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 \(\text{op}\)
  - Conditionals are comparisons against 0
- Come in instruction pairs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Condition Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq 5, (p)</code></td>
<td><code>je</code></td>
<td><code>*p+5 == 0</code></td>
</tr>
<tr>
<td></td>
<td><code>jne</code></td>
<td><code>*p+5 != 0</code></td>
</tr>
<tr>
<td></td>
<td><code>jg</code></td>
<td><code>*p+5 &gt; 0</code></td>
</tr>
<tr>
<td></td>
<td><code>jl</code></td>
<td><code>*p+5 &lt; 0</code></td>
</tr>
<tr>
<td><code>orq a, b</code></td>
<td><code>je</code></td>
<td>`b</td>
</tr>
<tr>
<td></td>
<td><code>jne</code></td>
<td>`b</td>
</tr>
<tr>
<td></td>
<td><code>jg</code></td>
<td>`b</td>
</tr>
<tr>
<td></td>
<td><code>jl</code></td>
<td>`b</td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

- Reminder: \texttt{cmp} is like \texttt{sub}; \texttt{test} is like \texttt{and}
  - Result is not stored anywhere

<table>
<thead>
<tr>
<th>Instruction</th>
<th>cmp \texttt{a,b}</th>
<th>test \texttt{a,b}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{je}</td>
<td>\texttt{b == a}</td>
<td>\texttt{b&amp;a == 0}</td>
</tr>
<tr>
<td>\texttt{jne}</td>
<td>\texttt{b != a}</td>
<td>\texttt{b&amp;a != 0}</td>
</tr>
<tr>
<td>\texttt{js}</td>
<td>\texttt{b-a &lt; 0}</td>
<td>\texttt{b&amp;a &lt; 0}</td>
</tr>
<tr>
<td>\texttt{jns}</td>
<td>\texttt{b-a &gt;=0}</td>
<td>\texttt{b&amp;a &gt;= 0}</td>
</tr>
<tr>
<td>\texttt{jg}</td>
<td>\texttt{b &gt; a}</td>
<td>\texttt{b&amp;a &gt; 0}</td>
</tr>
<tr>
<td>\texttt{jge}</td>
<td>\texttt{b &gt;= a}</td>
<td>\texttt{b&amp;a &gt;= 0}</td>
</tr>
<tr>
<td>\texttt{jl}</td>
<td>\texttt{b &lt; a}</td>
<td>\texttt{b&amp;a &lt; 0}</td>
</tr>
<tr>
<td>\texttt{jle}</td>
<td>\texttt{b &lt;= a}</td>
<td>\texttt{b&amp;a &lt;= 0}</td>
</tr>
<tr>
<td>\texttt{ja}</td>
<td>\texttt{b &gt;U a}</td>
<td>\texttt{b&amp;a &gt; 0U}</td>
</tr>
<tr>
<td>\texttt{jb}</td>
<td>\texttt{b &lt;U a}</td>
<td>\texttt{b&amp;a &lt; 0U}</td>
</tr>
</tbody>
</table>

\begin{align*}
\text{cmpq} & \quad 5, \ (p) \\
\text{je}: & \quad *p == 5 \\
\text{jne}: & \quad *p != 5 \\
\text{jg}: & \quad *p > 5 \\
\text{jl}: & \quad *p < 5 \\
\text{testq} & \quad a, \ a \\
\text{je}: & \quad a == 0 \\
\text{jne}: & \quad a != 0 \\
\text{jg}: & \quad a > 0 \\
\text{jl}: & \quad a < 0 \\
\text{testb} & \quad a, \ 0x1 \\
\text{je}: & \quad a_{\text{LSB}} == 0 \\
\text{jne}: & \quad a_{\text{LSB}} == 1
\end{align*}
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

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

cmp \(a, b\)  | test \(a, b\) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>je  &quot;Equal&quot;</td>
<td>(b == a)</td>
</tr>
<tr>
<td>jne &quot;Not equal&quot;</td>
<td>(b != a)</td>
</tr>
<tr>
<td>js &quot;Sign&quot; (negative)</td>
<td>(b-a &lt; 0)</td>
</tr>
<tr>
<td>jns (non-negative)</td>
<td>(b-a &gt;=0)</td>
</tr>
<tr>
<td>jg &quot;Greater&quot;</td>
<td>(b &gt; a)</td>
</tr>
<tr>
<td>jge &quot;Greater or equal&quot;</td>
<td>(b &gt;= a)</td>
</tr>
<tr>
<td>jl &quot;Less&quot;</td>
<td>(b &lt; a)</td>
</tr>
<tr>
<td>jle &quot;Less or equal&quot;</td>
<td>(b &lt;= a)</td>
</tr>
<tr>
<td>ja &quot;Above&quot; (unsigned &gt;)</td>
<td>(b &gt; a)</td>
</tr>
<tr>
<td>jb &quot;Below&quot; (unsigned &lt;)</td>
<td>(b &lt; a)</td>
</tr>
</tbody>
</table>
Practice!

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

🐶 **cmpq** %rsi, %rdi

猫咪 **cmpq** %rsi, %rdi

🐑 **testq** %rsi, %rdi

🦄 **testq** %rsi, %rdi

 banco **we’re lost…**

```c
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>“Equal”</td>
<td>b == a</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>b != a</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>b-a &lt; 0</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>b-a &gt;=0</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>b &gt;= a</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>b &lt;= a</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>

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

```assembly
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
```

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

- 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

```
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
```
How do we feel about branches?
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
Expressing with Goto Code

C allows `goto` as means of transferring control (jump)

- Closer to assembly programming style
- Generally considered bad coding style

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

```c
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;
}
```
Compiling Loops

C/Java code:

```java
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:

```c
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:   jmp   loopTop
```

**While Loop (ver. 2):**

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

x86-64:
```
loopTop:    testq  %rax, %rax
           je    loopDone
           <loop body code>
           testq  %rax, %rax
           jne   loopTop
loopDone:
```
For-Loop → While-Loop

For-Loop:

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

While-Loop Version:

```c
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;
}
Jump Table Structure

Switch Form

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

Jump Table

<table>
<thead>
<tr>
<th>JTab</th>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>Targn-1</th>
</tr>
</thead>
</table>

Jump

<table>
<thead>
<tr>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
</tr>
<tr>
<td>Block 0</td>
</tr>
<tr>
<td>Code</td>
</tr>
<tr>
<td>Block 1</td>
</tr>
<tr>
<td>Code</td>
</tr>
<tr>
<td>Block 2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Code</td>
</tr>
<tr>
<td>Block n-1</td>
</tr>
</tbody>
</table>

Approximate Translation

```c
target = JTab[x];
goto target;
```
Jump Table Structure

C code:

```c
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 \( x \leq 6 \):

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

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

**Register Use(s)**
- `%rdi` 1\(^{st}\) argument (x)
- `%rsi` 2\(^{nd}\) argument (y)
- `%rdx` 3\(^{rd}\) 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

`jump above` – unsigned > catches negative default cases

Take a look!
[https://godbolt.org/z/aY24el](https://godbolt.org/z/aY24el)
Switch Statement Example

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

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
```

Switch example:

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

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

- **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 $0 \leq x \leq 6$

---

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
```
How do we feel about switches?
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

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
```

8-byte memory alignment

```
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;
}
```

this data is 64-bits wide
Code Blocks (x == 1)

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Handling Fall-Through

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

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

```c
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)

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

.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)

```c
switch (x) {
    . . .
    case 5: // .L7
        w -= z;
        break;
    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
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
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
L09: Assembly III

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.

<table>
<thead>
<tr>
<th>Features:</th>
<th>8080A</th>
<th>Z80-CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Supplies</td>
<td>+5, -5, +12 V</td>
<td>+5 V</td>
</tr>
<tr>
<td>Clock</td>
<td>24, +12 Volt</td>
<td>16.5 Volt</td>
</tr>
<tr>
<td>Standard Clock</td>
<td>500 ns</td>
<td>400 ns</td>
</tr>
<tr>
<td>Speed</td>
<td></td>
<td>Addressing Modes</td>
</tr>
<tr>
<td>Interface</td>
<td>Requires 8222, 8223 &amp; 8224</td>
<td>Requires no other logic and includes dynamic RAM Refresh</td>
</tr>
<tr>
<td>Throughput</td>
<td></td>
<td>Up to 5 times greater than the 8080A</td>
</tr>
<tr>
<td>Interrupt</td>
<td>1 mode</td>
<td>3 modes up to 6X faster</td>
</tr>
<tr>
<td>Non-maskable Interrupt</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*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 debugging 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 16 ns 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.

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.

Your enemy entangled: 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 Z80 Assembler, Editor, Disc Operating System, File Maintenance and Debug.

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 over-taking 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.

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.

Zilog MICROCOMPUTERS

170 State Street, Los Altos, California 94022

(415) 944-5255/TWX 900-370-7965

Circle 33 on reader service card

AN AFFILIATE OF EXON 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NEC</td>
<td>NEC</td>
<td>Intel</td>
<td>Intel</td>
<td>Intel</td>
<td>Intel</td>
</tr>
<tr>
<td>2</td>
<td>TI</td>
<td>Toshiba</td>
<td>NEC</td>
<td>Toshiba</td>
<td>Samsung</td>
<td>Samsung</td>
</tr>
<tr>
<td>3</td>
<td>Motorola</td>
<td>Hitachi</td>
<td>Toshiba</td>
<td>NEC</td>
<td>Samsung</td>
<td>Toshiba</td>
</tr>
<tr>
<td>4</td>
<td>Hitachi</td>
<td>Intel</td>
<td>Hitachi</td>
<td>TI</td>
<td>ST</td>
<td>Ti</td>
</tr>
<tr>
<td>5</td>
<td>Toshiba</td>
<td>Motorola</td>
<td>Motorola</td>
<td>TI</td>
<td>ST</td>
<td>ReNesas</td>
</tr>
<tr>
<td>6</td>
<td>Fujitsu</td>
<td>Fujitsu</td>
<td>Samsung</td>
<td>Motorola</td>
<td>Renesas</td>
<td>ST</td>
</tr>
<tr>
<td>7</td>
<td>Philips</td>
<td>Mitsubishi</td>
<td>Ti</td>
<td>ST</td>
<td>Hynix</td>
<td>Qualcomm*</td>
</tr>
<tr>
<td>8</td>
<td>Intel</td>
<td>Ti</td>
<td>IBM</td>
<td>Hitachi</td>
<td>Freescale</td>
<td>Hynix</td>
</tr>
<tr>
<td>9</td>
<td>National</td>
<td>Philips</td>
<td>Mitsubishi</td>
<td>Infineon</td>
<td>NXP</td>
<td>Micron</td>
</tr>
<tr>
<td>10</td>
<td>Matsushita</td>
<td>Matsushita</td>
<td>Hynudai</td>
<td>Philips</td>
<td>NEC</td>
<td>Broadcom*</td>
</tr>
</tbody>
</table>

Total Top 10 ($B)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13.9</td>
<td>31.8</td>
<td>86.3</td>
<td>108.1</td>
<td>118.2</td>
<td>167.1</td>
</tr>
</tbody>
</table>

Semi Market ($B)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>23.3</td>
<td>54.3</td>
<td>154</td>
<td>218.6</td>
<td>264.6</td>
<td>321.3</td>
</tr>
</tbody>
</table>

Top 10 % of Total Semi Mrkt

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>60%</td>
<td>59%</td>
<td>56%</td>
<td>49%</td>
<td>45%</td>
<td>52%</td>
</tr>
</tbody>
</table>

Source: IC Insights

*Fabless
From CPU to software, the 8080 Microcomputer is here.

Intel's new 8080 n-channel microcomputer is here—incridibly 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.

The 8080 microcomputer has 78 basic instructions, including the 8086 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- assembrer 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® CORE™2 DUO PROCESSOR, 40% MORE PERFORMANCE FOR BUSINESS.

Boosting 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

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

**ARM**
- **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
- **Endianness**: Big (little as default)

**RISC-V**
- **Designer**: University of California, Berkeley
- **Bits**: 32 · 64 · 128
- **Introduced**: 2010
- **Version**: unprivileged ISA 20191213, privileged ISA 20190606
- **Design**: RISC
- **Type**: Load-store
- **Encoding**: Variable
- **Branching**: Compare-and-branch
- **Endianness**: Little

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

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

**Architecture research, with some notable industry buy-in**
- **RISC-V ISA**
Exceedingly Dominant ISAs

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

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

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

**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

**ARM**
- **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)

**RISC-V**
- **Designer**: University of California, Berkeley
- **Bits**: 32 · 64 · 128
- **Introduced**: 2010
- **Version**: unprivileged ISA 20191213,\(^1\) privileged ISA 20190606\(^2\)
- **Design**: RISC
- **Type**: Load-store
- **Encoding**: Variable
- **Branching**: Compare-and-branch
- **Endianness**: Little\(^1\),\(^3\)
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.