept hidden.
 - E.g., MIPS delay slots

- 22 -

CS 740 F'14

What Goes Into an ISA?

- Operands
 - How many?
 - What kind?
 - Addressing mechanisms
- Operations
 - What kind?
 - How many?
- Format/Encoding
 - Length(s) of bit pattern
 - Which bits mean what

- 23 -

CS 740 F'14

Operands ↔ Machine Model

- Three basic types of machine
 - Stack
 - Accumulator
 - Register
- Two types of register machines
 - Register-memory
 - Most operands in most instructions can be either a register or a memory address
 - Load-store
 - Instructions are either load/store or register-based

- 24 -

CS 740 F'14

Operands Per Instruction

Depends on underlying model of machine:

- Stack

0 address add $\text{mem}[\text{sp}] \leftarrow \text{mem}[\text{sp}] + \text{mem}[\text{sp}+1]$

- Accumulator

1 address add A $\text{Acc} \leftarrow \text{Acc} + \text{mem}[A]$

- Register-Memory

2 address add R1, A $R1 \leftarrow R1 + \text{mem}[\text{EA}(A)]$

3 address add R1, R2, A $R1 \leftarrow R2 + \text{mem}[\text{EA}(A)]$

- Load-Store

3 address add R1, R2, R3 $R1 \leftarrow R2 + R3$

load R1, R2 $R1 \leftarrow \text{mem}[R2]$

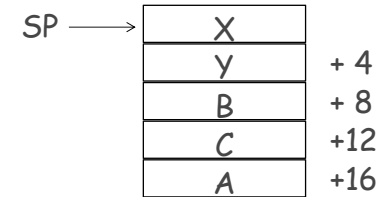
Store R1, R2 $\text{mem}[R1] \leftarrow R2$

- 25 -

CS 740 F'14

Examples

- Code for: $A = X * Y - B * C$



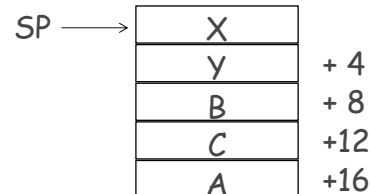
Stack
push 8(SP)
push 16(SP)
mult
push 4(sp)
push 12(sp)
mult
sub
st 20(sp)
pop

- 26 -

CS 740 F'14

Examples

- Code for: $A = X * Y - B * C$



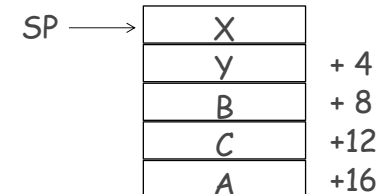
Stack	Accumulator
push 8(SP)	ld 8(SP)
push 16(SP)	mult 12(SP)
mult	st 20(SP)
push 4(sp)	ld 4(SP)
push 12(sp)	
mult	mult 0(SP)
sub	sub 20(sp)
st 20(sp)	st 16(sp)
pop	

- 27 -

CS 740 F'14

Examples

- Code for: $A = X * Y - B * C$



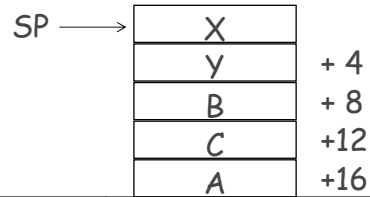
Stack	Accumulator	reg-mem
push 8(SP)	ld 8(SP)	
push 16(SP)	mult 12(SP)	
mult	st 20(SP)	mult R1, 8(SP), 12(SP)
push 4(sp)	ld 4(SP)	
push 12(sp)		
mult	mult 0(SP)	mult R2, 0(SP), 4(SP)
sub	sub 20(sp)	
st 20(sp)	st 16(sp)	sub 16(sp), R2, R1
pop		

- 28 -

CS 740 F'14

Examples

- Code for: $A = X * Y - B * C$



Accumulator	reg-mem	ld/st
ld 8(SP)		ld r1, 8(SP)
mult 12(SP)		ld r2, 12(SP)
st 20(SP)	mult R1, 8(SP), 12(SP)	ld r3, 4(SP)
ld 4(SP)		ld r4, 0(SP)
		mult r5, r1, r2
mult 0(SP)	mult R2, 0(SP), 4(SP)	mult r6, r3, r4
sub 20(sp)		sub r7, r6, r5
st 16(sp)	sub 16(sp), R2, R1	st 16(SP), r7
- 29 -		CS 740 F14

Model Trade-offs

- Stack and Accumulator:
 - Each instruction encoding is short
 - IC is high
 - Very simple exposed architecture
- Register-Memory:
 - Instruction encoding is much longer
 - More work per instruction
 - IC is low
 - Architectural state more complex
- Load/Store:
 - medium encoding length (EA longer than reg spec)
 - less work per instruction
 - IC is high
 - Architectural state more complex

CS 740 F14

Common Operand Types

- Register
 - add r1, r2, r3
 - add r1, r2
- Immediate
 - add r1, #7
- Memory
 - direct
 - add r1, [0x1000]
 - register indirect
 - add r1, (r2)
 - displacement
 - add r1, 100(r2)
 - indexed
 - add r1, (r2+r3)
 - indexed+displacement
 - add r1, 100(r2+r3)
 - scaled+displacement
 - add r1, 100(r2+r3*s)
 - memory indirect
 - add r1, ([0x1000])
 - autoincrement
 - add r1, (r2)+
 - autodecrement
 - add r1, (r2)-

CS 740 F14

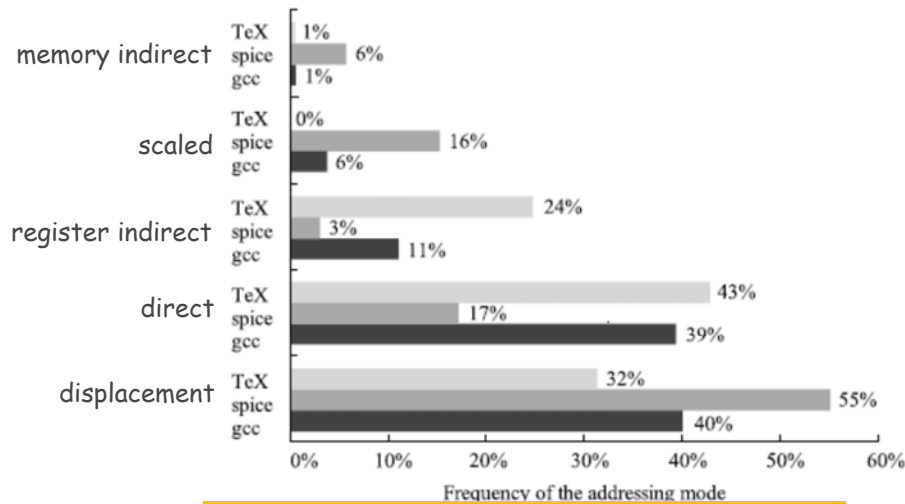
Memory Operands

- Memory addressing modes, i.e., How to specify an effective address
- How many?
- How complex?
- How much memory can be addressed?
- Trade-offs?
 - How useful is the addressing mode?
 - What is the impact on CPI? IC? Freq?
 - How many bits needed to encode in instruction?

- 32 -

CS 740 F14

Frequency of Addressing Modes



Another question: How big a displacement?

- 33 -

CS 740 F'14

How many registers?

- More registers means:
 - longer instruction encoding
 - Each register access is slower and/or
 - More power per access
 - More state is exposed (more saves/restores per func call, context switch, ...)
- Fewer registers means:
 - Harder for the compiler
 - Think of registers as cache level-0
 - small instructions
 - more instructions
- Trend towards more registers. Why?

- 34 -

CS 740 F'14

Operations

- Arithmetic
- Logical
- Data transfer
- Control flow
- OS support
- Parallelism support

- 35 -

CS 740 F'14

Control Flow

- Types:
 - Jump
 - Conditional Branch
 - Indirect Jump
 - call
 - return
 - Trap
- Destination Specified
 - Register
 - Displacement
- Condition Codes
 - set as side-effect?
 - set explicitly?

CS 740 F'14

Instruction Encoding

- Length
 - How long?
 - Fixed or Variable?
- Format
 - consistent? Specialized?
- Trade-offs:

- 37 -

CS 740 F14

Instruction Encoding

- Length
 - How long?
 - Fixed or Variable?
- Format
 - consistent? Specialized?
- Trade-offs:
 - fixed length
 - simple fetch/decode/next
 - not efficient use of instruction memory
 - Variable length
 - complex fetch/decode/next
 - improved code density

- 38 -

CS 740 F14

Intel x86 Processors

- Totally 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 Linux 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.

- 39 -

CS 740 F14

Intel x86 Evolution: Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
• 8086	1978	29K	5-10
• First 16-bit Intel processor. Basis for IBM PC & DOS			
• 1MB address space			
• 386	1985	275K	16-33
• First 32 bit Intel processor, referred to as IA32			
• Added "flat addressing", capable of running Unix			
• Pentium 4F	2004	125M	2800-3800
• First 64-bit Intel processor, referred to as x86-64			
• Core 2	2006	291M	1060-3500
• First multi-core Intel processor			
• Core i7	2008	731M	1700-3900
• Four cores (our shark machines)			

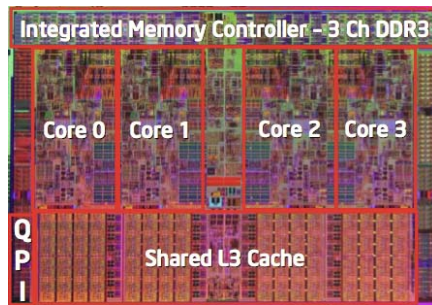
- 40 -

CS 740 F14

Intel x86 Processors, cont.

• 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



• Added Features

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

- 41 -

CS 740 F'14

x86 Clones: (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

- 42 -

CS 740 F'14

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!

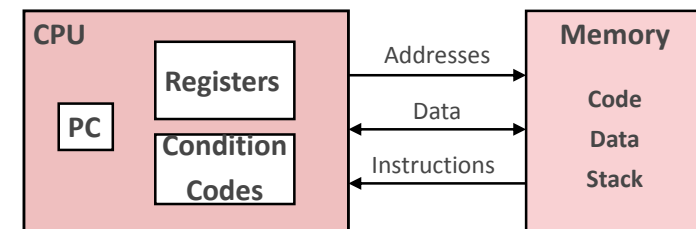
• All but low-end x86 processors support x86-64

- But, lots of code still runs in 32-bit mode

- 43 -

CS 740 F'14

Assembly 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

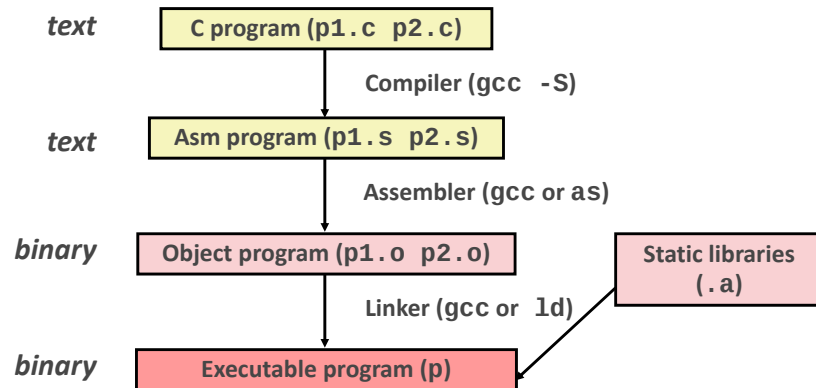
• Memory

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

CS 740 F'14

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`



- 45 -

CS 740 F'14

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

```
/usr/local/bin/gcc -O1 -S code.c
```

Produces file `code.s`

- 46 -

CS 740 F'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

- 47 -

CS 740 F'14

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

- 48 -

CS 740 F'14

Object Code

Code for sum

0x401040 <sum>:

0x55

0x89

0xe5

0x8b

0x45

0x0c

0x03

0x45

0x08

0x5d

0xc3

- 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**
- Linking occurs when program begins execution

- **Total of 11 bytes** Some libraries are *dynamically linked*
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address 0x401040**

- 49 -

CS 740 F14

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 two 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**

- 50 -

CS 740 F14

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

- 51 -

CS 740 F14

Alternate Disassembly

Object

0x401040:

0x55

0x89

0xe5

0x8b

0x45

0x0c

0x03

0x45

0x08

0x5d

0xc3

Disassembled

Dump of assembler code for function sum:

```
0x080483c4 <sum+0>:  push    %ebp
0x080483c5 <sum+1>:  mov     %esp,%ebp
0x080483c7 <sum+3>:  mov     0xc(%ebp),%eax
0x080483ca <sum+6>:  add     0x8(%ebp),%eax
0x080483cd <sum+9>:  pop     %ebp
0x080483ce <sum+10>:  ret
```

- Within gdb Debugger

gdb p

disassemble sum

- Disassemble procedure

x/11xb sum

- Examine the 11 bytes starting at sum

- 52 -

CS 740 F14

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:  file format pei-i386
```

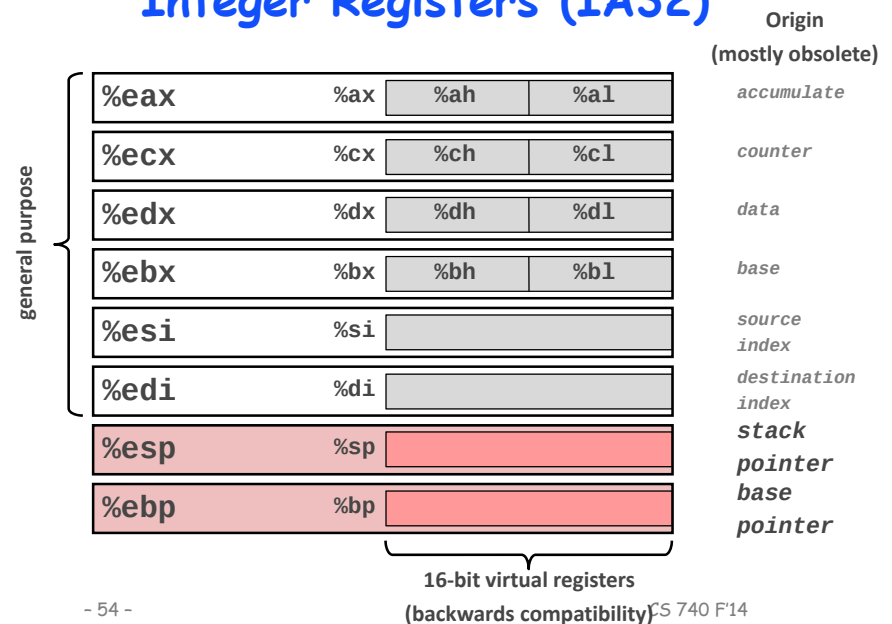
```
No symbols in "WINWORD.EXE".
Disassembly of section .text:
```

```
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

CS 740 F14

Integer Registers (IA32)



- 54 -

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 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

CS 740 F14

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 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

CS 740 F14

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

- 57 -

CS 740 F'14

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- Aha! Pointer dereferencing in C

movl (%ecx), %eax

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

- Register R specifies start of memory region
- Constant displacement D specifies offset
 - D is an arbitrary integer constrained to fit in 1-4 bytes

movl 8(%ebp), %edx

- 58 -

CS 740 F'14

Using Simple Addressing Modes

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

swap:

```
pushl %ebp
movl %esp, %ebp
pushl %ebx

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

popl %ebx
popl %ebp
ret
```

- 59 -

CS 740 F'14

Using Simple Addressing Modes

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

swap:

```
pushl %ebp
movl %esp, %ebp
pushl %ebx

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

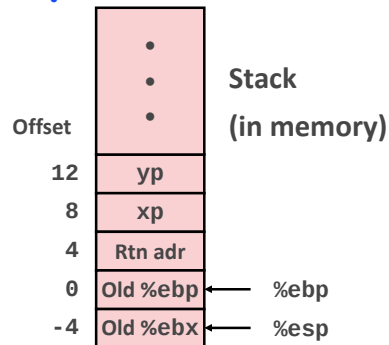
popl %ebx
popl %ebp
ret
```

- 60 -

CS 740 F'14

Understanding Swap

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



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

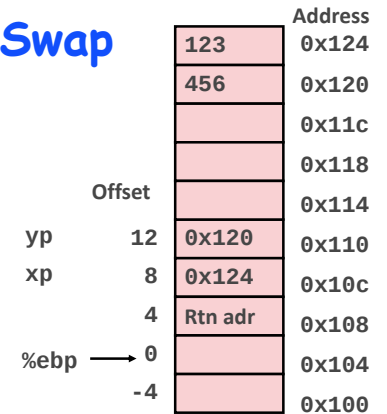
```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl (%ecx), %eax # eax = *yp (t1)
movl %eax, (%edx) # *xp = t1
movl %ebx, (%ecx) # *yp = t0
```

- 61 -

CS 740 F14

Understanding Swap

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104



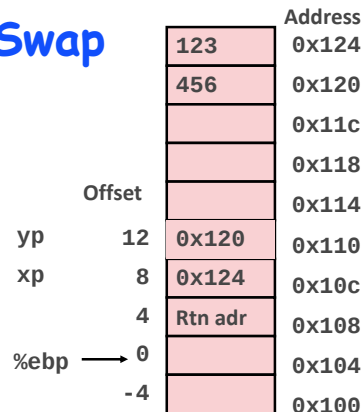
```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl (%ecx), %eax # eax = *yp (t1)
movl %eax, (%edx) # *xp = t1
movl %ebx, (%ecx) # *yp = t0
```

- 62 -

CS 740 F14

Understanding Swap

%eax	
%edx	0x124
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104



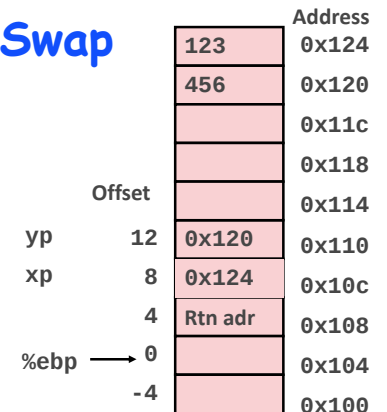
```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl (%ecx), %eax # eax = *yp (t1)
movl %eax, (%edx) # *xp = t1
movl %ebx, (%ecx) # *yp = t0
```

- 63 -

CS 740 F14

Understanding Swap

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104



```
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
```

- 64 -

CS 740 F14

Understanding Swap

%eax	
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
Offset	123	0x124
	456	0x120
yp	12	0x120
	8	0x124
xp	4	Rtn adr
	0	
%ebp	-4	

```

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

```

- 65 -

CS 740 F14

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
Offset	123	0x124
	456	0x120
yp	12	0x120
	8	0x124
xp	4	Rtn adr
	0	
%ebp	-4	

```

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

```

- 66 -

CS 740 F14

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
Offset	456	0x124
	456	0x120
yp	12	0x120
	8	0x124
xp	4	Rtn adr
	0	
%ebp	-4	

```

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

```

- 67 -

CS 740 F14

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
Offset	456	0x124
	123	0x120
yp	12	0x120
	8	0x124
xp	4	Rtn adr
	0	
%ebp	-4	

```

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

```

- 68 -

CS 740 F14

Complete Memory Addressing Modes

- Most General Form

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

- 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

$(Rb, Ri) \quad Mem[Reg[Rb] + Reg[Ri]]$

$D(Rb, Ri) \quad Mem[Reg[Rb] + Reg[Ri] + D]$

$(Rb, Ri, S) \quad Mem[Reg[Rb] + S * Reg[Ri]]$

- 69 -

CS 740 F'14

Data Representations: IA32 + x86-64

- Sizes of C Objects (in Bytes)

<i>C Data Type</i>	<i>Generic 32-bit</i>	<i>Intel IA32</i>	<i>x86-64</i>
-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

- 70 -

CS 740 F'14

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

CS 740 F'14

Instructions

- Long word l (4 Bytes) ↔ Quad word q (8 Bytes)
- New instructions:
 - movl → movq
 - addl → addq
 - sall → salq
 - etc.
- 32-bit instructions that generate 32-bit results
 - Set higher order bits of destination register to 0
 - Example: addl

- 72 -

CS 740 F'14

32-bit code for swap

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

```
swap:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx

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

    popl %ebx
    popl %ebp
    ret
```

Set Up

Body

Finish

- 73 -

CS 740 F'14

64-bit code for swap

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

```
swap:
    movl (%rdi), %edx
    movl (%rsi), %eax
    movl %eax, (%rdi)
    movl %edx, (%rsi)
```

Set Up

Body

Finish

- Operands passed in registers (why useful?)
 - First (**xp**) in **%rdi**, second (**yp**) in **%rsi**
 - 64-bit pointers
- No stack operations required
- 32-bit data
 - Data held in registers **%eax** and **%edx**
 - **movl** operation

- 74 -

CS 740 F'14

64-bit code for long int swap

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

```
swap_l:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)

    ret
```

Set Up

Body

Finish

- 64-bit data
 - Data held in registers **%rax** and **%rdx**
 - **movq** operation
 - "q" stands for quad-word

- 75 -

CS 740 F'14

CISC v. RISC

- RISC: Reduced Instruction Set Computer
 - Introduced Early 80's
 - RISC-I (berkeley), MIPS (stanford), IBM 801
 - Today: ARM
- CISC: Complex Instruction Set Computer
 - What everything was before RISC
 - Vax, x86, 68000
 - Today: x86
- Outcome:
 - RISC in academy (and in technology)
 - CISC in commercial space, but ...
 - RISC in Embedded (and under the covers)

- 76 -

CS 740 F'14

Basic Comparison

• CISC

- variable length instructions: 1-321 bytes
- GP registers+special purpose registers+PC+SP+conditions
- Data: bytes to strings
- memory-memory instructions
- special instructions: e.g., crc, polyf, ...

• RISC

- fixed length instructions: 4 bytes
- GP registers + PC
- load/store with few addressing modes

- 77 -

CS 740 F'14

ADD—Add

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
04 <i>ib</i>	ADD AL, <i>imm8</i>	I	Valid	Valid	Add <i>imm8</i> to AL.
05 <i>iw</i>	ADD AX, <i>imm16</i>	I	Valid	Valid	Add <i>imm16</i> to AX.
05 <i>id</i>	ADD EAX, <i>imm32</i>	I	Valid	Valid	Add <i>imm32</i> to EAX.
REX.W + 05 <i>id</i>	ADD RAX, <i>imm32</i>	I	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to RAX.
80 /0 <i>ib</i>	ADD r/m8, <i>imm8</i>	MI	Valid	Valid	Add <i>imm8</i> to r/m8.
REX.W + 80 /0 <i>ib</i>	ADD r/m8*, <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to r/m64.
81 /0 <i>iw</i>	ADD r/m16, <i>imm16</i>	MI	Valid	Valid	Add <i>imm16</i> to r/m16.
81 /0 <i>id</i>	ADD r/m32, <i>imm32</i>	MI	Valid	Valid	Add <i>imm32</i> to r/m32.
REX.W + 81 /0 <i>id</i>	ADD r/m64, <i>imm32</i>	MI	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to r/m64.
83 /0 <i>ib</i>	ADD r/m16, <i>imm8</i>	MI	Valid	Valid	Add sign-extended <i>imm8</i> to r/m16.
83 /0 <i>ib</i>	ADD r/m32, <i>imm8</i>	MI	Valid	Valid	Add sign-extended <i>imm8</i> to r/m32.
REX.W + 83 /0 <i>ib</i>	ADD r/m64, <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to r/m64.
00 /r	ADD r/m8, r8	MR	Valid	Valid	Add r8 to r/m8.
REX.W + 00 /r	ADD r/m8*, r8*	MR	Valid	N.E.	Add r8 to r/m8.
01 /r	ADD r/m16, r16	MR	Valid	Valid	Add r16 to r/m16.
01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.
02 /r	ADD r8, r/m8	RM	Valid	Valid	Add r/m8 to r8.
REX.W + 02 /r	ADD r8*, r/m8*	RM	Valid	N.E.	Add r/m8 to r8.
03 /r	ADD r16, r/m16	RM	Valid	Valid	Add r/m16 to r16.
03 /r	ADD r32, r/m32	RM	Valid	Valid	Add r/m32 to r32.
REX.W + 03 /r	ADD r64, r/m64	RM	Valid	N.E.	Add r/m64 to r64.

NOTES:

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Technology Trends

• Pre-1980

- lots of hand written assembly
- Compiler technology in its infancy
- multi-chip implementations
- memory speed and CPU speed

• Early 80's

- VLSI makes single chip possible (But only if very simple)

- Compiler technology improving

• Late 90's

- CPU speed vastly faster than memory speed
- More transistors makes μ ops possible

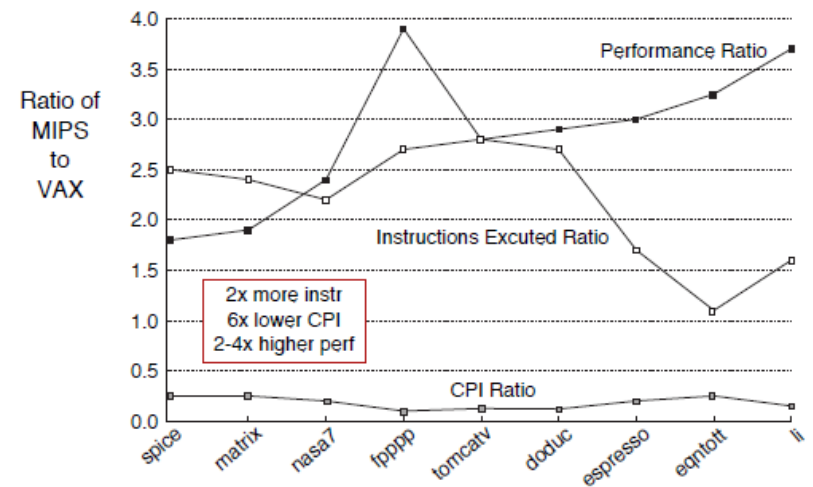
RISC Goals:

- enable single-chip CPU
- Rely on compiler
- Aim for high frequency & low CPI

- 79 -

CS 740 F'14

MIPS v. VAX



-- H&P, Appendix J, from Bhandarkar and Clark, 1991

- 80 -

CS 740 F'14

The RISC Design Tenets

- **Single-cycle execution**
 - CISC: many multicycle operations
- **Hardwired (simple) control**
 - CISC: **microcode** for multi-cycle operations
- **Load/store architecture**
 - CISC: register-memory and memory-memory
- **Few memory addressing modes**
 - CISC: many modes
- **Fixed-length instruction format**
 - CISC: many formats and lengths
- **Reliance on compiler optimizations**
 - CISC: hand assemble to get good performance
- **Many registers** (compilers can use them effectively)
 - CISC: few registers

CS 740 F'14

RISC vs CISC Performance Argument

$$\text{CPU Time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- **CISC** (Complex Instruction Set Computing)
 - Reduce "instructions/program" with "complex" instructions
 - But tends to increase "cycles/instruction" or clock period
 - Easy for assembly-level programmers, good code density
- **RISC** (Reduced Instruction Set Computing)
 - Improve "cycles/instruction" with many 1-cycle instructions
 - Increases "instruction/program", but hopefully not as much
 - Help from smart compiler
 - Perhaps improve clock cycle time (seconds/cycle)
 - via aggressive implementation allowed by simpler insn

- 82 -

CS 740 F'14

The Debate

- RISC argument
 - CISC is fundamentally handicapped
 - For a given technology, RISC implementation will be faster
 - Current technology enables single-chip RISC
 - When it enables single-chip CISC, RISC will be pipelined
 - When it enables pipelined CISC, RISC will have caches
 - When it enables CISC with caches, RISC will have ...
- CISC rebuttal
 - CISC flaws not fundamental, can be fixed with **more Ts**
 - Moore's Law will narrow the RISC/CISC gap (true)
 - Good pipeline: RISC = 100K transistors, CISC = 300K
 - By 1995: 2M+ transistors had evened playing field
 - Software costs dominate, **compatibility** is paramount

- 83 -

CS 740 F'14

Intel's x86 Trick: RISC Inside

- 1993: Intel wanted "out-of-order execution" in Pentium Pro
 - Hard to do with a coarse grain ISA like x86
- Solution? Translate x86 to RISC micro-ops (**μops**) in hardw
 - push \$eax → store \$eax, -4(\$esp)
 - addi \$esp, \$esp, -4
- + Processor maintains **x86 ISA externally for compatibility**
- + But executes **RISC μISA internally for implementability**
- Given translator, x86 almost as easy to implement as RISC
 - Intel implemented "out-of-order" before any RISC company
 - "OoO" also helps x86 more (because ISA limits compiler)
- Also used by other x86 implementations (AMD)
- Different **μops** for different designs
 - Not part of the ISA specification

- 84 -

84 740 F'14

Potential Micro-op Scheme

- Most instructions are a **single** micro-op
 - Add, xor, compare, branch, etc.
 - Loads example: `mov -4(%rax), %ebx`
 - Stores example: `mov %ebx, -4(%rax)`
- Each memory access adds a micro-op
 - "`addl -4(%rax), %ebx`" is two micro-ops (load, add)
 - "`addl %ebx, -4(%rax)`" is three micro-ops (load, add, store)
- Function call (CALL) - 4 uops
 - Get program counter, store program counter to stack, adjust stack pointer, unconditional jump to function start
- Return from function (RET) - 3 uops
 - Adjust stack pointer, load return address from stack, jump register
- Again, just a basic idea, micro-ops are specific to each chip

- 85 -

85 740 F14

More About Micro-ops

- Two forms of μ ops "cracking"
 - Hard-coded logic: fast, but complex (for insn in few μ ops)
 - Table: slow, but "off to the side", doesn't complicate rest of machine
 - Handles the really complicated instructions
- x86 code is becoming more "RISC-like"
 - In 32-bit to 64-bit transition, x86 made two key changes:
 - 2x number of registers, better function conventions
 - More registers, fewer pushes/pops
 - Result? Fewer complicated instructions
 - Smaller number of μ ops per x86 insn

- 86 -

CS 740 F14

Winner for Desktop PCs: CISC

- x86 was first mainstream 16-bit microprocessor by ~2 years
 - IBM put it into its PCs...
 - Rest is historical inertia, Moore's law, and "financial feedback"
 - x86 is most difficult ISA to implement and do it fast but...
 - Because Intel sells the most **non-embedded** processors...
 - It hires more and better engineers...
 - Which help it maintain competitive performance ...
 - **And given competitive performance, compatibility wins...**
 - So Intel sells the most **non-embedded** processors...
 - AMD as a competitor keeps pressure on x86 performance
- Moore's Law has helped Intel in a big way
 - Most engineering problems can be solved with more transistors

- 87 -

CS 740 F14

Winner for Embedded: RISC

- ARM (Acorn RISC Machine → Advanced RISC Machine)
 - First ARM chip in mid-1980s (from Acorn Computer Ltd).
 - 3 billion units sold in 2009 (>60% of all 32/64-bit CPUs)
 - Low-power and **embedded** devices (phones, for example)
 - Significance of embedded? ISA Compatibility less powerful force
- 32-bit RISC ISA
 - 16 registers, PC is one of them
 - Rich addressing modes, e.g., auto increment
 - Condition codes, each instruction can be conditional
- ARM does not sell chips; it licenses its ISA & core designs
- ARM chips from many vendors
 - Qualcomm, Freescale (was Motorola), Texas Instruments, STMicroelectronics, Samsung, Sharp, Philips, etc.

- 88 -

CS 740 F14

Redux: Are ISAs Important?

- Does "quality" of ISA actually matter?
 - Not for performance (mostly)
 - Mostly comes as a design complexity issue
 - Insn/program: everything is compiled, compilers are good
 - Cycles/insn and seconds/cycle: μ ISA, many other tricks
 - What about power efficiency? Maybe
 - ARMs are most power efficient today...
 - » ...but Intel is moving x86 that way (e.g, Intel's Atom)
 - **Open question: can x86 be as power efficient as ARM?**
- Does "nastiness" of ISA matter?
 - Mostly no, only compiler writers and hardware designers see it
- Even compatibility is not what it used to be
 - Software emulation, cloud services
 - **Open question: will "ARM compatibility" be the next x86?**

Request

- PDF with
 - picture
 - name
 - andrew id
 - major
 - area
- Email to seth@cmu.edu
- subject "15740 picture"

- Also: paper reviews should have subject: "15740 review for m/d"