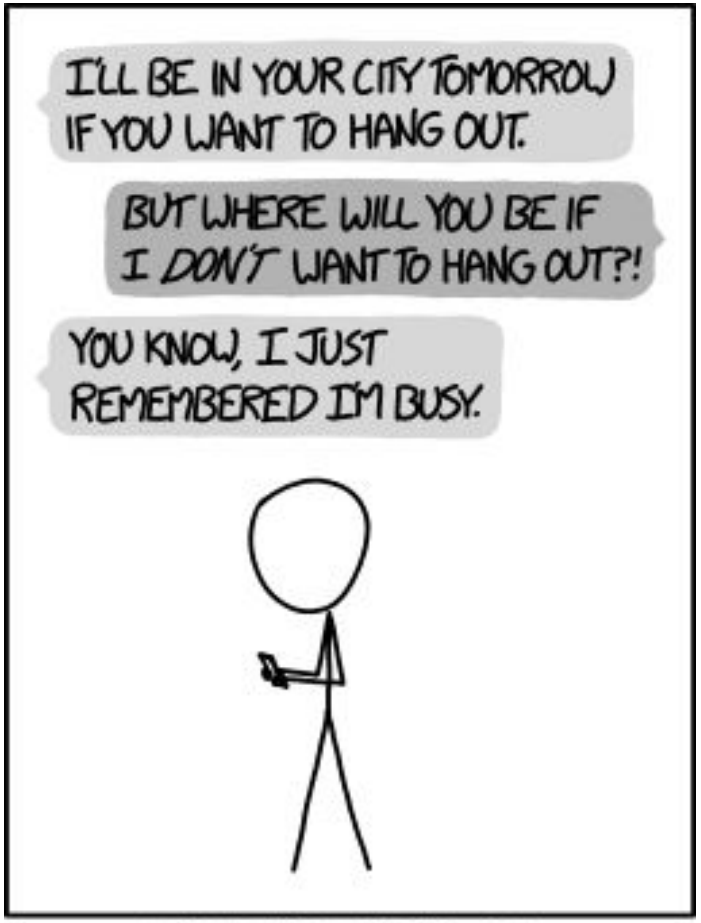
x86-64 Programming III

CSE 351 Summer 2020

Instructor:
Mara Kirdani-Ryan

Teaching Assistants:
Kashish Aggarwal
Nick Durand
Colton Jobes
Tim Mandzyuk



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

http://xkcd.com/1652/

Gentle, Loving Reminders

- Unit Summary 1 due tonight! (7/12) -- 8pm
  - Can still use late days until 7/14
- Mid-quarter Survey due Friday (7/16) – 8pm
  - Submit via Canvas!
- hw8 due tonight, hw9 due Wednesday, hw10 due friday, all at 8pm
- Justin's lecturing on Wednesday
  - My office hours are moving to Thursday

Learning Objectives

Understanding this lecture means you can…

- Choose a conditional instruction that matches your programming intent
- Translate **branches** from x86 ↔ C
- Translate **loops** from x86 ↔ C
- Translate **switches** from x86 ↔ C
- Explain why there's so much monopolization, across industries, and give a few examples of how that manifests in computing

Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation (`op`)
  - Conditionals are comparisons against 0
- Come in instruction *pairs*

```
      addq 5, (p)
 je:    *p+5 == 0
 jne:   *p+5 != 0
 jg:    *p+5 >  0
 jl:    *p+5 <  0
```

```
      orq a, b
 je:    b|a == 0
 jne:   b|a != 0
 jg:    b|a >  0
 jl:    b|a <  0
```

		(op) s, d
je	"Equal"	d (op) s == 0
jne	"Not equal"	d (op) s != 0
js	"Sign" (negative)	d (op) s < 0
jns	(non-negative)	d (op) s >= 0
jg	"Greater"	d (op) s > 0
jge	"Greater or equal"	d (op) s >= 0
jl	"Less"	d (op) s < 0
jle	"Less or equal"	d (op) s <= 0
ja	"Above" (unsigned >)	d (op) s > 0U
jb	"Below" (unsigned <)	d (op) s < 0U

Choosing instructions for conditionals

- Reminder: `cmp` is like `sub`; `test` is like `and`
  - Result is not stored anywhere

		`cmp a,b`	`test a,b`
`je`	"Equal"	`b == a`	`b&a == 0`
`jne`	"Not equal"	`b != a`	`b&a != 0`
`js`	"Sign" (negative)	`b-a < 0`	`b&a < 0`
`jns`	(non-negative)	`b-a >=0`	`b&a >= 0`
`jg`	"Greater"	`b > a`	`b&a > 0`
`jge`	"Greater or equal"	`b >= a`	`b&a >= 0`
`jl`	"Less"	`b < a`	`b&a < 0`
`jle`	"Less or equal"	`b <= a`	`b&a <= 0`
`ja`	"Above" (unsigned >)	$b >_U a$	`b&a > 0U`
`jb`	"Below" (unsigned <)	$b <_U a$	`b&a < 0U`

```
    cmpq 5, (p)
je:   *p == 5
jne:  *p != 5
jg:   *p >  5
jl:   *p <  5
```

```
    testq a, a
je:    a == 0
jne:   a != 0
jg:    a >  0
jl:    a <  0
```

```
    testb a, 0x1
je:    a_LSB == 0
jne:   a_LSB == 1
```

Choosing instructions for conditionals

		cmp a,b	test a,b
je	"Equal"	b == a	b&a == 0
jne	"Not equal"	b != a	b&a != 0
js	"Sign" (negative)	b-a < 0	b&a < 0
jns	(non-negative)	b-a >=0	b&a >= 0
jg	"Greater"	b > a	b&a > 0
jge	"Greater or equal"	b >= a	b&a >= 0
jl	"Less"	b < a	b&a < 0
jle	"Less or equal"	b <= a	b&a <= 0
ja	"Above" (unsigned >)	b > a	b&a > 0U
jb	"Below" (unsigned <)	b < a	b&a < 0U

Register	Use(s)
%rdi	argument x
%rsi	argument y
%rax	return value

```
if (x < 3) {
  return 1;
}
return 2;
```

```
  cmpq $3, %rdi
  jge T2
T1: # x < 3:
  movq $1, %rax
  ret
T2: # !(x < 3):
  movq $2, %rax
  ret
```

Practice!

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

```
cmpq   %rsi, %rdi
jle    .L4
```

```
cmpq   %rsi, %rdi
jg     .L4
```

```
testq %rsi, %rdi
jle    .L4
```

```
testq %rsi, %rdi
jg     .L4
```

We're lost…

```
long absdiff(long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    ____________________
    ____________________
                          # x > y:
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                      # x <= y:
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Choosing instructions for conditionals

		cmp a,b	test a,b
je	"Equal"	b == a	b&a == 0
jne	"Not equal"	b != a	b&a != 0
js	"Sign" (negative)	b-a < 0	b&a < 0
jns	(non-negative)	b-a >=0	b&a >= 0
jg	"Greater"	b > a	b&a > 0
jge	"Greater or equal"	b >= a	b&a >= 0
jl	"Less"	b < a	b&a < 0
jle	"Less or equal"	b <= a	b&a <= 0
ja	"Above" (unsigned >)	b > a	b&a > 0U
jb	"Below" (unsigned <)	b < a	b&a < 0U

- https://godbolt.org/z/GNxpqv

```
if (x < 3 && x == y) {
  return 1;
} else {
  return 2;
}
```

```
  cmpq $3, %rdi
  setl %al

  cmpq %rsi, %rdi
  sete %bl

  testb %al, %bl
  je T2

T1: # x < 3 && x == y:
  movq $1, %rax
  ret
T2: # else
  movq $2, %rax
  ret
```

Labels

```
swap:
    movq   (%rdi), %rax
    movq   (%rsi), %rdx
    movq   %rdx, (%rdi)
    movq   %rax, (%rsi)
    ret
```

```
max:
    movq %rdi, %rax
    cmpq %rsi, %rdi
    jg   done
    movq %rsi, %rax
done:
    ret
```

- A jump changes the program counter (`%rip`)
  - `%rip` holds the *address* of the next instruction to execute
- **Labels** give us a way to refer to a specific instruction in our assembly/machine code
  - Associated with the *next* instruction found in the assembly code (ignores whitespace)
  - Each ***use*** of the label will eventually be replaced with a reference to the final address of the labeled instruction

How do we feel about branches?

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- **Loops**
- Switches

Expressing with Goto Code

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 else:
    result = y-x;
 done:
    return result;
}
```

- C allows `goto` as means of transferring control (`jump`)
  - Closer to assembly programming style
  - Generally considered bad coding style

Compiling Loops

C/Java code:

```
while ( sum != 0 ) {
    <loop body>
}
```

Assembly code:

```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

- Other loops compiled similarly
- Most important to consider:
  - When should conditionals be evaluated? (*while* vs. *do-while*)
  - How much jumping is involved?

Compiling Loops

C/Java code:

```
while ( Test ) {
    Body
}
```

Goto version:

```
Loop: if ( !Test )  goto Exit;
      Body
      goto Loop;
Exit:
```

- What are the Goto versions of the following?
  - Do…while: Test and Body
  - For loop: Init, Test, Update, and Body

Compiling Loops

While Loop:

C:

```
while ( sum != 0 ) {
    <loop body>
}
```

x86-64:

```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

Do-while Loop:

C:

```
do {
    <loop body>
} while ( sum != 0 )
```

x86-64:

```
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```

While Loop (ver. 2):

C:

```
while ( sum != 0 ) {
    <loop body>
}
```

x86-64:

```
            testq %rax, %rax
            je    loopDone
loopTop:
            <loop body code>
            testq %rax, %rax
            jne   loopTop
loopDone:
```

For-Loop → While-Loop

For-Loop:

```
for (Init; Test; Update) {
    Body
}
```

While-Loop Version:

```
Init;
while (Test) {
    Body
    Update;
}
```

Caveat: C and Java have `break` and `continue`

- Conversion works fine for break
  - Jump to same label as loop exit condition
- But not continue: would skip doing *Update*, which it should do with for-loops
  - Introduce new label at *Update*

How do we feel about loops?

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- **Switches**

```
long switch_ex
    (long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4
- Implemented with:
  - *Jump table*
  - *Indirect jump instruction*

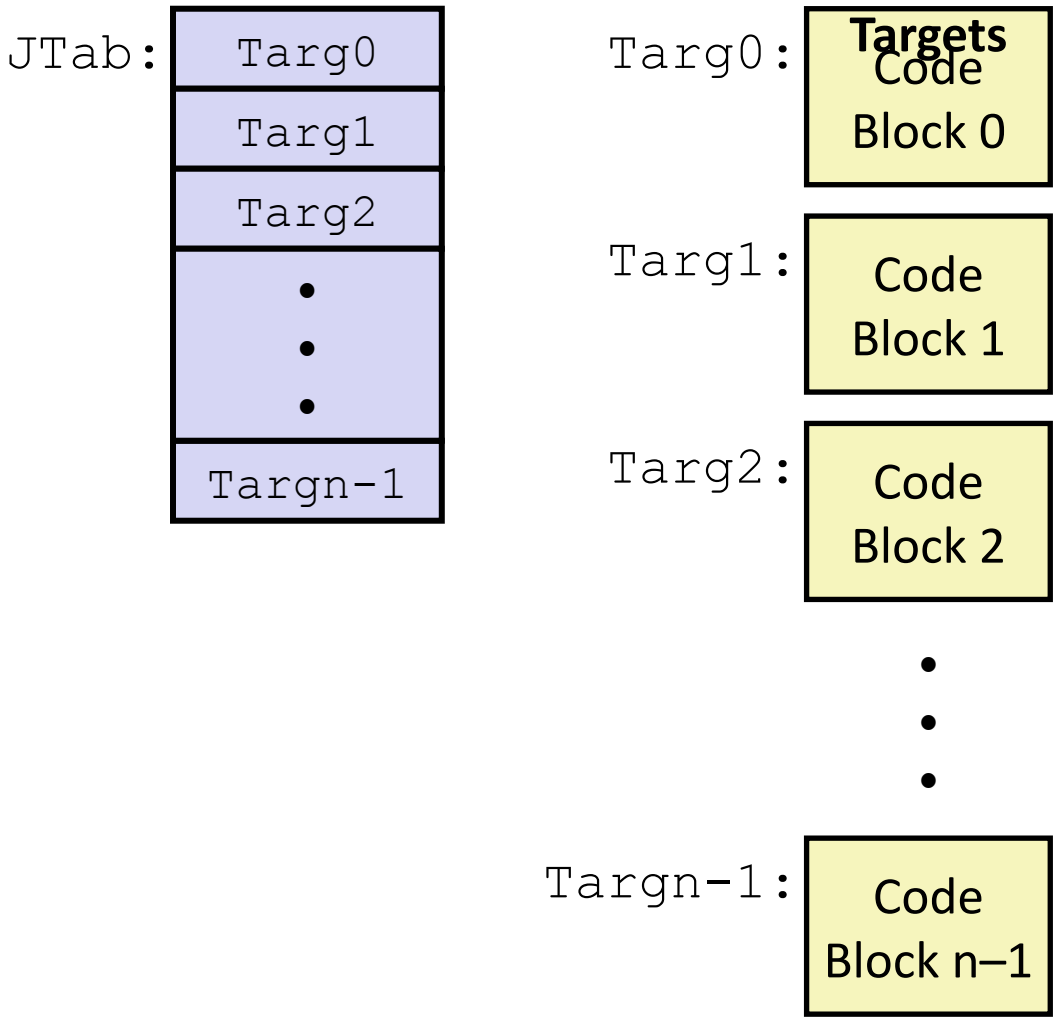
Jump Table Structure

Switch Form

```
switch (x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Approximate Translation

```
target = JTab[x];
goto target;
```



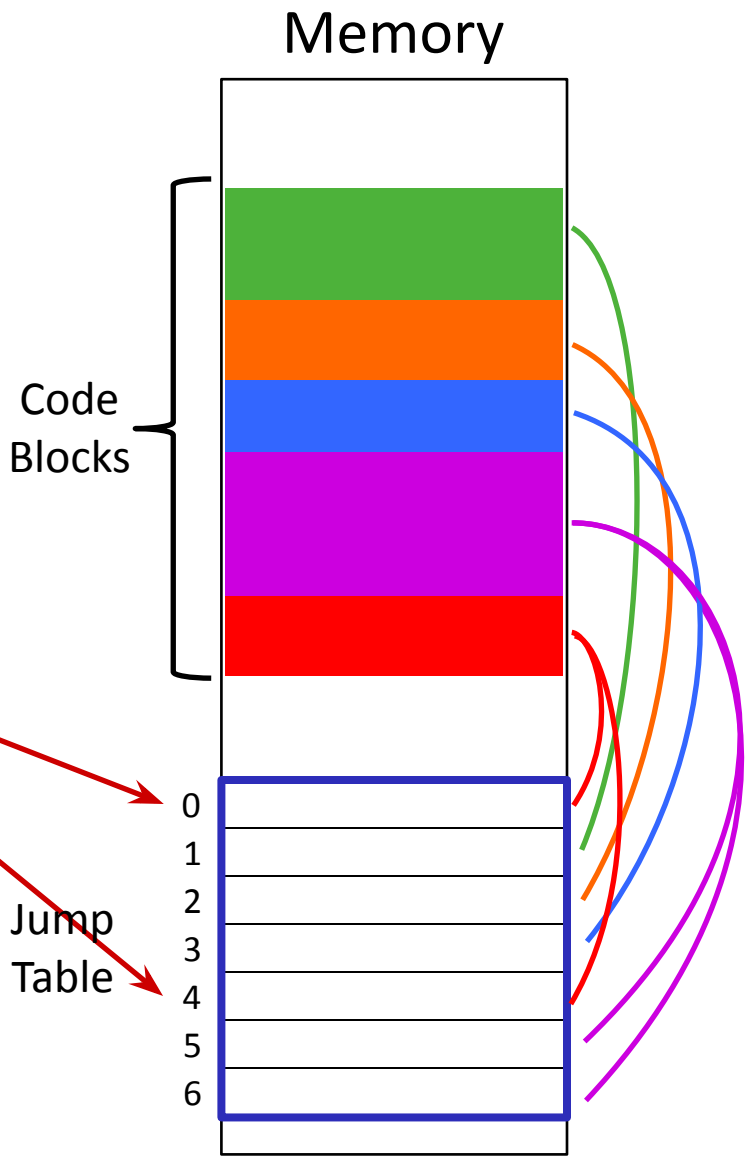
Jump Table Structure

C code:

```
switch (x) {
  case 1: <some code>
          break;
  case 2: <some code>
  case 3: <some code>
          break;
  case 5:
  case 6: <some code>
          break;
  default: <some code>
}
```

Use the jump table when x ≤ 6:

```
if (x <= 6)
  target = JTab[x];
  goto target;
else
  goto default;
```



Switch Statement Example

```
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
      . . .
    }
    return w;
}
```

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	return value

Note compiler chose to not initialize w

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # default
    jmp     *.L4(,%rdi,8)   # jump table
```

Take a look!
https://godbolt.org/z/aY24el

jump above – unsigned > catches negative default cases

Switch Statement Example

```
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
      . . .
    }
    return w;
}
```

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # default
    jmp     *.L4(,%rdi,8)   # jump table
```

Indirect jump

Jump table

```
.section .rodata
    .align 8
.L4:
    .quad    .L8  # x = 0
    .quad    .L3  # x = 1
    .quad    .L5  # x = 2
    .quad    .L9  # x = 3
    .quad    .L8  # x = 4
    .quad    .L7  # x = 5
    .quad    .L7  # x = 6
```

Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes (address)
  - Base address at `.L4`

Jump table

```
.section .rodata
    .align 8
.L4:
    .quad    .L8  # x = 0
    .quad    .L3  # x = 1
    .quad    .L5  # x = 2
    .quad    .L9  # x = 3
    .quad    .L8  # x = 4
    .quad    .L7  # x = 5
    .quad    .L7  # x = 6
```

- **Direct jump:** `jmp .L8`
  - Jump target is denoted by label `.L8`

- **Indirect jump:** `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective address `.L4 + x*8`
    - Only for $0 \le x \le 6$

How do we feel about switches?

BONUS SLIDES

Slides that expand on the simple switch code in assembly. These slides expand on material covered today, so while you don't need to read these, the information is "fair game."

Jump Table

declaring data, not instructions

8-byte memory alignment

Jump table

```
.section   .rodata
    .align 8
.L4:
    .quad   .L8 # x = 0
    .quad   .L3 # x = 1
    .quad   .L5 # x = 2
    .quad   .L9 # x = 3
    .quad   .L8 # x = 4
    .quad   .L7 # x = 5
    .quad   .L7 # x = 6
```

this data is 64-bits wide

```
switch(x) {
case 1:         // .L3
    w = y*z;
    break;
case 2:         // .L5
    w = y/z;
    /* Fall Through */
case 3:         // .L9
    w += z;
    break;
case 5:
case 6:         // .L7
    w -= z;
    break;
default:        // .L8
    w = 2;
}
```

Code Blocks (x == 1)

```
switch(x) {
  case 1:     // .L3
    w = y*z;
    break;
  . . .
}
```

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

```
.L3:
   movq    %rsi, %rax   # y
   imulq   %rdx, %rax   # y*z
   ret
```

Handling Fall-Through

```
long w = 1;
    . . .
switch (x) {
    . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
    . . .
}
```

```
case 2:
        w = y/z;
        goto merge;
```

```
case 3:
        w = 1;
merge:
        w += z;
```

More complicated choice than "just fall-through" forced by "migration" of `w = 1;`

- *Example compilation trade-off*

Code Blocks (x == 2, x == 3)

```
long w = 1;

      . . .
switch (x) {
    . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
    . . .
}
```

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

```
.L5:                        # Case 2:
   movq    %rsi, %rax       #  y in rax
   cqto                     #  div prep
   idivq   %rcx             #  y/z
   jmp     .L6              #  goto merge
.L9:                        # Case 3:
   movl    $1, %eax         #  w = 1
.L6:                        # merge:
   addq    %rcx, %rax       #  w += z
   ret
```

Code Blocks (rest)

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	Return value

```
switch (x) {
    . . .
  case 5:   // .L7
  case 6:   // .L7
    w -= z;
    break;
  default: // .L8
    w = 2;
}
```

```
.L7:                    # Case 5,6:
  movl   $1, %eax       #   w = 1
  subq   %rdx, %rax     #   w -= z
  ret
.L8:                    # Default:
  movl   $2, %eax       #   2
  ret
```

GDB Demo

- The `movz` and `movs` examples on a real machine!
  - `movzbq %al, %rbx`
  - `movsbl (%rax), %ebx`
- You will *need* to use GDB to get through Lab 2
  - Useful debugger in this class and beyond!
- Pay attention to:
  - Setting breakpoints (`break`)
  - Stepping through code (`step`/`next` and `stepi`/`nexti`)
  - Printing out data (`print` – works with regs & vars)
  - Examining **memory** (`x`)

Now, the fun bits!

Processor History and Values

In your groups:

- How many different phone brands? OS brands?
- Computer brands? OS brands?
- How many different companies total?

we'll be talking about monopolies, I just want us to get started

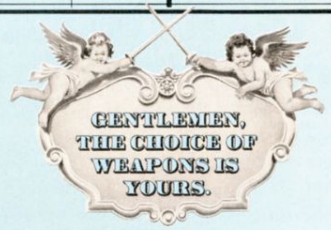
x86 History

- x86: Compatible to 8086, released in 1977
  - 8086: 16-bit processor designed along iAPX 432
- iAPX 432
  - First 32-bit processor, completely new ISA
  - OOP, garbage collection, multitasking from hardware!
  - No visible registers! First IEEE 754 implementation!
  - Too many new features, ended up being slower and more expensive, lots of product delays
- Intel: Release something so we can compete with Zilog, Motorola, others

The Battle of the 80's

Think of your next microcomputer as a weapon against horrendous inefficiencies, outrageous costs and antiquated speeds. We invite you to peruse this chart.

Features:	8080A	Z80-CPU	Features:	8080A	Z80-CPU
Power Supplies	+5,−5,+12	+5	Instructions	78	158*
Clock	2Φ,+12 Volt	1Φ,5 Volt	OP Codes	244	696
Standard Clock Speed	500 ns	400 ns	Addressing Modes	7	11
Interface	Requires 8222, 8228 & 8224	Requires no other logic and includes dynamic RAM Refresh	Working Registers	8	17
			Throughput	Up to 5 times greater than the 8080A	
Interrupt	1 mode	3 modes; up to 6X faster	Program Memory Space	Generally 50% less than the 8080A	
Non-maskable Interrupt	No	Yes	*Including all of the 8080A's instructions.		



Announcing Zilog Z-80 microcomputer products. With the next generation, the battle is joined.

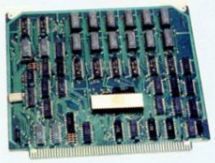
The Z-80: A new generation LSI component set including CPU and I/O Controllers.

The Z-80: Full software support with emphasis on high-level languages.

The Z-80: A floppy disc-based development system with advanced real-time debug and in-circuit emulation capabilities.

The Z-80: Multiple sourcing available now.

Your ammunition: A chip off a new block.



A single chip, N-channel processor arms you with a super-set of 158 instructions that include *all* of the 8080A's 78 instructions with *total* software compatibility. The new instructions include 1, 4, 8 and 16-bit operations. And that means less programming time, less paper and less end costs.

And you'll be in command of powerful instructions: Memory-to-memory or memory-to-I/O block transfers and searches, 16-bit arithmetic, 9 types of rotates and shifts, bit manipulation and a legion of addressing modes. Along with this army you'll also get a standard instruction speed of 1.6 μs and all Z-80 circuits require only a single 5V power supply and a single phase 5V clock. And you should know that a family of Z-80 programmable circuits allow for direct interface to a wide range of both parallel and serial interface peripherals and even dynamic memories without other external logic.

With these features, the Z80-CPU generally requires approximately 50% less memory space for program storage yet provides up to 500% more throughput than the 8080A. Powerful ammunition at a surprisingly low cost and ready for immediate shipment.

Mighty weapons against an enemy entrenched: The Z-80 development system.

You'll be equipped with performance and versatility unmatched by any other microcomputer development system in the field. Thanks to a floppy disc operating system in alliance with a sophisticated Real-Time Debug Module.

The Zilog battalion includes:

- Z80-CPU Card.
- 16K Bytes of RAM Memory, expandable to 60K Bytes.
- 4K Bytes of ROM/RAM Monitor software.
- Real-Time Debug Module and In-Circuit Emulation Module.
- Dual Floppy Disc System.
- Optional I/O Ports for other High Speed Peripherals are also available.
- Complete Software Package including Z-80 Assembler, Editor, Disc Operating System, File Maintenance and Debug.

On standby: Software support.

All this is supported by a contingent of software including: resident microcomputer software, time sharing programs, libraries and high-level languages such as PL/Z.

On standby: User support.

Zilog conducts a wide range of strategic meetings and design oriented workshops to provide the know-how required to implement the Z-80 Microcomputer Product line into your design. All hardware, software and the development system are thoroughly explained with "hands-on" experience in the classroom. Your Zilog representative can provide you with further details on our user support program.

Reinforcements: A reserve of technological innovations.

The Zilog Z-80 brings to the battlefront new levels of performance and ease of programming not available in second generation systems. And while all the others busy themselves with overtaking the Z-80, we're busy on the next generation—continuing to demonstrate our pledge to stay a generation ahead.

The Z-80's troops are the specialists who were directly responsible for the development of the most successful first and second generation microprocessors. Nowhere in the field is there a corps of seasoned veterans with such a distinguished record of victory.

Signal us for help. We'll dispatch appropriate assistance.

Zilog MICROCOMPUTERS

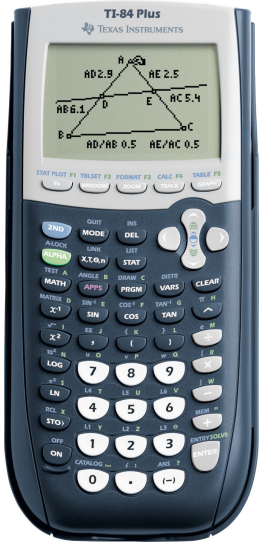
170 State Street, Los Altos, California 94022
(415) 941-5055/TWX 910-370-7955

Circle 33 on reader service card

AN AFFILIATE OF EXXON ENTERPRISES INC.

The "Battle of the 80s"

- Many companies making processors/computers
  - Dominated by IBM
  - Also National Semiconductor, Motorola
- Notably Zilog, started by former Intel engineers
  - Z-80 was spreading, Intel wanted market power



Computer Chip Manufacturers

1978:

- Intel
- National Semiconductor
- Harris Corp
- NEC
- DEC
- IBM
- Motorola
- Hitachi
- Zilog
- ...

2021*:

- Intel
- TSMC
- Samsung

*Others, but these dominate

Intel's domination

Worldwide Semiconductor Sales Leaders ($B)

Rank	1985		1990		1995		2000		2006		2011F	
1	NEC	2.1	NEC	4.8	Intel	13.6	Intel	29.7	Intel	31.6	Intel	50.6
2	TI	1.8	Toshiba	4.8	NEC	12.2	Toshiba	11.0	Samsung	19.7	Samsung	34.5
3	Motorola	1.8	Hitachi	3.9	Toshiba	10.6	NEC	10.9	TI	13.7	Toshiba	13.5
4	Hitachi	1.7	Intel	3.7	Hitachi	9.8	Samsung	10.6	Toshiba	10.0	TI	12.8
5	Toshiba	1.5	Motorola	3.0	Motorola	8.6	TI	9.6	ST	9.9	Renesas	11.3
6	Fujitsu	1.1	Fujitsu	2.8	Samsung	8.4	Motorola	7.9	Renesas	8.2	ST	9.6
7	Philips	1.0	Mitsubishi	2.6	TI	7.9	ST	7.9	Hynix	7.4	Qualcomm*	9.6
8	Intel	1.0	TI	2.5	IBM	5.7	Hitachi	7.4	Freescale	6.1	Hynix	9.4
9	National	1.0	Philips	1.9	Mitsubishi	5.1	Infineon	6.8	NXP	5.9	Micron	8.7
10	Matsushita	0.9	Matsushita	1.8	Hynudai	4.4	Philips	6.3	NEC	5.7	Broadcom*	7.1
Top 10 Total ($B)		13.9		31.8		86.3		108.1		118.2		167.1
Semi Market ($B)		23.3		54.3		154		218.6		264.6		321.3
Top 10 % of Total Semi Mrkt		60%		59%		56%		49%		45%		52%

Source: IC Insights

*Fabless

From CPU to software, the 8080 Microcom puter is here.

Intel's new 8080 n-channel microcomputer is here—incredibly easy to interface, simple to program and with up to 100 times the performance of p-channel MOS microcomputers.

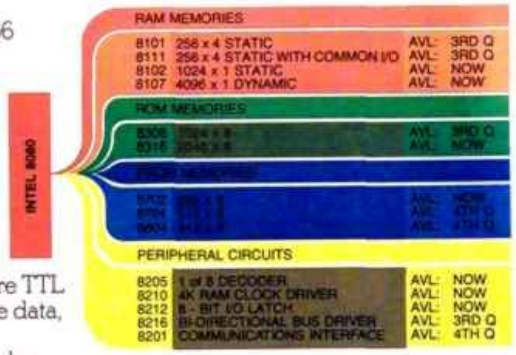
Best of all, the 8080 is real—in production at Intel and available in volume quantities, today. It's also available through distributors along with a growing line of peripheral circuits and a new version of the Intellec 8, a program and hardware development system for the 8080, all supported with software packages, design documentation and manuals, and backed by more than 100 man years of microcomputer expertise.

The 8080 is the inevitable successor to complex custom MOS and many large discrete logic subsystems. It is the industry's first general purpose n-channel microcomputer and the first high performance single-chip CPU, with extremely simple interface requirements and straightforward programming. It runs a full instruction cycle in 2 microseconds.

As such, the 8080 extends the economic benefits of Intel's p-channel microcomputers to a new universe of systems that need fast, multi-port controllers and processors. These systems include intelligent terminals, point of sale systems, process and numeric controllers, advanced calculators, word processors, self-calibrating instruments, data loggers, communications controllers, and many more.

You can use 256 input and 256 output channels, handle almost unlimited interrupt levels, directly access 64 kilobytes of memory, and put many satellite 8080 processors around a single memory.

Interfacing is minimal and design is easy with the 8080 because all controls are fully decoded on the CPU chip itself and inputs and outputs are TTL compatible. There are separate data, address and control buses.



INTEL 8080 PRODUCT FAMILY

The 8080 microcomputer has 78 basic instructions, including the 8008 set plus new ones that make possible such features as vectored multi-level interrupt, unlimited subroutine nesting and very fast decimal and binary arithmetic.

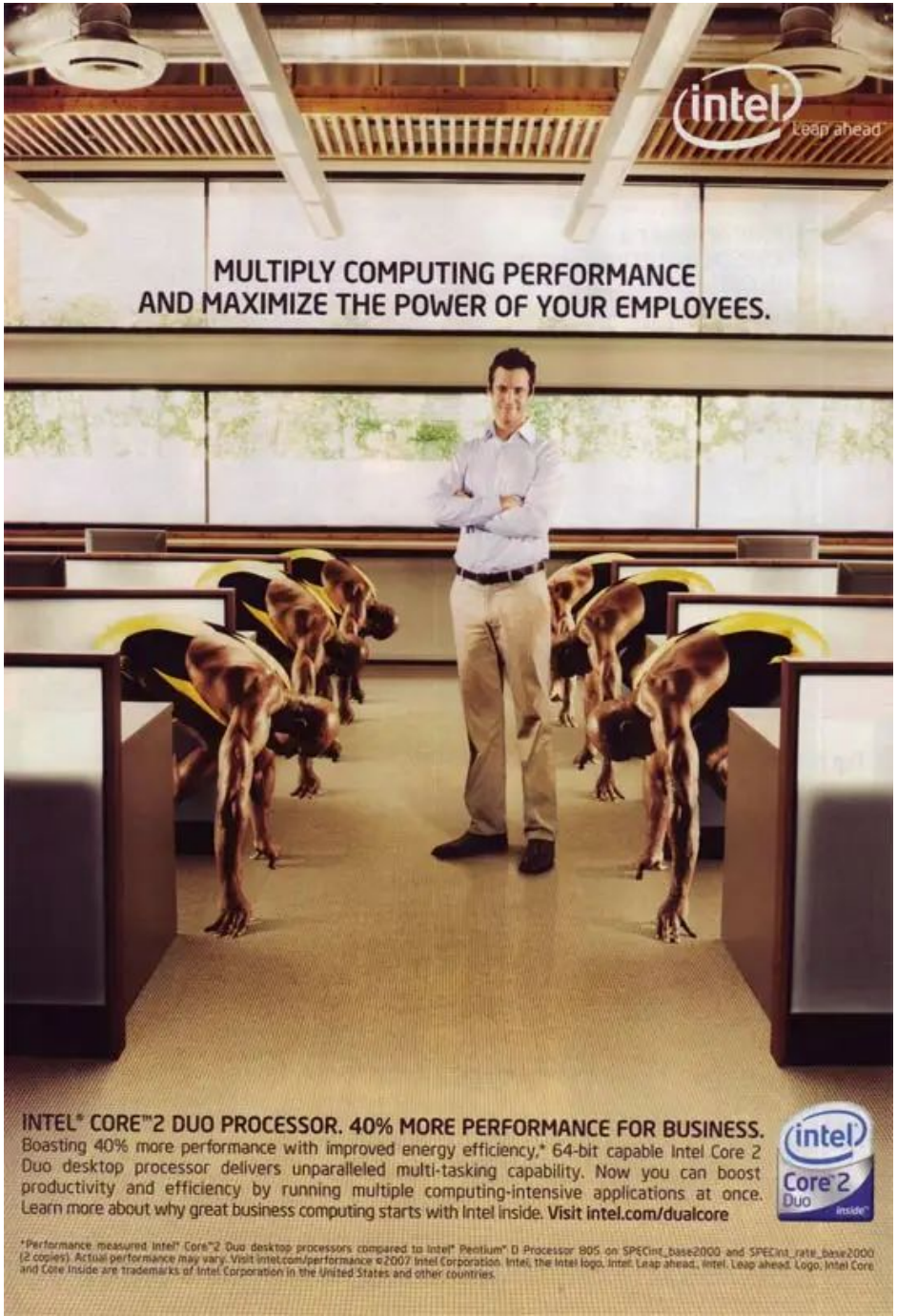
Program development for the 8080 can be done either on a large computer using the Intel software cross products (PL/M systems language compiler, macro-assembler and simulator), or on an Intellec 8 development system with a resident monitor, text editor and macro-assembler.

The new 8080 product family includes performance matched peripheral and memory circuits configured to minimize design effort and maximize system performance. Large, low cost RAMs, ROMs, PROMs and I/O devices are available now and we will soon announce other 8080 LSI support circuits.

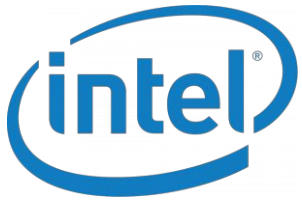
The 8080 is easier to use and more economical than any high performance microcomputer in sight. It's here now, in volume, from the inventors of the microcomputer and the people who lead the industry in production and design support.

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051. (408) 246-7501.

intel Microcomputers. First from the beginning.


intel
Leap ahead
MULTIPLY COMPUTING PERFORMANCE
AND MAXIMIZE THE POWER OF YOUR EMPLOYEES.
INTEL® CORE™2 DUO PROCESSOR. 40% MORE PERFORMANCE FOR BUSINESS.
Boasting 40% more performance with improved energy efficiency,* 64-bit capable Intel Core 2 Duo desktop processor delivers unparalleled multi-tasking capability. Now you can boost productivity and efficiency by running multiple computing-intensive applications at once. Learn more about why great business computing starts with Intel inside. Visit intel.com/dualcore
intel
Core 2 Duo
inside
*Performance measured Intel® Core™2 Duo desktop processors compared to Intel® Pentium® D Processor 805 on SPECint_base2000 and SPECint_rate_base2000 (2 copies). Actual performance may vary. Visit intel.com/performance ©2007 Intel Corporation. Intel, the Intel logo, Intel Leap ahead., Intel. Leap ahead. Logo, Intel Core and Core Inside are trademarks of Intel Corporation in the United States and other countries.

Exceedingly Dominant ISAs



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

Macbooks & PCs
(Core i3, i5, i7, i9)
x86-64 Instruction Set

ARM architectures

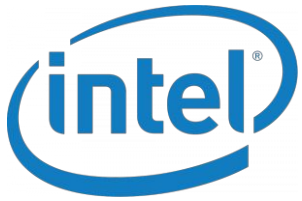
Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

Designer	University of California, Berkeley
Bits	32 · 64 · 128
Introduced	2010
Version	unprivileged ISA 20191213,[1] privileged ISA 20190608[2]
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little[1][3]

Architecture research, with some notable industry buy-in
RISC-V ISA

Exceedingly Dominant ISAs



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

Macbooks & PCs
(Core i3, i5, i7, i9)
x86-64 Instruction Set

ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

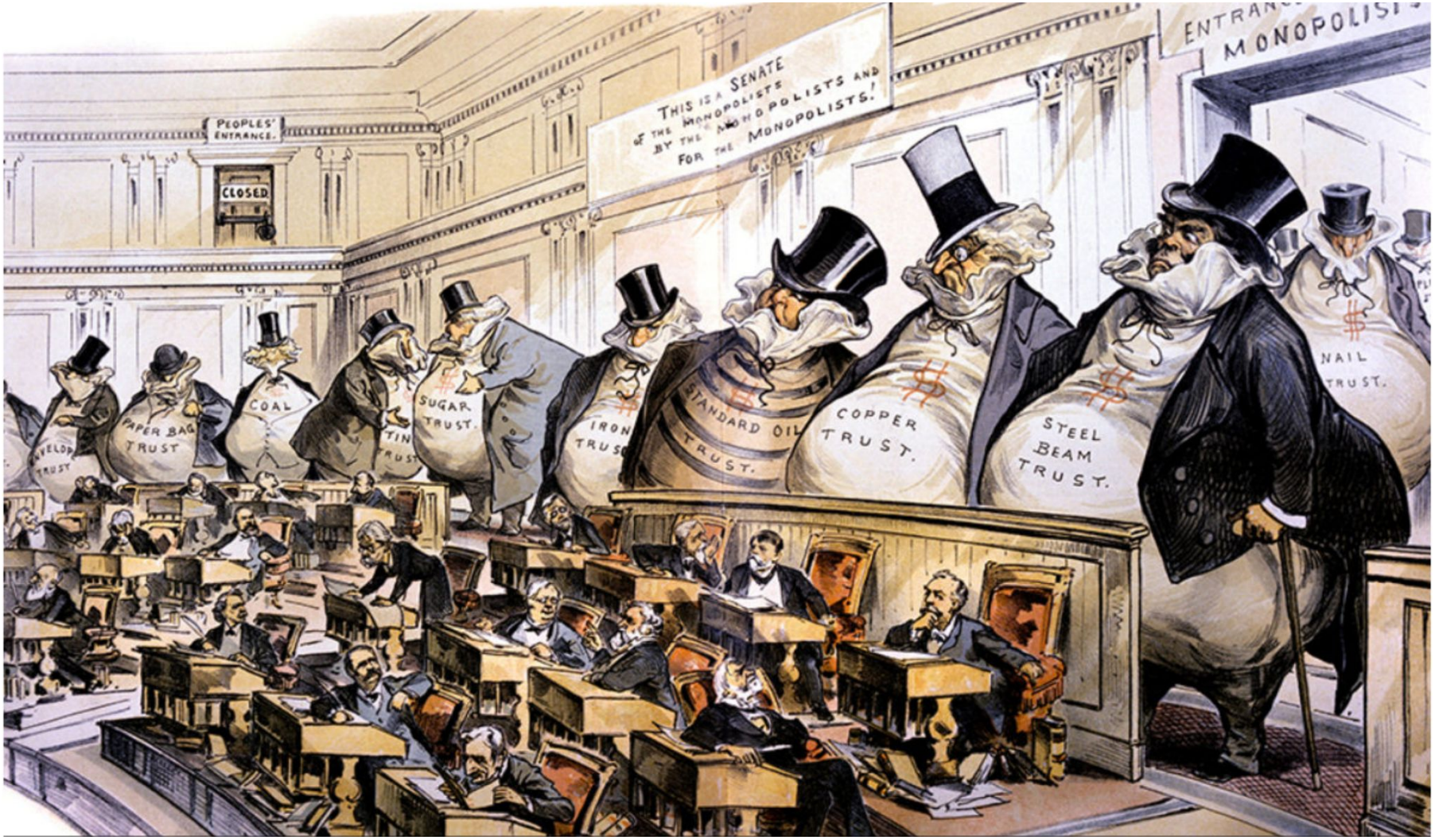
Designer	University of California, Berkeley
Bits	32 · 64 · 128
Introduced	2010
Version	unprivileged ISA 20191213,[1] privileged ISA 20190608[2]
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little[1][3]

Architecture research, with some notable industry buy-in
RISC-V ISA

We only have 3* chip manufacturers, two* ISAs, what happened?

Narrative from Cory Doctorow's 2020 Colloq

Sherman Antitrust Act (1890)



Sherman Antitrust Act (1890)

- Outlaws monopolies
  - Also "*every* contract, combination, or conspiracy in restraint of trade."
- Standard Oil Co. controlled 91% of oil production!
  - John D. Rockefeller's company, if you were wondering
  - Split into 36 companies (Exxon, Mobil, among others)
  - "We didn't restrain trade, we were just superior competitors…"
  - ...sure...

Illegal under Sherman Antitrust

- Merging with major competitor, like...
  - Exxon and Mobil in 1999
  - AOL and Time Warner in 2000
  - Comcast and AT&T in 2001
  - Heinz and Kraft in 2015
- Buying a smaller competitor, like...
  - Facebook buying instagram for $1B
  - Facebook buying whatsapp for $22B
  - Salesforce buying Slack for $27.7B
  - Amazon buys Whole Foods for $13.7B
  - Verizon buys AOL/Yahoo for $4.4B

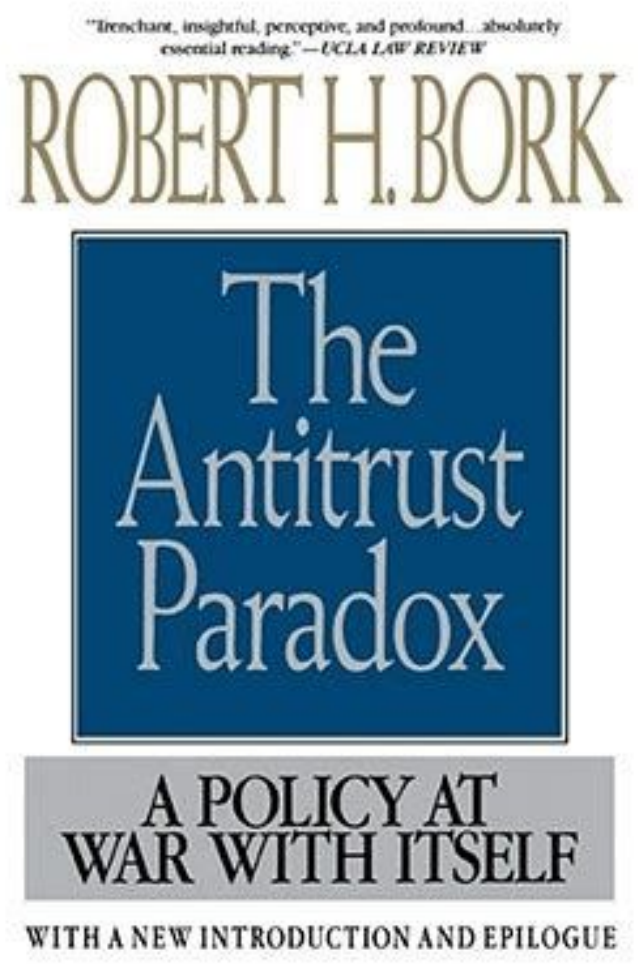
But, wasn't this illegal?

It was….but...

"To be Bork'd"

Consumer Harm Theory

- Argued "inefficiency" of antitrust law
  - Raised prices, etc.
- Antitrust *should* focus on efficiency, prices
  - "Consumer Welfare"



We should ignore monopolistic strategies unless we can prove that it'll harm "consumer welfare" (i.e. price increase)

We should ignore monopolistic strategies unless we can prove that it'll harm "consumer welfare" (i.e. price increase)

Mind you, this is almost impossible to prove.

Guess who loved this?

Neoliberalism!

Neoliberalism

- "The market knows best, we shouldn't interfere"
  - Society should be shaped by the free market
  - Deregulation of private industries
  - Reduction of low-income government supports
  - Tax breaks for wealthy creates jobs
  - Reshape public services in private image
  - "Corportizing" of education, healthcare, prisons, etc.
  - Individual is more important than collective
  - "Liberalism" (freedom) for corporations
- **Go learn more!**
  - *"Being in the US and not understanding neoliberalism is like being in the USSR and not understanding communism"*

The First IBM PC

- IBM: dominated "mini" computer market (70%), wanted "micro" computer market
- Market research showed that non-proprietary parts were preferred by retailers (for repairs)
- Most engineers were hobbyists
- Low-cost, quick design (30 days to prototype)
- Open architecture!
- Used Intel 8088!

IBM & Antitrust

- Spent more on lawyers than the ENTIRE DoJ antitrust division (1969 - 1981)
- Was so scared of inviting antitrust scrutiny that they outsourced OS creation to Paul Allen, Bill Gates
- Open-source ISAs and peripherals helps make the case too, especially to regulators

- It's not perfect, but a company controlling 70% of computer, punch card, tabulating markets was scared of the DoJ!

Illegal under Sherman Antitrust

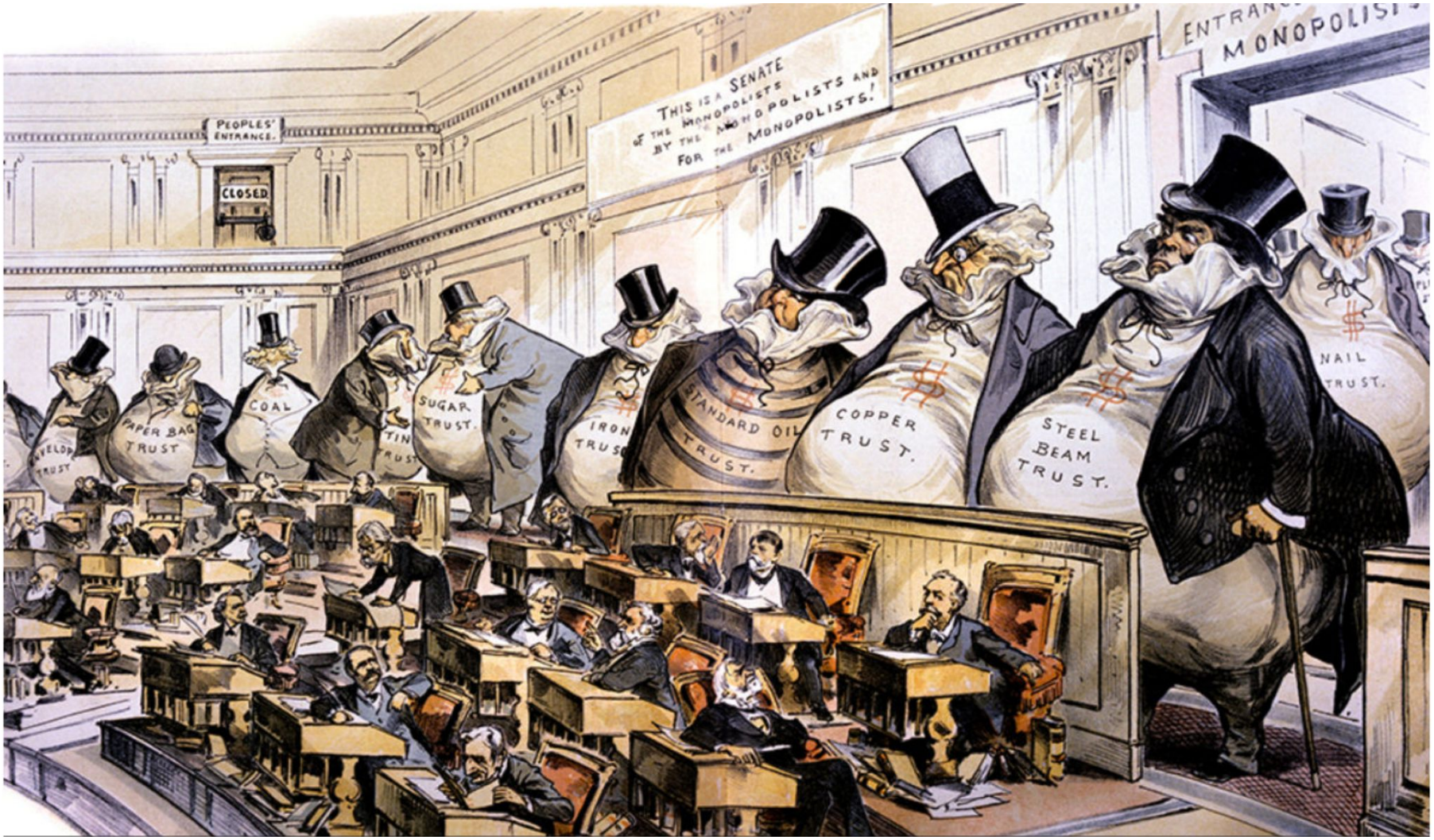
- Merging with major competitor, like...
  - Exxon and Mobil in 1999
  - AOL and Time Warner in 2000
  - Comcast and AT&T in 2001
  - Heinz and Kraft in 2015
- Buying a smaller competitor, like...
  - Facebook buying instagram for $1B
  - Facebook buying whatsapp for $22B
  - Salesforce buying Slack for $27.7B
  - Amazon buys Whole Foods for $13.7B
  - Verizon buys AOL/Yahoo for $4.4B

Later:

- "Microsoft is trying to become the IBM of the 1990s"
  - First ruling to break up Microsoft,
  - Appealed, still violated law, but no longer breaking up
- Trump met with big tech leaders in 2016, *all fit around one table*
- 3 companies make operating systems
- 3-4 companies make phones
- "5 giant websites filled with screen-shots from the other four" (Doctorow)

Tech consistently has more money than anyone knows what to do with...so, why not buy legislative action?

Different today? Not entirely.




NBC
Parks and Recreation