

Instruction Set Architecture

15-740 FALL'18

NATHAN BECKMANN

Topics

Instruction set architecture (ISA) design tradeoffs

X86

RISC vs CISC

Assignment 1 released

Due: 19 September 2018 (12 days)

9 problems

Time **not** equally divided!

Problem 9 uses PIN, which you must learn ($\approx 45\%$ of assignment)

- Read online PIN tutorial
- Don't put this off
- PIN available at [.../15740-f18/public/](#)

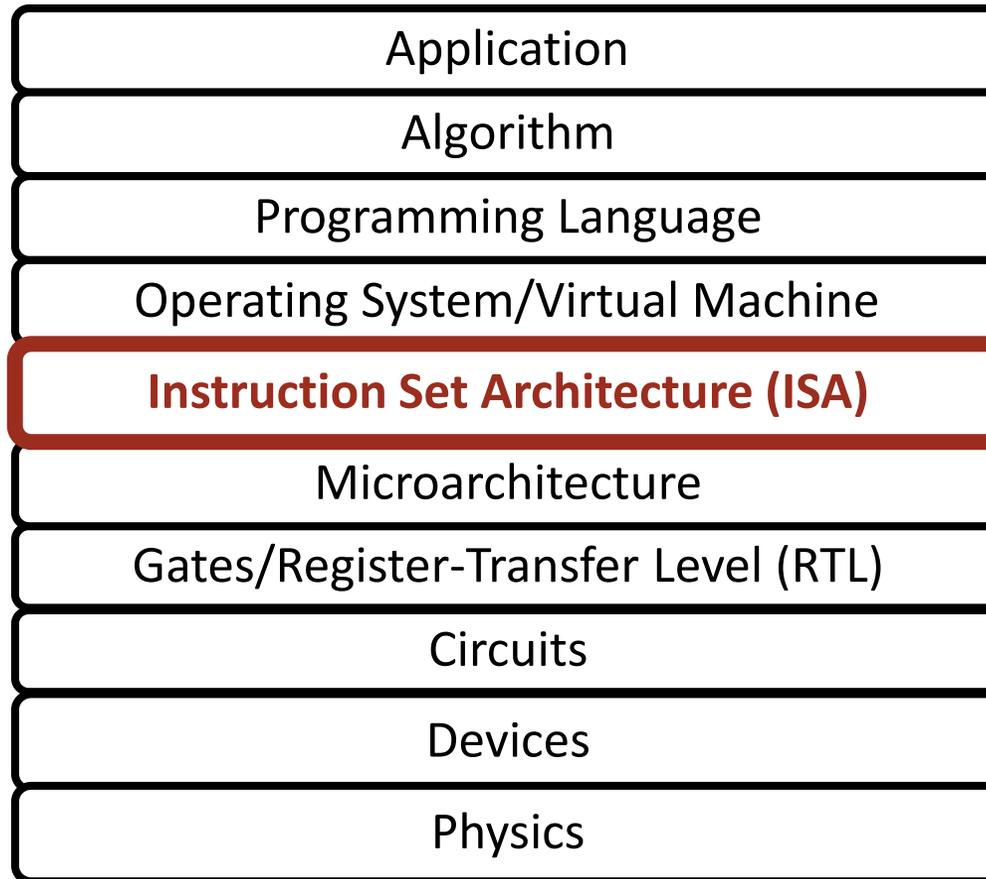
Readings

Everyone should have access to Gradescope now

Turn in reading review before **following** lecture

Questions/comments/concerns?

Abstraction layers in modern systems



← ISA defines **functional contract** between hardware and software

I.e., what the hardware does and (doesn't) guarantee will happen

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)

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Compiler

Assembly language program (for MIPS)

```
swap:
  muli $2, $5, 4
  add  $2, $4, $2
  lw   $15, 0($2)
  lw   $16, 4($2)
  sw   $16, 0($2)
  sw   $15, 4($2)
  jr   $31
```

Assembler

Binary machine language program (for MIPS)

```
000000001010000100000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
100011001111001000000000000000100
101011001111001000000000000000000
101011000110001000000000000000100
00000011111000000000000000001000
```

Microarchitecture

Instruction set architecture (ISA)

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

→ The ISA is functional abstraction of the processor (a “mental model”)

- What operations can be performed
- How to name storage locations
- The format (bit pattern) of the instructions

ISA typically does **NOT** define

- Timing of the operations
- Power used by operations
- How operations/storage are implemented

*If timing leaks information,
is it really Intel's fault?*

ISA design goals

Ease of programming / Code generation

(software perspective)

Ease of implementation

(hardware perspective)

Good performance

Compatibility

Completeness (eg, Turing)

Compactness – reduce program size

Scalability / extensibility

Features: Support for OS / parallelism / ...

Etc

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
 - Q: What does that mean for the ISA designer?
 - A: High-level operations
- Modern days (~1980 and beyond): Compilers
 - Today's optimizing compiler do a much better job than most humans
 - Q: What does that mean for the ISA designer?
 - A: Fine-grained, low-level instructions

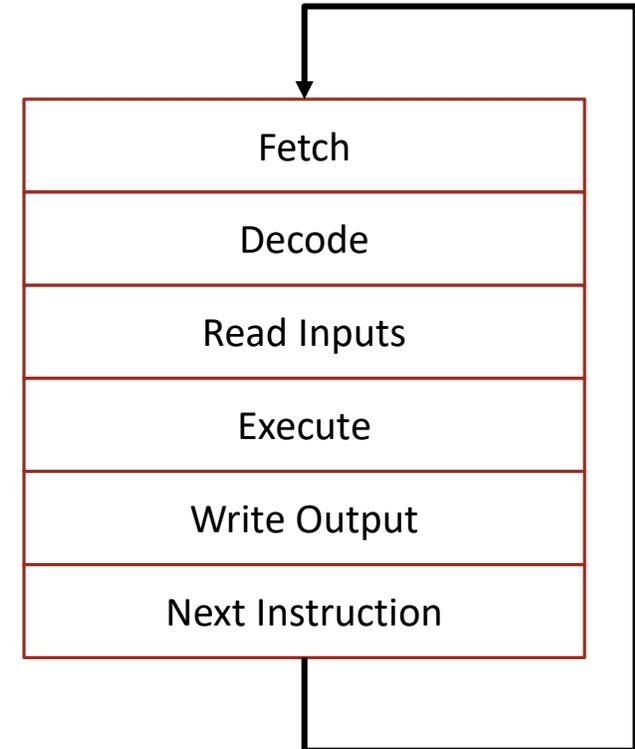
Ease of implementation

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

Simple processor pipeline:



What is performance?

Throughput – how many programs can I execute per unit time?

Elapsed time – how long does my program take to execute?

- Total time from start to finish including everything (e.g., I/O)

CPU time – only time spent in processor ← **Architects mostly focus on this**

Response time (latency)

Elapsed time

- Total time from start to finish including everything

CPU time (only time spent in processor)

$$\text{CPU Time} = \text{CPU clock cycles} \times \text{CPU clock time}$$

CPU clock cycles = number of cycles needed to execute program

- # of instructions executed
- Cycles per instruction

CPU clock time = $1 / \text{frequency}$

“The Iron Law of Performance”

$$CPU\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

What determines each factor? How does ISA impact each?

Instructions / program = **dynamic** instruction count (not code size)

- Determined by program, compiler, and ISA

Cycles / instruction (CPI)

- Determined by ISA, μ arch, program, and compiler

Seconds / cycle (critical path)

- Determined by μ arch and technology

Cycles per instruction (CPI)

Different instruction classes take different numbers of cycles

In fact, even the same instruction can take a different number of cycles

- Example?

When we say CPI, we really mean:

Weighted CPI

$$CPI = \frac{\textit{Clock cycles}}{\textit{Instruction count}} = \sum_{i=1}^n CPI_i \times \frac{\textit{Instruction count}_i}{\textit{Instruction count}}$$

CPU time

$$CPU\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

How to improve performance:

Reduce instruction count

Reduce cycles per instruction

Reduce clock time

But there is a tension between these...

CPI example – which machine is faster?

Computer A: Cycle time = 250ps, CPI = 2.0

Computer B: Cycle time = 500ps, CPI = 1.2

Same ISA

$$\begin{aligned} \text{CPU time}_A &= \text{Instruction count} \times \text{CPI}_A \times \text{Cycle time}_A \\ &= \text{Instruction count} \times 2 \times 250\text{ps} \\ &= \text{Instruction count} \times 500\text{ps} \end{aligned}$$

A is faster...

$$\begin{aligned} \text{CPU time}_B &= \text{Instruction count} \times \text{CPI}_B \times \text{Cycle time}_B \\ &= \text{Instruction count} \times 1.2 \times 500\text{ps} \\ &= \text{Instruction count} \times 600\text{ps} \end{aligned}$$

$$\frac{\text{CPU time}_B}{\text{CPU time}_A} = \frac{600\text{ps}}{500\text{ps}} = 1.2$$

...by this much

Other measures of “performance”

Performance is not just CPU time

Or, even elapsed time

E.g., ?

Other measures of “performance”

Performance is not just CPU time

Or, even elapsed time

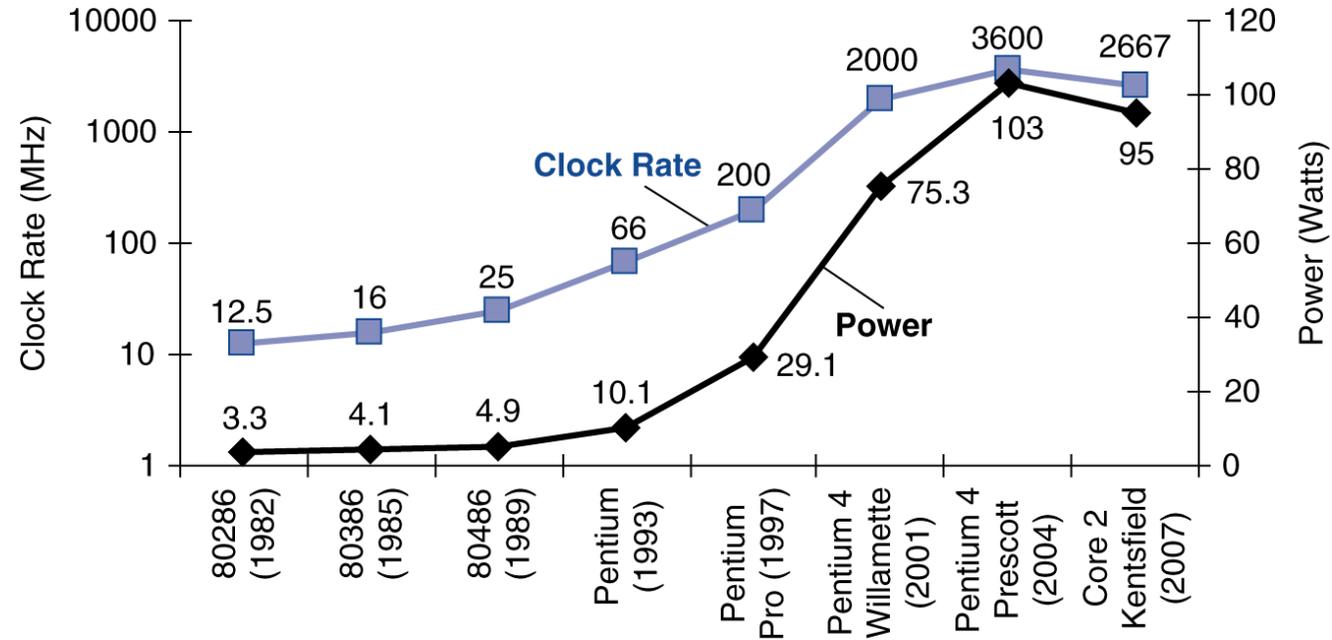
Power

Area (in mm^2 of Si, a.k.a., # transistors)

Complexity

Compatibility

CMOS & power



In CMOS IC technology:

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

×30

5V → 1V

×1000

Compatibility

“Between 1970 and 1985 many thought the primary job of the computer architect was the design of instruction sets. ...The educated architect was expected to have strong opinions about the strengths and especially the weaknesses of the popular computers. **The importance of binary compatibility in quashing innovation** in instruction set design was unappreciated by many researchers and textbook writers, giving the impression that many architects would get a chance to design an instruction set.” - H&P, Appendix A

Compatability

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 8086 → core i7 → Xeon Phi → ?
- ARM ISA mobile → server

Protects software investment

Important to decide what should be exposed and what should be kept hidden.

- E.g., MIPS “branch delay slots”

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

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

Operands per instruction

Stack

0 address	add	$\text{push}(\text{pop}()) + \text{pop}()$
-----------	-----	--

Accumulator

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

Register-Memory

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

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

Load-Store

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

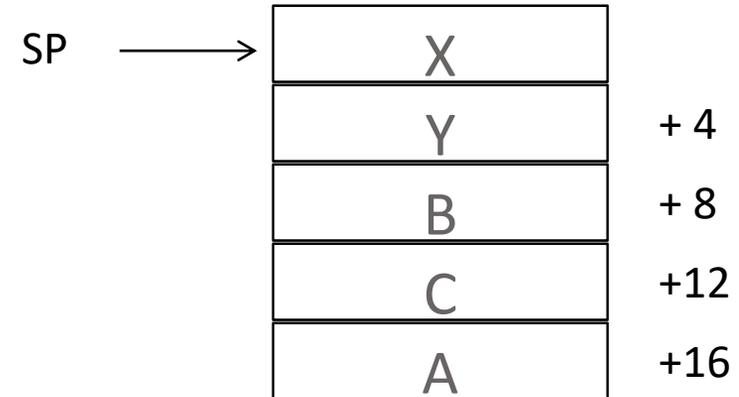
2 address	load R1, R2	$R1 \leftarrow \text{mem}[R2]$
-----------	-------------	--------------------------------

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

Examples

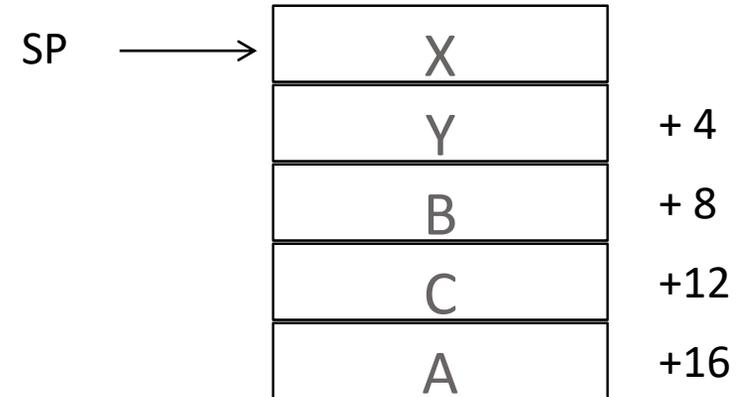
Code for: $A = X * Y - B * C$

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



Examples

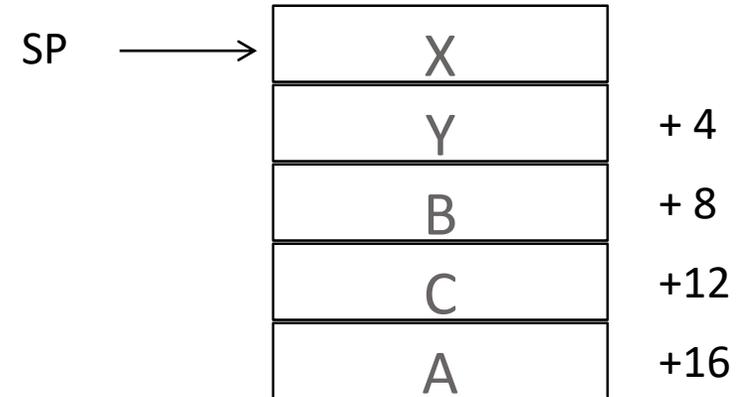
Code for: $A = X * Y - B * C$



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			

Examples

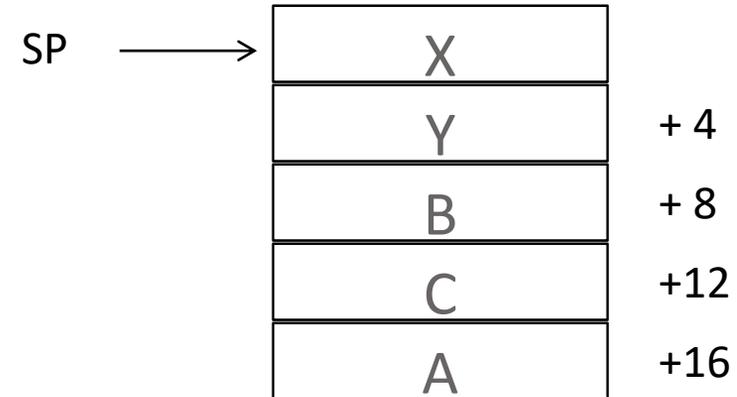
Code for: $A = X * Y - B * C$



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 R2, 0 (SP), 4 (SP)
mult	mult 0 (SP)	
sub	sub 20 (sp)	sub 16 (sp), R2, R1
st 20 (sp)	st 16 (sp)	
pop		

Examples

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



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

Machine model tradeoffs

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
- Less work per instruction
- IC is high
- Architectural state more complex

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)-

Memory operands

Memory addressing modes, i.e.,
How to specify an effective address

How many?

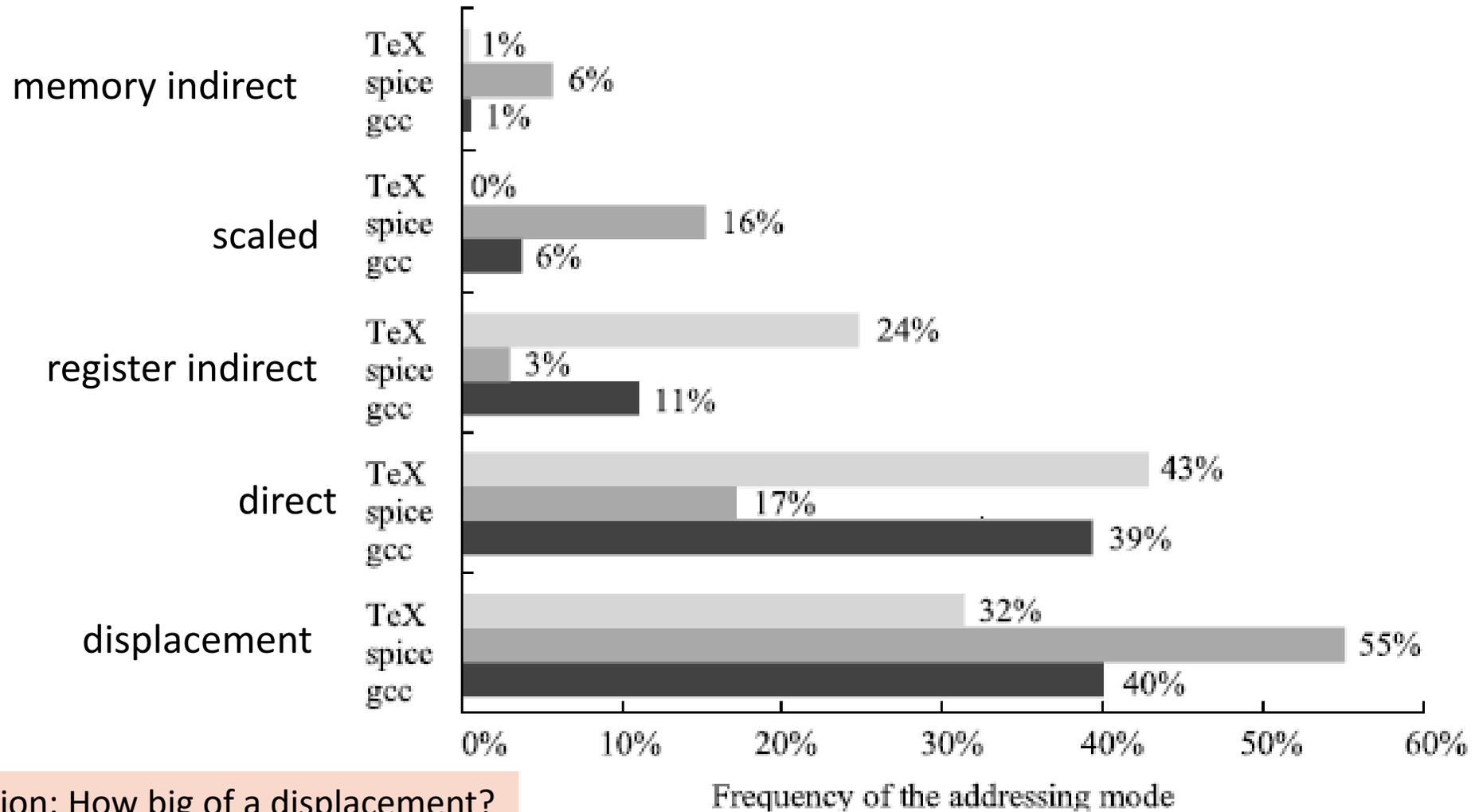
How complex?

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?

Frequency of addressing modes



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?

Operations

Arithmetic

Logical

Data transfer

Control flow

OS support

Parallelism support

Control flow

Types:

- Jump
- Conditional Branch
- Indirect Jump
 - call
 - return
- Trap

Destination Specified

- Register
- Displacement

Condition Codes

- Set as side-effect?
- Set explicitly?

Instruction encoding

Length

- How long?
- Fixed or Variable?

Format

- consistent? Specialized?

Trade-offs:

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

x86 Overview

Intel x86 Processors

Totally dominate laptop/desktop/server market

- ARM trying to gain a foothold

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 ...not so much for power

Intel x86 Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
8086	1978	29K	5-10
<ul style="list-style-type: none">◦ First 16-bit Intel processor. Basis for IBM PC & DOS◦ 1MB address space			
386	1985	275K	16-33
<ul style="list-style-type: none">◦ First 32 bit Intel processor , referred to as IA32◦ Added “flat addressing”, capable of running Unix			
Pentium 4F	2004	125M	2800-3800
<ul style="list-style-type: none">◦ First 64-bit Intel processor, referred to as x86-64			
Core 2	2006	291M	1060-3500
<ul style="list-style-type: none">◦ First multi-core Intel processor			
Kaby Lake	2016	~1.7B	2700-3500
<ul style="list-style-type: none">◦ Recent “Core i7” branded processor			

x86 Clones (AMD)

Historically

- AMD has followed just behind Intel
- A little bit slower, a lot cheaper

The Best of Times...

- 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

x86 Clones (AMD)

The worst of times...

“Bulldozer”: re-designed from scratch (2011)

- Focus on threading
- Poor single-thread performance (low IPC)
- Built for parallel software that didn't arrive!

Intel dominates performance for years...

“Zen”: re-re-design (2017)

- Re-focused on single-thread IPC
- AMD is back!

Many proclaimed the death of core microarchitecture, but parallelism is hard.

Intel's 64-Bit

Intel Attempted Radical Shift from IA32 to IA64

- Totally different architecture (Itanium)
- Executes IA32 code only as legacy
- Relied on compiler → disappointing performance (will talk more about this later)

AMD Stepped in with Evolutionary Solution

- x86-64 (now called “AMD64”)

Intel felt obligated to focus on IA64

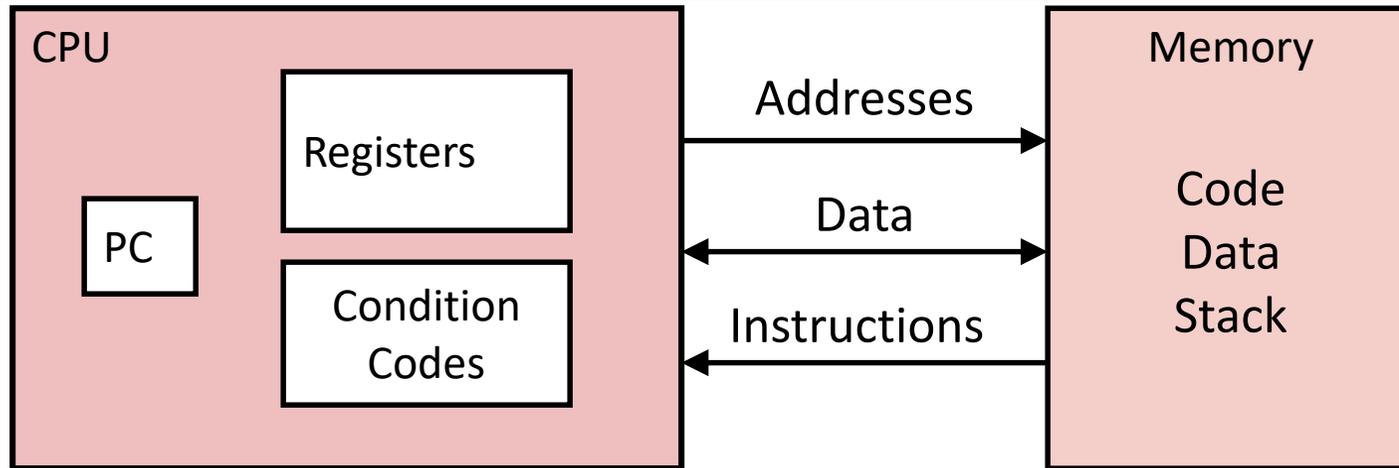
- Hard to admit mistake or that AMD is better

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

Assembly programmer's view



Programmer-Visible State

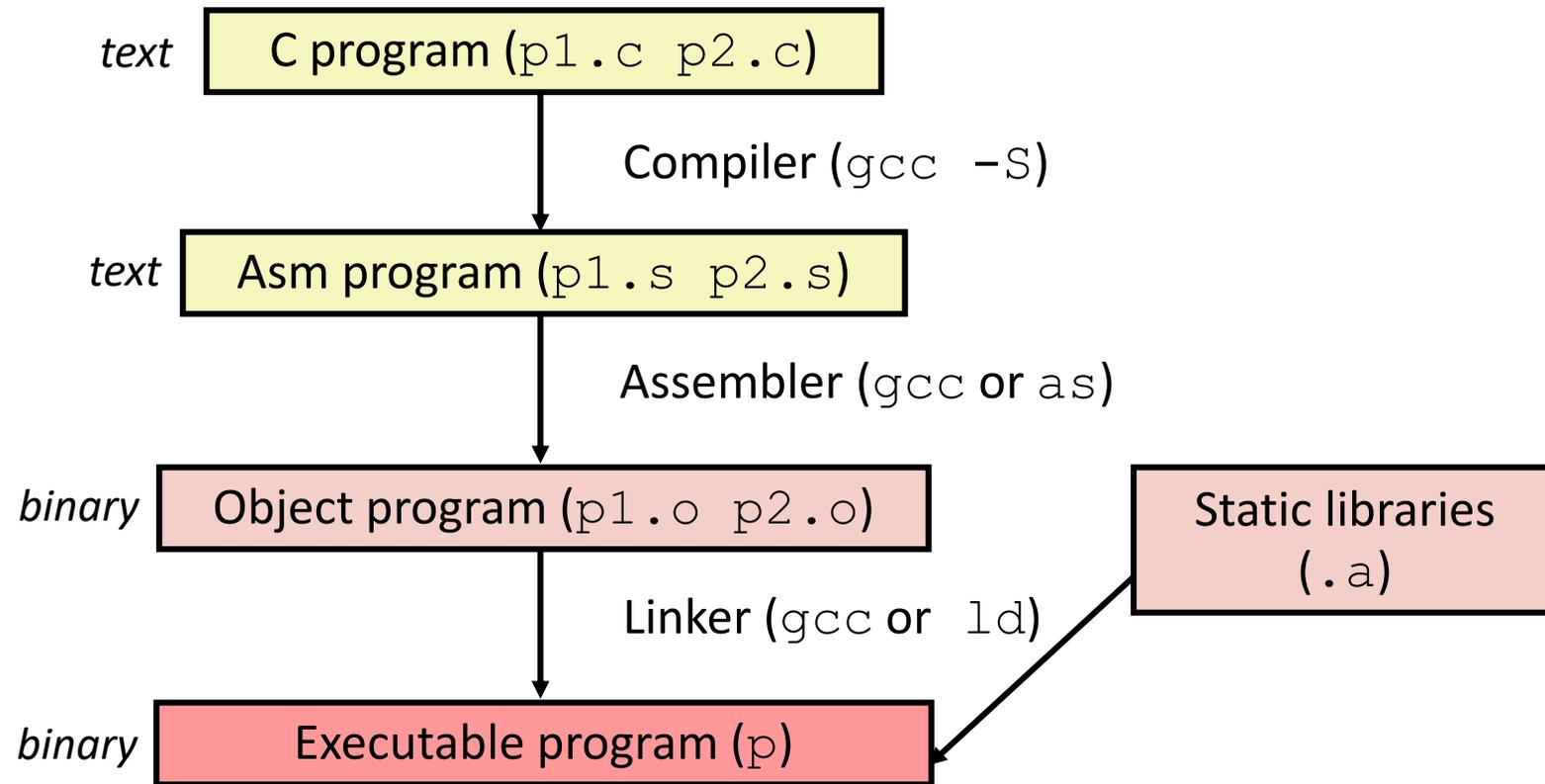
- **PC: Program counter**
 - Address of next instruction
 - Called “EIP” (IA32) or “RIP” (x86-64)
- **Register file**
 - Heavily used program data
- **Condition codes**
 - Store status information about most recent arithmetic operation
 - Used for conditional branching
- **Memory**
 - Byte addressable array
 - Code and user data
 - Stack to support procedures

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`



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

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 bytes

No aggregate types such as arrays or structures

- Just contiguously allocated bytes in memory

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

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

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**

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

Alternative 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

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

Integer registers (IA32)

Origin
(mostly obsolete)

general purpose	%eax	%ax	%ah	%al	<i>accumulate</i>
	%ecx	%cx	%ch	%cl	<i>counter</i>
	%edx	%dx	%dh	%dl	<i>data</i>
	%ebx	%bx	%bh	%bl	<i>base</i>
	%esi	%si			<i>source index</i>
	%edi	%di			<i>destination index</i>
	%esp	%sp			<i>stack pointer</i>
	%ebp	%bp			<i>base pointer</i>

16-bit virtual registers
(backwards compatibility)

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 addressing modes

<code>%eax</code>
<code>%ecx</code>
<code>%edx</code>
<code>%ebx</code>
<code>%esi</code>
<code>%edi</code>
<code>%esp</code>
<code>%ebp</code>

Moving Data: IA32

Moving Data

`mov` **1** *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 addressing modes

<code>%eax</code>
<code>%ecx</code>
<code>%edx</code>
<code>%ebx</code>
<code>%esi</code>
<code>%edi</code>
<code>%esp</code>
<code>%ebp</code>

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

Simple memory addressing modes

Indirect

(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
```


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
```

Setup

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

Body

```
popl  %ebx
popl  %ebp
ret
```

Finish

Using simple addressing modes

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

Machine specifies calling convention (a.k.a. application binary interface or ABI):

- Args passed on stack
- Result in EAX
- Everything except EAX, ECX, and EDX are callee-saved

swap:

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

Setup

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

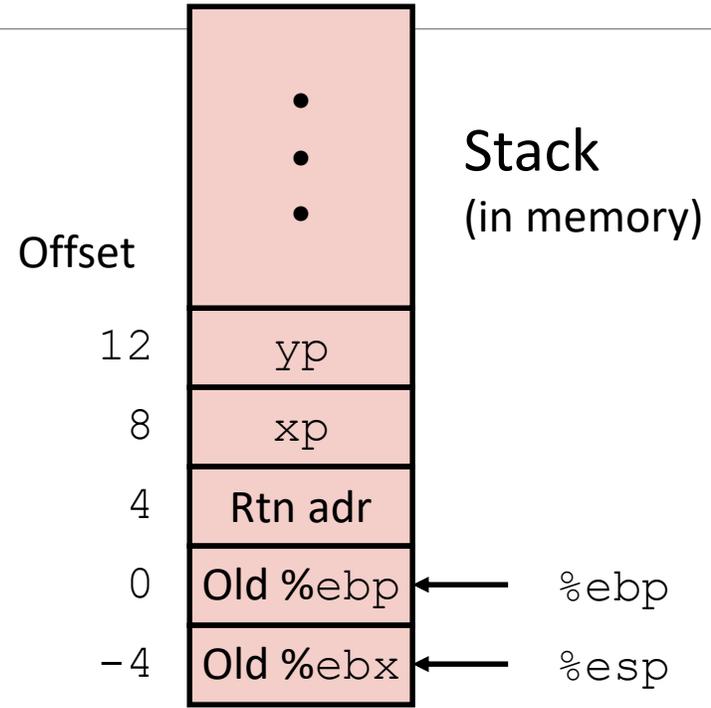
Body

```
popl  %ebx
popl  %ebp
ret
```

Finish

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
```

Understanding Swap

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

		Offset	Address
			123
			456
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

```

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
    
```

Understanding Swap

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

		Offset	Address
			123
			456
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

```

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
    
```

Understanding Swap

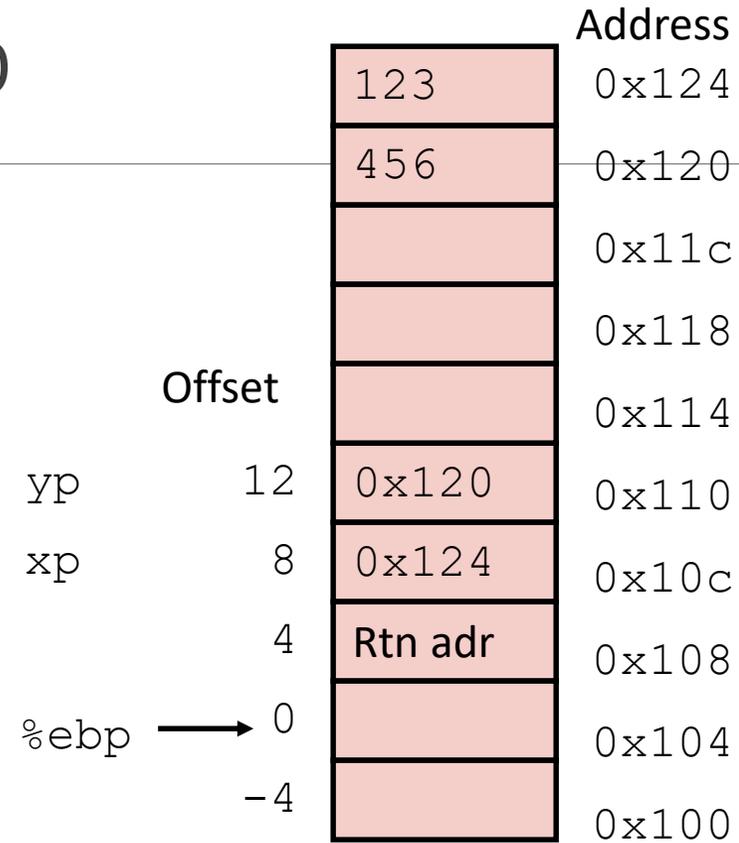
%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address	
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

```
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
```

Understanding Swap

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

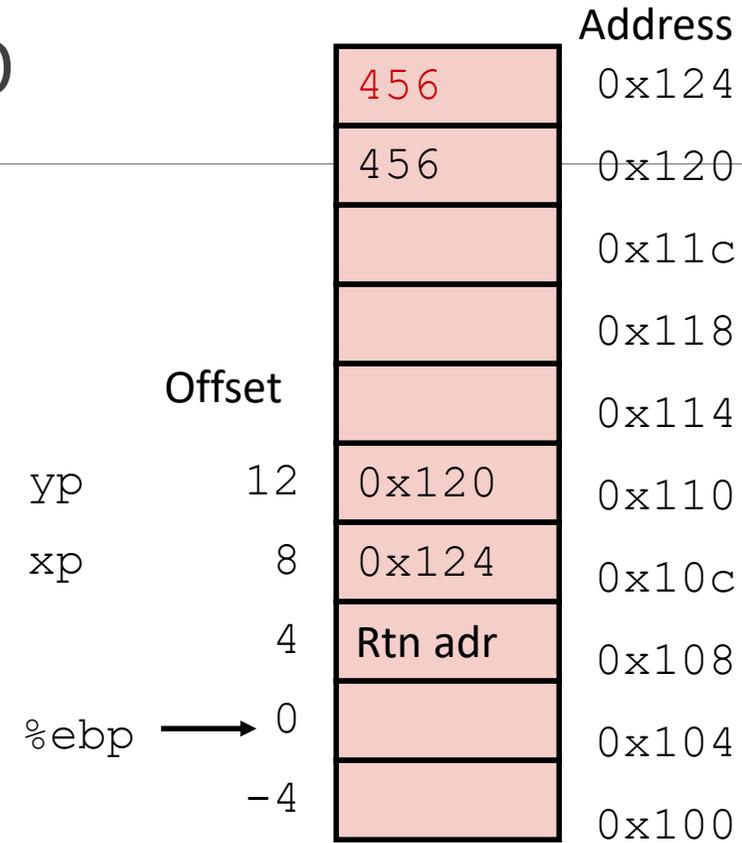


```

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
    
```


Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%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
    
```


x86-64 Integer Registers

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

<code>%rax</code>	<code>%eax</code>
<code>%rbx</code>	<code>%ebx</code>
<code>%rcx</code>	<code>%ecx</code>
<code>%rdx</code>	<code>%edx</code>
<code>%rsi</code>	<code>%esi</code>
<code>%rdi</code>	<code>%edi</code>
<code>%rsp</code>	<code>%esp</code>
<code>%rbp</code>	<code>%ebp</code>

<code>%r8</code>	<code>%r8d</code>
<code>%r9</code>	<code>%r9d</code>
<code>%r10</code>	<code>%r10d</code>
<code>%r11</code>	<code>%r11d</code>
<code>%r12</code>	<code>%r12d</code>
<code>%r13</code>	<code>%r13d</code>
<code>%r14</code>	<code>%r14d</code>
<code>%r15</code>	<code>%r15d</code>

Instructions

Long word **l** (4 Bytes) \leftrightarrow Quad word **q** (8 Bytes)

New instructions:

- **movl** \rightarrow **movq**
- **addl** \rightarrow **addq**
- **sall** \rightarrow **salq**
- etc.

32-bit instructions that generate 32-bit results

- Set higher order bits of destination register to 0
- Example: **addl**

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)
```

ret

} Setup
} 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

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:
```

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

```
    ret
```

} Setup

} Body

} Finish

64-bit data

- Data held in registers **%rax** and **%rdx**
- **movq** operation
 - “q” stands for quad-word

RISC vs CISC

RISC vs CISC

RISC: Reduced Instruction Set Computer

- Introduced Early 80's
- RISC-I (Berkeley), MIPS (Stanford), IBM 801
- Today: ARM, RISC-V (Berkeley)

CISC: Complex Instruction Set Computer

- What everything was before RISC
- VAX, x86, 68000
- Today: x86

Outcome:

- RISC in academy
- CISC in commercial space, but ...
- RISC in embedded (and under the covers)

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

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 <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Add <i>imm8</i> to <i>r/m8</i> .
REX + 80 /0 <i>ib</i>	ADD <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to <i>r/m64</i> .
81 /0 <i>iw</i>	ADD <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Add <i>imm16</i> to <i>r/m16</i> .
81 /0 <i>id</i>	ADD <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Add <i>imm32</i> to <i>r/m32</i> .
REX.W + 81 /0 <i>id</i>	ADD <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to <i>r/m64</i> .
83 /0 <i>ib</i>	ADD <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Add sign-extended <i>imm8</i> to <i>r/m16</i> .
83 /0 <i>ib</i>	ADD <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Add sign-extended <i>imm8</i> to <i>r/m32</i> .
REX.W + 83 /0 <i>ib</i>	ADD <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to <i>r/m64</i> .
00 <i>lr</i>	ADD <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Add <i>r8</i> to <i>r/m8</i> .
REX + 00 <i>lr</i>	ADD <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	Add <i>r8</i> to <i>r/m8</i> .
01 <i>lr</i>	ADD <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Add <i>r16</i> to <i>r/m16</i> .
01 <i>lr</i>	ADD <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Add <i>r32</i> to <i>r/m32</i> .
REX.W + 01 <i>lr</i>	ADD <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Add <i>r64</i> to <i>r/m64</i> .
02 <i>lr</i>	ADD <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Add <i>r/m8</i> to <i>r8</i> .
REX + 02 <i>lr</i>	ADD <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	Add <i>r/m8</i> to <i>r8</i> .
03 <i>lr</i>	ADD <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Add <i>r/m16</i> to <i>r16</i> .
03 <i>lr</i>	ADD <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Add <i>r/m32</i> to <i>r32</i> .
REX.W + 03 <i>lr</i>	ADD <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Add <i>r/m64</i> to <i>r64</i> .

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.

How did RISC happen?

Pre-1980

- Lots of hand written assembly
- Compiler technology in its infancy
- Multi-chip implementations
- Small memories at ~CPU speed

Early 80's

- VLSI makes single chip processor possible
(But only if very simple)
- Compiler technology improving

How did RISC happen?

Pre-1980

- Lots of hand written assembly
- Compiler technology in its infancy
- Multi-chip implementations
- Small memories at ~CPU speed

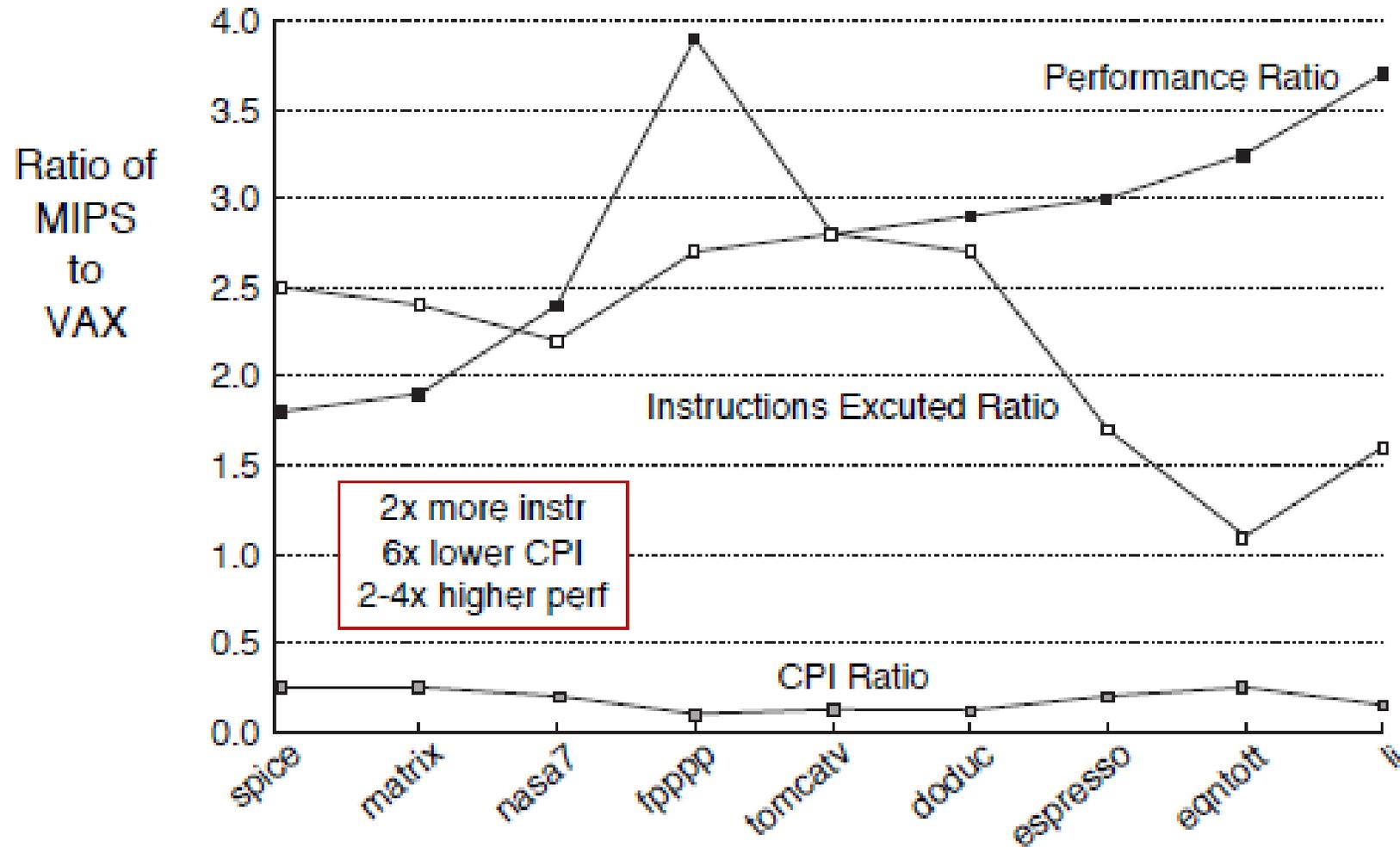
Early 80's

- VLSI makes single chip processor possible
(But only if very simple)
- Compiler technology improving

RISC goals:

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

MIPS v. VAX



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

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

Schools of ISA design & performance

$$CPU\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

Complex instruction set computer (CISC)

- Complex instructions → lots of work per instruction → fewer instructions per program
- But... more cycles per instruction & longer clock period
- *Modern μ arch gets around most of this!*

Reduced instruction set computer (RISC)

- Fine-grain instructions → less work per instruction → more instructions per program
- But... lower cycles per instruction & shorter clock period
- *Heavy reliance on compiler to “do the right thing”*

The case for RISC

CISC is fundamentally handicapped

At any 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 ...

➔ RISC will always be one step ahead of CISC!

What actually happened?

Pre-1980

- lots of hand written assembly
- Compiler technology in its infancy
- multi-chip implementations
- Small memories at ~CPU speed

Early 80's

- VLSI makes single chip processor 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

CISC's rebuttal

CISC flaws not fundamental, can be fixed with **more transistors**

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

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 internally (**μops**)

```
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)
- **Different μops for different designs**
 - **Not part of the ISA specification → Implementation flexibility**

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

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

Core precept of architecture:

Make the common case fast, make the rare case correct.

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

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

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.

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? Somewhat
 - RISC is most power-efficient today
 - ...but Intel has implemented low-power(-ish) x86 in Atom

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?**