Influence of Technology and Software on Instruction Sets: Up to the dawn of IBM 360

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Administrivia

• We’ve moved!
  – Lectures in 2-105
  – Recitations and quizzes in 37-212

• Second TA: Mark Seifter

• Self-assessment test due today
• Lab 0 due Wed
And then there was IBM 701

**IBM 701** -- 30 machines were sold in 1953-54

**IBM 650** -- a cheaper, drum based machine, more than 120 were sold in 1954 and there were orders for 750 more!
And then there was IBM 701

IBM 701 -- 30 machines were sold in 1953-54

IBM 650 -- a cheaper, drum based machine, more than 120 were sold in 1954 and there were orders for 750 more!

*Users stopped building their own machines.*
And then there was IBM 701

IBM 701 -- 30 machines were sold in 1953-54

IBM 650 -- a cheaper, drum based machine, more than 120 were sold in 1954 and there were orders for 750 more!

*Users stopped building their own machines.*

Why was IBM late getting into computers?
And then there was IBM 701

IBM 701 -- 30 machines were sold in 1953-54

IBM 650 -- a cheaper, drum based machine, more than 120 were sold in 1954 and there were orders for 750 more!

*Users stopped building their own machines.*

Why was IBM late getting into computers?

*IBM was making too much money!*

Even without computers, IBM revenues were doubling every 4 to 5 years in 40’s and 50’s.
Computers in mid 50’s
Computers in mid 50’s

• Hardware was expensive
Computers in mid 50’s

• Hardware was expensive
• Stores were small (1000 words)
Computers in mid 50’s

- Hardware was expensive
- Stores were small (1000 words)
  ⇒ No resident system-software!
Computers in mid 50’s

• Hardware was expensive
• Stores were small (1000 words)
  ⇒ No resident system-software!
• Memory access time was 10 to 50 times slower than the processor cycle
Computers in mid 50’s

- Hardware was expensive
- Stores were small (1000 words)
  ⇒ No resident system-software!
- Memory access time was 10 to 50 times slower than the processor cycle
  ⇒ Instruction execution time was totally dominated by the memory reference time.
Computers in mid 50’s

- Hardware was expensive
- Stores were small (1000 words)
  ⇒ No resident system-software!
- Memory access time was 10 to 50 times slower than the processor cycle
  ⇒ Instruction execution time was totally dominated by the memory reference time.
- The ability to design complex control circuits to execute an instruction was the central design concern as opposed to the speed of decoding or an ALU operation
Computers in mid 50’s

• Hardware was expensive

• Stores were small (1000 words)
  ⇒ No resident system-software!

• Memory access time was 10 to 50 times slower than the processor cycle
  ⇒ Instruction execution time was totally dominated by the memory reference time.

• The ability to design complex control circuits to execute an instruction was the central design concern as opposed to the speed of decoding or an ALU operation

• Programmer’s view of the machine was inseparable from the actual hardware implementation
Accumulator-based computing

- **Single Accumulator**
  - Calculator design carried over to computers
Accumulator-based computing

- **Single Accumulator**
  - Calculator design carried over to computers

*Why?*
Accumulator-based computing

- Single Accumulator
  - Calculator design carried over to computers

Why?

Registers expensive
The Earliest Instruction Sets

Burks, Goldstein & von Neumann  ~1946
# The Earliest Instruction Sets

*Burks, Goldstein & von Neumann ~1946*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>x</td>
<td>AC ← M[x]</td>
</tr>
<tr>
<td>STORE</td>
<td>x</td>
<td>M[x] ← (AC)</td>
</tr>
<tr>
<td>ADD</td>
<td>x</td>
<td>AC ← (AC) + M[x]</td>
</tr>
<tr>
<td>SUB</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>x</td>
<td>Involved a quotient register</td>
</tr>
<tr>
<td>DIV</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>SHIFT LEFT</td>
<td></td>
<td>AC ← 2 × (AC)</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### The Earliest Instruction Sets

*Burks, Goldstein & von Neumann ~1946*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>$x$</td>
<td>$AC \leftarrow M[x]$</td>
</tr>
<tr>
<td>STORE</td>
<td>$x$</td>
<td>$M[x] \leftarrow (AC)$</td>
</tr>
<tr>
<td>ADD</td>
<td>$x$</td>
<td>$AC \leftarrow (AC) + M[x]$</td>
</tr>
<tr>
<td>SUB</td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>$x$</td>
<td>Involved a quotient register</td>
</tr>
<tr>
<td>DIV</td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td>SHIFT LEFT</td>
<td></td>
<td>$AC \leftarrow 2 \times (AC)$</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUMP</td>
<td>$x$</td>
<td>$PC \leftarrow x$</td>
</tr>
<tr>
<td>JGE</td>
<td>$x$</td>
<td>if $(AC) \geq 0$ then $PC \leftarrow x$</td>
</tr>
</tbody>
</table>
# The Earliest Instruction Sets

*Burks, Goldstein & von Neumann ~1946*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>AC ← M[x]</td>
</tr>
<tr>
<td>STORE</td>
<td>M[x] ← (AC)</td>
</tr>
<tr>
<td>ADD</td>
<td>AC ← (AC) + M[x]</td>
</tr>
<tr>
<td>SUB</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>Involved a quotient register</td>
</tr>
<tr>
<td>DIV</td>
<td></td>
</tr>
<tr>
<td>SHIFT LEFT</td>
<td>AC ← 2 \times (AC)</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td></td>
</tr>
<tr>
<td>JUMP</td>
<td>PC ← x</td>
</tr>
<tr>
<td>JGE</td>
<td>if (AC) ≥ 0 then PC ← x</td>
</tr>
<tr>
<td>LOAD ADR</td>
<td>AC ← Extract address field(M[x])</td>
</tr>
<tr>
<td>STORE ADR</td>
<td></td>
</tr>
</tbody>
</table>
## The Earliest Instruction Sets

**Burks, Goldstein & von Neumann ~1946**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>AC ← M[x]</td>
</tr>
<tr>
<td>STORE</td>
<td>M[x] ← (AC)</td>
</tr>
<tr>
<td>ADD</td>
<td>AC ← (AC) + M[x]</td>
</tr>
<tr>
<td>SUB</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>Involved a quotient register</td>
</tr>
<tr>
<td>DIV</td>
<td></td>
</tr>
<tr>
<td>SHIFT LEFT</td>
<td>AC ← 2 × (AC)</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td></td>
</tr>
<tr>
<td>JUMP</td>
<td>PC ← x</td>
</tr>
<tr>
<td>JGE</td>
<td>if (AC) ≥ 0 then PC ← x</td>
</tr>
<tr>
<td>LOAD ADR</td>
<td>AC ← Extract address field(M[x])</td>
</tr>
<tr>
<td>STORE ADR</td>
<td></td>
</tr>
</tbody>
</table>

*Typically less than 2 dozen instructions!*
Programming: Single Accumulator Machine

\[ C_i \leftarrow A_i + B_i, \ 1 \leq i \leq n \]
Programming:
Single Accumulator Machine

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]
Programming: Single Accumulator Machine

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>JGE</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>F1</td>
<td>LOAD</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>F2</td>
<td>ADD</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>DONE</td>
<td>HLT</td>
<td></td>
<td>LOOP</td>
</tr>
</tbody>
</table>

Problem?

```
Problem?
```

February 10, 2014
Programming: Single Accumulator Machine

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

**Problem?**

How to modify the addresses A, B and C?
Self-Modifying Code

\[ C_i \leftarrow A_i + B_i, \ 1 \leq i \leq n \]

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGE</td>
<td>DONE</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ONE</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F1</th>
<th>LOAD</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>ADD</td>
<td>B</td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td>C</td>
</tr>
<tr>
<td>JUMP</td>
<td>LOOP</td>
<td></td>
</tr>
</tbody>
</table>

DONE   HLT
Self-Modifying Code

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGE</td>
<td>DONE</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ONE</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>LOAD</td>
<td>A</td>
</tr>
<tr>
<td>F2</td>
<td>ADD</td>
<td>B</td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td>C</td>
</tr>
<tr>
<td>JUMP</td>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td>DONE</td>
<td>HLT</td>
<td></td>
</tr>
</tbody>
</table>

$C_i \leftarrow A_i + B_i, \ 1 \leq i \leq n$

modify the program for the next iteration
Self-Modifying Code

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGE</td>
<td>DONE</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ONE</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

| F1 | LOAD | A |
| F2 | ADD | B |
| F3 | STORE | C |

modify the program for the next iteration

| LOAD ADR | F1 |
| ADD | ONE |
| STORE ADR | F1 |
| LOAD ADR | F2 |
| ADD | ONE |
| STORE ADR | F2 |
| LOAD ADR | F3 |
| ADD | ONE |
| STORE ADR | F3 |

| JUMP | LOOP |
| DONE | HLT |
Self-Modifying Code

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
<th>JGE</th>
<th>DONE</th>
<th>ADD</th>
<th>ONE</th>
<th>STORE</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>LOAD</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADD</td>
<td>B</td>
</tr>
<tr>
<td>F2</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STORE</td>
<td>C</td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LOAD ADR</td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STORE ADR</td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td>LOAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STORE ADR</td>
<td>F3</td>
</tr>
<tr>
<td></td>
<td>LOAD ADR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td>STORE ADR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JUMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HLT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ C_i \leftarrow A_i + B_i, \; 1 \leq i \leq n \]

Each iteration involves total bookkeeping:
- instruction fetches
- operand fetches
- stores

modify the program for the next iteration
Self-Modifying Code

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

Each iteration involves

- **total book-keeping**
- instruction fetches: 17
- operand fetches
- stores

modify the program for the next iteration
## Self-Modifying Code

Each iteration involves

<table>
<thead>
<tr>
<th>Total book-keeping</th>
<th>Instruction fetches</th>
<th>Operand fetches</th>
<th>Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

$$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$$

### Loop

<table>
<thead>
<tr>
<th>ACTION</th>
<th>DATA</th>
<th>DIRECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>N</td>
<td>STORE</td>
</tr>
<tr>
<td>ADD</td>
<td>N</td>
<td>STORE</td>
</tr>
<tr>
<td>STORE</td>
<td>ONE</td>
<td>STORE</td>
</tr>
</tbody>
</table>

**modify the program for the next iteration**

### F1

<table>
<thead>
<tr>
<th>ACTION</th>
<th>DATA</th>
<th>DIRECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>A</td>
<td>STORE</td>
</tr>
<tr>
<td>ADD</td>
<td>B</td>
<td>STORE</td>
</tr>
<tr>
<td>STORE</td>
<td>C</td>
<td>STORE</td>
</tr>
</tbody>
</table>

### F2

<table>
<thead>
<tr>
<th>ACTION</th>
<th>DATA</th>
<th>DIRECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>F1</td>
<td>STORE</td>
</tr>
<tr>
<td>ADD</td>
<td>ONE</td>
<td>STORE</td>
</tr>
<tr>
<td>STORE</td>
<td>F1</td>
<td>STORE</td>
</tr>
</tbody>
</table>

### F3

<table>
<thead>
<tr>
<th>ACTION</th>
<th>DATA</th>
<th>DIRECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>F2</td>
<td>STORE</td>
</tr>
<tr>
<td>ADD</td>
<td>ONE</td>
<td>STORE</td>
</tr>
<tr>
<td>STORE</td>
<td>F2</td>
<td>STORE</td>
</tr>
</tbody>
</table>

### Done

<table>
<thead>
<tr>
<th>ACTION</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP</td>
<td>F3</td>
</tr>
<tr>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td>DONE</td>
<td>HLT</td>
</tr>
</tbody>
</table>

February 10, 2014
Self-Modifying Code

\[
C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n
\]

Each iteration involves

- total book-keeping
- instruction fetches: 17
- operand fetches: 10
- stores: 5

modify the program for the next iteration

February 10, 2014
Self-Modifying Code

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JGE</td>
<td>DONE</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F1</th>
<th>LOAD</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>ADD</td>
<td>B</td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td>C</td>
</tr>
</tbody>
</table>

**modify the program for the next iteration**

| LOAD ADR | F1 |
| ADD  | ONE |
| STORE ADR | F1 |
| LOAD ADR | F2 |
| ADD  | ONE |
| STORE ADR | F2 |
| LOAD ADR | F3 |
| ADD  | ONE |
| STORE ADR | F3 |

| JUMP | LOOP |
| DONE | HLT |

Each iteration involves

| Instruction fetches | 17 | 14 |
| Operand fetches     | 10 |
| Stores              | 5  |
Self-Modifying Code

C_i ← A_i + B_i, 1 ≤ i ≤ n

Each iteration involves

- **total book-keeping**
- **instruction fetches**: 17 14
- **operand fetches**: 10 8
- **stores**: 5

Modify the program for the next iteration

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGE</td>
<td></td>
<td>DONE</td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td>ONE</td>
</tr>
<tr>
<td>STORE</td>
<td></td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F1</th>
<th>LOAD</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>STORE</td>
<td></td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F2</th>
<th>LOAD ADR</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td></td>
<td>ONE</td>
</tr>
<tr>
<td>STORE ADR</td>
<td></td>
<td>F1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F3</th>
<th>LOAD ADR</th>
<th>F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td></td>
<td>ONE</td>
</tr>
<tr>
<td>STORE ADR</td>
<td></td>
<td>F2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DONE</th>
<th>JUMP ADR</th>
<th>F3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLT</td>
<td></td>
<td>ONE</td>
</tr>
<tr>
<td>LOOP</td>
<td>STORE ADR</td>
<td>F3</td>
</tr>
</tbody>
</table>
## Self-Modifying Code

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load N</td>
<td>F1</td>
<td>LOAD N</td>
</tr>
<tr>
<td>Jump</td>
<td>F1</td>
<td>JUMP LOOP</td>
</tr>
<tr>
<td>Add A</td>
<td>F2</td>
<td>ADD A</td>
</tr>
<tr>
<td>Store N</td>
<td>F3</td>
<td>STORE N</td>
</tr>
<tr>
<td>Load ADR F1</td>
<td>F1</td>
<td>LOAD ADR F1</td>
</tr>
<tr>
<td>Add ONE</td>
<td>F3</td>
<td>ADD ONE</td>
</tr>
<tr>
<td>Store ADR F1</td>
<td>F1</td>
<td>STORE ADR F1</td>
</tr>
<tr>
<td>Load ADR F2</td>
<td>F2</td>
<td>LOAD ADR F2</td>
</tr>
<tr>
<td>Add A</td>
<td>F3</td>
<td>ADD A</td>
</tr>
<tr>
<td>Store ADR F2</td>
<td>F3</td>
<td>STORE ADR F3</td>
</tr>
<tr>
<td>Load ADR F3</td>
<td>F3</td>
<td>LOAD ADR F3</td>
</tr>
<tr>
<td>Add ONE</td>
<td>F3</td>
<td>ADD ONE</td>
</tr>
<tr>
<td>Store ADR F3</td>
<td>F3</td>
<td>STORE ADR F3</td>
</tr>
</tbody>
</table>

The code for each iteration involves:

- **Total Instruction Fetches**: 17
- **Total Operand Fetches**: 10
- **Total Stores**: 5

Each iteration modifies the program for the next iteration.

C<sub>i</sub> ← A<sub>i</sub> + B<sub>i</sub>, 1 ≤ i ≤ n
## Self-Modifying Code

Each iteration involves total book-keeping:

- **instruction fetches:**
  - F1: 17
  - F2: 10
  - F3: 8

- **operand fetches:**
  - F1: 14
  - F2: 8

- **stores:**
  - F1: 5
  - F2: 4

Most of the executed instructions are for book keeping!
Processor-Memory Bottleneck: Early Solutions

- Indexing capability

- Fast local storage in the processor
  - 8-16 registers as opposed to one accumulator

- Complex instructions

- Compact instructions
  - implicit address bits for operands
Processor-Memory Bottleneck: Early Solutions

- Indexing capability
  - to reduce book keeping instructions

- Fast local storage in the processor
  - 8-16 registers as opposed to one accumulator

- Complex instructions

- Compact instructions
  - implicit address bits for operands
Processor-Memory Bottleneck: Early Solutions

- Indexing capability
  - to reduce book keeping instructions

- Fast local storage in the processor
  - 8-16 registers as opposed to one accumulator
  - to save on loads/stores

- Complex instructions

- Compact instructions
  - implicit address bits for operands
Processor-Memory Bottleneck: Early Solutions

- Indexing capability
  - to reduce book keeping instructions

- Fast local storage in the processor
  - 8-16 registers as opposed to one accumulator
  - to save on loads/stores

- Complex instructions
  - to reduce instruction fetches

- Compact instructions
  - implicit address bits for operands
Processor-Memory Bottleneck: Early Solutions

- **Indexing capability**
  - to reduce book keeping instructions

- **Fast local storage in the processor**
  - 8-16 registers as opposed to one accumulator
  - to save on loads/stores

- **Complex instructions**
  - to reduce instruction fetches

- **Compact instructions**
  - implicit address bits for operands
  - to reduce instruction fetch cost
Index Registers
Tom Kilburn, Manchester University, mid 50’s

One or more specialized registers to simplify address calculation
Index Registers
Tom Kilburn, Manchester University, mid 50’s

One or more specialized registers to simplify address calculation

Modify existing instructions

LOAD   x, IX  AC ← M[x + (IX)]
ADD    x, IX  AC ← (AC) + M[x + (IX)]

...
Index Registers
Tom Kilburn, Manchester University, mid 50’s

One or more specialized registers to simplify address calculation

Modify existing instructions

LOAD x, IX \quad AC \leftarrow M[x + (IX)]
ADD x, IX \quad AC \leftarrow (AC) + M[x + (IX)]
...

Add new instructions to manipulate index registers

JZi x, IX \quad \text{if } (IX)=0 \text{ then } PC \leftarrow x
\text{ else } IX \leftarrow (IX) + 1
LOADi x, IX \quad IX \leftarrow M[x] \text{ (truncated to fit IX)}
...

February 10, 2014
Index Registers
Tom Kilburn, Manchester University, mid 50’s

One or more specialized registers to simplify address calculation

Modify existing instructions

```
LOAD x, IX  AC ← M[x + (IX)]
ADD x, IX  AC ← (AC) + M[x + (IX)]
...
```

Add new instructions to manipulate index registers

```
JZi x, IX  if (IX)=0 then  PC ← x
            else  IX ← (IX) + 1
LOADi x, IX  IX ← M[x]  (truncated to fit IX)
...
```

Index registers have accumulator-like characteristics
Using Index Registers

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOADI N, IX</th>
<th>JZi DONE, IX</th>
<th>LOAD LASTA, IX</th>
<th>ADD LASTB, IX</th>
<th>STORE LASTC, IX</th>
<th>JUMP LOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DONE</td>
<td>HALT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N starts with \(-n\)
Using Index Registers

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

- LOADi  N, IX
- LOOP JZi DONE, IX
- LOAD LASTA, IX
- ADD LASTB, IX
- STORE LASTC, IX
- JUMP LOOP

- \text{DONE} HALT

- Program does not modify itself

N starts with \(-n\)
Using Index Registers

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

- Program does not modify itself
- Efficiency has improved dramatically (ops / iter)

\[
\begin{align*}
\text{LOADi} & \quad N, \text{IX} \\
\text{LOOP} & \quad JZi \quad \text{DONE, IX} \\
\text{LOAD} & \quad \text{LASTA, IX} \\
\text{ADD} & \quad \text{LASTB, IX} \\
\text{STORE} & \quad \text{LASTC, IX} \\
\text{JUMP} & \quad \text{LOOP} \\
\text{DONE} & \quad \text{HALT}
\end{align*}
\]

N starts with -n
Using Index Registers

\[ C_i \leftarrow A_i + B_i, \ 1 \leq i \leq n \]

<table>
<thead>
<tr>
<th>LOAD</th>
<th>N, IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td>JZ</td>
<td>DONE, IX</td>
</tr>
<tr>
<td>LOAD</td>
<td>LASTA, IX</td>
</tr>
<tr>
<td>ADD</td>
<td>LASTB, IX</td>
</tr>
<tr>
<td>STORE</td>
<td>LASTC, IX</td>
</tr>
<tr>
<td>JUMP</td>
<td>LOOP</td>
</tr>
<tr>
<td>DONE</td>
<td>HALT</td>
</tr>
</tbody>
</table>

- Program does not modify itself
- Efficiency has improved dramatically (ops / iter) with index regs without index regs
  - instruction fetch
    - with index regs: 17 (14)
    - without index regs: 10 (8)
  - operand fetch
    - with index regs: 10 (8)
    - without index regs: 5 (4)
  - store
    - with index regs: 5 (4)
    - without index regs: 4 (4)
Using Index Registers

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

- **LOADi** \( N, IX \)
- **LOOP** \( JZi \) \( DONE, IX \)
- **LOAD** \( LASTA, IX \)
- **ADD** \( LASTB, IX \)
- **STORE** \( LASTC, IX \)
- **JUMP** \( LOOP \)
- **DONE** \( HALT \)

- **Program does not modify itself**
- **Efficiency has improved dramatically** \( (\text{ops} / \text{iter}) \) with index regs

<table>
<thead>
<tr>
<th></th>
<th>with index regs</th>
<th>without index regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction fetch</td>
<td>5 (2)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>operand fetch</td>
<td></td>
<td>10 (8)</td>
</tr>
<tr>
<td>store</td>
<td></td>
<td>5 (4)</td>
</tr>
</tbody>
</table>

N starts with \(-n\)

February 10, 2014
Using Index Registers

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

- **Program does not modify itself**
- **Efficiency has improved dramatically (ops / iter)**

<table>
<thead>
<tr>
<th></th>
<th>with index regs</th>
<th>without index regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction fetch</td>
<td>5 (2)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>operand fetch</td>
<td>2</td>
<td>10 (8)</td>
</tr>
<tr>
<td>store</td>
<td>2</td>
<td>5 (4)</td>
</tr>
</tbody>
</table>

N starts with \(-n\)

February 10, 2014
Using Index Registers

\[ C_i \leftarrow A_i + B_i, \ 1 \leq i \leq n \]

```
LOADi  N, IX
LOOP   JZi  DONE, IX
LOAD   LASTA, IX
ADD    LASTB, IX
STORE  LASTC, IX
JUMP   LOOP
DONE   HALT
```

- Program does not modify itself
- Efficiency has improved dramatically (ops / iter)
  
<table>
<thead>
<tr>
<th></th>
<th>with index regs</th>
<th>without index regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction fetch</td>
<td>5 (2)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>operand fetch</td>
<td>2</td>
<td>10 (8)</td>
</tr>
<tr>
<td>store</td>
<td>1</td>
<td>5 (4)</td>
</tr>
</tbody>
</table>

\( N \) starts with \(-n\)
Using Index Registers

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

- **Program does not modify itself**
- **Efficiency has improved dramatically (ops / iter)**
  - with index regs
  - without index regs
  
<table>
<thead>
<tr>
<th></th>
<th>with index regs</th>
<th>without index regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction fetch</td>
<td>5 (2)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>operand fetch</td>
<td>2</td>
<td>10 (8)</td>
</tr>
<tr>
<td>store</td>
<td>1</td>
<td>5 (4)</td>
</tr>
</tbody>
</table>

- **Costs?**

N starts with \(-n\)
Using Index Registers

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]

- **LOADi N, IX**
- **JZi DONE, IX**
- **LOAD LASTA, IX**
- **ADD LASTB, IX**
- **STORE LASTC, IX**
- **JUMP LOOP**
- **DONE HALT**

- **Program does not modify itself**
- **Efficiency has improved dramatically (ops / iter)**
  - **with index regs**
    - instruction fetch: 5 (2)
    - operand fetch: 2
    - store: 1
  - **without index regs**
    - instruction fetch: 17 (14)
    - operand fetch: 10 (8)
    - store: 5 (4)

- **Costs?**
  - Complex control
  - Index register computations (ALU-like circuitry)
  - 1 to 2 bits longer Instructions
Operations on Index Registers
Operations on Index Registers

To increment index register by $k$

\[
\begin{align*}
AC & \leftarrow (IX) \\
AC & \leftarrow (AC) + k \\
IX & \leftarrow (AC)
\end{align*}
\]

new instruction

new instruction
Operations on Index Registers

To increment index register by k

\[ AC \leftarrow (IX) \]
\[ AC \leftarrow (AC) + k \]
\[ IX \leftarrow (AC) \]

(new instruction)

(new instruction)

also the AC must be saved and restored
Operations on Index Registers

To increment index register by k

\[
\begin{align*}
AC & \leftarrow (IX) & \text{new instruction} \\
AC & \leftarrow (AC) + k \\
IX & \leftarrow (AC) & \text{new instruction}
\end{align*}
\]

also the AC must be saved and restored

It may be better to increment IX directly

\[
\text{INCI} \quad k, \text{IX} \quad IX \leftarrow (IX) + k
\]
Operations on Index Registers

To increment index register by \( k \)
- \( AC \leftarrow (IX) \)
- \( AC \leftarrow (AC) + k \)
- \( IX \leftarrow (AC) \)

new instruction

also the AC must be saved and restored

It may be better to increment IX directly

\( \text{INCl} \quad k, IX \quad IX \leftarrow (IX) + k \)

More instructions to manipulate index register

\( \text{STOREl} \quad x, IX \quad M[x] \leftarrow (IX) \) (extended to fit a word)

...
Operations on Index Registers

To increment index register by \( k \)

\[
\begin{align*}
AC &\leftarrow (IX) & \text{\textit{new instruction}} \\
AC &\leftarrow (AC) + k \\
IX &\leftarrow (AC) & \text{\textit{new instruction}}
\end{align*}
\]

also the AC must be saved and restored

It may be better to increment IX directly

\[
\text{INCI} \quad k, \text{ IX} \quad IX \leftarrow (IX) + k
\]

More instructions to manipulate index register

\[
\text{STOREi} \quad x, \text{ IX} \quad M[x] \leftarrow (IX) \quad \text{(extended to fit a word)}
\]

\( IX \) begins to look like an accumulator

\[ \Rightarrow \text{several index registers} \]
\[ \Rightarrow \text{several accumulators} \]

\[ \Rightarrow \text{General Purpose Registers} \]
Support for Subroutine Calls

Main Program

<table>
<thead>
<tr>
<th>call F</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
</tr>
<tr>
<td>a2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>call F</td>
</tr>
<tr>
<td>b1</td>
</tr>
<tr>
<td>b2</td>
</tr>
</tbody>
</table>

F:

|       |
|       |
|       |
| return |

Subroutine F
Support for Subroutine Calls

Main Program

<table>
<thead>
<tr>
<th>call F</th>
<th>a1</th>
<th>a2</th>
<th>.</th>
<th>.</th>
<th>call F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b1</td>
<td>b2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subroutine F

F:

. . .

return
Support for Subroutine Calls

Main Program

<table>
<thead>
<tr>
<th>call F</th>
<th>a1</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>call F</td>
<td>b1</td>
<td>b2</td>
</tr>
</tbody>
</table>

Subroutine F

F:

return
Support for Subroutine Calls

<table>
<thead>
<tr>
<th>Main Program</th>
<th>Subroutine F</th>
</tr>
</thead>
<tbody>
<tr>
<td>call F</td>
<td></td>
</tr>
<tr>
<td>a1</td>
<td></td>
</tr>
<tr>
<td>a2</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>call F</td>
<td></td>
</tr>
<tr>
<td>b1</td>
<td></td>
</tr>
<tr>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>F:</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>
Support for Subroutine Calls

Main Program

| call F |
| a1    |
| a2    |
| .     |
| .     |

Subroutine F

| call F |
| b1    |
| b2    |

F: return
Support for Subroutine Calls

A special subroutine jump instruction

A: JSR F

M[F] ← A + 1 and jump to F + 1
Indirect Addressing and Subroutine Calls

**Indirect addressing**
LOAD (x) means AC ← M[M[x]]
...

Figure:
- **Caller**
  - JSR F
  - arg
  - result
  - A
  - A+3

- **Subroutine**
  - F
  - F+1
  - S1: LOAD (F) inc F
  - S2: STORE(F) inc F
  - S3: JUMP (F)

*fetch arg*
*store result*

February 10, 2014
Indirect Addressing and Subroutine Calls

**Indirect addressing**

\[ \text{LOAD} \ (x) \ \text{means} \ AC \leftarrow M[M[x]] \]

Events:

- **S1** LOAD (F)
- **S2** STORE(F)
- **S3** JUMP (F)

Subroutine

- **F**
- **F+1**

Fetch

- **arg**
- **result**

Caller

- **A**
- **JSR**
- **F**
- **A+3**
- **arg**
- **result**

Indirect addressing

\[ \text{LOAD} \ (x) \ \text{means} \ AC \leftarrow M[M[x]] \]
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD (x) means AC ← M[M[x]]

... 

**Events:**

- **S1:** LOAD (F)
  - Execute A
  - inc F
  - F
  - F+1

- **S2:** STORE(F)
  - inc F
  - F
  - F+1

- **S3:** JUMP (F)

**Subroutine**

**fetch**

**arg**

**store**

**result**
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD \( (x) \) means \( AC \leftarrow M[M[x]] \)

...  

**Events:**

Execute A

**Subroutine**

fetch arg

store result

---

A

JSR F
arg
result

A+3

F
F+1
S1 LOAD (F)
inc F
S2 STORE(F)
inc F
S3 JUMP (F)
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD \( (x) \) means \( AC \leftarrow M[M[x]] \)

...  

---

**Events:**

Execute A

Execute S1

---

**Subroutine**

**fetch**

arg

**store**

result

---

February 10, 2014
Indirect Addressing and Subroutine Calls

**Indirect addressing**
LOAD (x) means AC ← M[M[x]]

Events:
- Execute A
- Execute S1

Subroutine
- LOAD (F)
- STORE(F)
- JUMP (F)

**Caller**
A
- JSR
- F
- arg
- result

A+3
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD \( (x) \) means \( AC \leftarrow M[M[x]] \)

...  

**Events:**

- Execute A
- Execute S1

---

**Subroutine**

Fetch

- arg

Store

- result

---

February 10, 2014
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD (x) means AC ← M[M[x]]

... 

**Events:**

Execute A
Execute S1
Execute S2

**Subroutine**

Fetch

arg

store

result

February 10, 2014
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD \((x)\) means \(AC \leftarrow M[M[x]]\)

... 

**Events:**

- Execute A
- Execute S1
- Execute S2

**Subroutine**

- **S1:** LOAD (F)
  - inc F
- **S2:** STORE(F)
  - inc F
- **S3:** JUMP (F)

February 10, 2014
Indirect Addressing and Subroutine Calls

Indirect addressing
LOAD (x) means AC ← M[M[x]]
...

Events:
- Execute A
- Execute S1
- Execute S2

Subroutine
F
F+1

S1
LOAD (F)
inc F

S2
STORE(F)
inc F

S3
JUMP (F)
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD (x) means AC ← M[M[x]]

... 

**Events:**

Execute A
Execute S1
Execute S2
Execute S3

**Subroutine**

S1: LOAD (F)  
inc F

S2: STORE(F)  
inc F

S3: JUMP (F)
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD (x) means AC ← M[M[x]]

... 

**Events:**

- Execute A
- Execute S1
- Execute S2
- Execute S3

Indirect addressing almost eliminates the need to write self-modifying code (location F still needs to be modified)
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD (x) means AC ← M[M[x]]

...  

**Events:**

execute A
execute S1
execute S2
execute S3

Indirect addressing almost eliminates the need to write self-modifying code (location F still needs to be modified)

Problems? ⇒
**Indirect Addressing and Subroutine Calls**

**Indirect addressing**

$\text{LOAD} \ (x) \ \text{means} \ AC \leftarrow M[M[x]]$

... 

Events:

- Execute A
- Execute S1
- Execute S2
- Execute S3

Indirect addressing almost eliminates the need to write self-modifying code (location F still needs to be modified)

**Problems? ⇒ recursive procedure calls**
Recursive Procedure Calls and Reentrant Codes

*Indirect Addressing through a register*

\[
\text{LOAD } R_1, (R_2)
\]

Load register \( R_1 \) with the contents of the word whose address is contained in register \( R_2 \)
Evolution of Addressing Modes
Evolution of Addressing Modes

1. Single accumulator, absolute address
   LOAD x
Evolution of Addressing Modes

1. Single accumulator, absolute address
   LOAD x

2. Single accumulator, index registers
   LOAD x, IX
Evolution of Addressing Modes

1. Single accumulator, absolute address
   - LOAD x

2. Single accumulator, index registers
   - LOAD x, IX

3. Indirection
   - LOAD (x)
Evolution of Addressing Modes

1. Single accumulator, absolute address
   \[\text{LOAD } x\]

2. Single accumulator, index registers
   \[\text{LOAD } x, \text{IX}\]

3. Indirection
   \[\text{LOAD } (x)\]

4. Multiple accumulators, index registers, indirection
   \[\text{LOAD } R, \text{IX}, x\]
   \[\text{or } \text{LOAD } R, \text{IX}, (x)\]
   \[\text{the meaning?}\]
   \[R \leftarrow M[M[x] + (\text{IX})]\]
   \[\text{or } R \leftarrow M[M[x + (\text{IX})]]\]
Evolution of Addressing Modes

1. Single accumulator, absolute address
   \[ \text{LOAD } x \]

2. Single accumulator, index registers
   \[ \text{LOAD } x, \text{IX} \]

3. Indirection
   \[ \text{LOAD } (x) \]

4. Multiple accumulators, index registers, indirection
   \[ \text{LOAD } R, \text{IX}, x \]
   or \[ \text{LOAD } R, \text{IX}, (x) \]
   the meaning?
   \[ R \leftarrow M[M[x] + (\text{IX})] \]
   or \[ R \leftarrow M[M[x + (\text{IX})]] \]

5. Indirect through registers
   \[ \text{LOAD } R_I, (R_J) \]
Evolution of Addressing Modes

1. Single accumulator, absolute address
   \[ \text{LOAD } x \]

2. Single accumulator, index registers
   \[ \text{LOAD } x, \text{IX} \]

3. Indirection
   \[ \text{LOAD } (x) \]

4. Multiple accumulators, index registers, indirection
   \[ \text{LOAD } R, \text{IX}, x \]
   or \[ \text{LOAD } R, \text{IX}, (x) \]

5. Indirect through registers
   \[ \text{LOAD } R_\text{I}, (R_\text{J}) \]

6. The works
   \[ \text{LOAD } R_\text{I}, R_\text{J}, (R_\text{K}) \]
   \[ R_\text{J} = \text{index}, \ R_\text{K} = \text{base addr} \]
Variety of Instruction Formats
Variety of Instruction Formats

- *Three address formats:* One destination and up to two operand sources per instruction

  \[
  (\text{Reg op Reg}) \text{ to Reg} \quad R_I \leftarrow (R_J) \text{ op } (R_K) \\
  (\text{Reg op Mem}) \text{ to Reg} \quad R_I \leftarrow (R_J) \text{ op } M[x]
  \]

  - \(x\) can be specified directly or via a register
  - effective address calculation for \(x\) could include indexing, indirection, ...
Variety of Instruction Formats

- **Three address formats**: One destination and up to two operand sources per instruction

  \[
  \begin{align*}
  \text{(Reg op Reg) to Reg} & \quad R_I \leftarrow (R_J) \text{ op } (R_K) \\
  \text{(Reg op Mem) to Reg} & \quad R_I \leftarrow (R_J) \text{ op } M[x]
  \end{align*}
  \]

  - x can be specified directly or via a register
  - effective address calculation for x could include indexing, indirection, ...

- **Two address formats**: the destination is same as one of the operand sources

  \[
  \begin{align*}
  \text{(Reg op Reg) to Reg} & \quad R_I \leftarrow (R_I) \text{ op } (R_J) \\
  \text{(Reg op Mem) to Reg} & \quad R_I \leftarrow (R_I) \text{ op } M[x]
  \end{align*}
  \]
More Instruction Formats
More Instruction Formats

• *One address formats:* Accumulator machines
  – Accumulator is always other implicit operand
More Instruction Formats

• **One address formats:** Accumulator machines
  - Accumulator is always other implicit operand

• **Zero address formats:** operands on a stack
  
  \[
  \text{add} \quad M[sp-1] \leftarrow M[sp] + M[sp-1]
  \]
  
  \[
  \text{load} \quad M[sp] \leftarrow M[M[sp]]
  \]

  - Stack can be in registers or in memory
    - usually top of stack cached in registers
More Instruction Formats

- **One address formats:** Accumulator machines
  - Accumulator is always other implicit operand

- **Zero address formats:** operands on a stack
  - Stack can be in registers or in memory
  - Usually top of stack cached in registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td></td>
</tr>
</tbody>
</table>

- add $M[sp-1] \leftarrow M[sp] + M[sp-1]$
- load $M[sp] \leftarrow M[M[sp]]$
More Instruction Formats

- **One address formats:** Accumulator machines
  - Accumulator is always other implicit operand

- **Zero address formats:** operands on a stack

  \[
  \begin{align*}
  \text{add} & \quad M[sp-1] \leftarrow M[sp] + M[sp-1] \\
  \text{load} & \quad M[sp] \leftarrow M[M[sp]]
  \end{align*}
  \]

  - Stack can be in registers or in memory
    - usually top of stack cached in registers

  *Many different formats are possible!*
Data Formats and Memory Addresses

Data formats:

- Bytes, Half words, words and double words

Some issues

- **Byte addressing**
  
<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs. Little Endian</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Word alignment**
  
  Suppose the memory is organized in 32-bit words. Can a word address begin only at 0, 4, 8, .... ?

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Some Tradeoffs

• Should all addressing modes be provided for every operand?
  \[ \Rightarrow \text{regular vs. irregular instruction formats} \]

• Separate instructions to manipulate Accumulators, Index registers, Base registers
  \[ \Rightarrow \text{large number of instructions} \]

• Instructions contained implicit memory references -- several contained more than one
  \[ \Rightarrow \text{very complex control} \]
Some Tradeoffs

• Should all addressing modes be provided for every operand?

  ⇒ regular vs. irregular instruction formats

• Separate instructions to manipulate Accumulators, Index registers, Base registers

  ⇒ large number of instructions

• Instructions contained implicit memory references -- several contained more than one

  ⇒ very complex control

Great variety of instruction sets
The first definition of the Instruction Set Abstraction: IBM 360
The IBM 650 (1953-4)

Magnetic Drum (1,000 or 2,000 10-digit decimal words)

Active instruction (including next program counter)

Digit-serial ALU

20-digit accumulator

[From 650 Manual, © IBM]
Programmer’s view of a machine: IBM 650

A drum machine with 44 instructions

Instruction: 60 1234 1009
- “Load the contents of location 1234 into the distribution; put it also into the upper accumulator; set lower accumulator to zero; and then go to location 1009 for the next instruction.”
Programmer’s view of a machine: IBM 650

A drum machine with 44 instructions

Instruction: 60 1234 1009
- “Load the contents of location 1234 into the distribution; put it also into the upper accumulator; set lower accumulator to zero; and then go to location 1009 for the next instruction.”

- Programmer’s view of the machine was inseparable from the actual hardware implementation
A drum machine with 44 instructions

Instruction: 60 1234 1009
- “Load the contents of location 1234 into the distribution; put it also into the upper accumulator; set lower accumulator to zero; and then go to location 1009 for the next instruction.”

- Programmer’s view of the machine was inseparable from the actual hardware implementation

- Good programmers optimized the placement of instructions on the drum to reduce latency!
Compatibility Problem at IBM
Compatibility Problem at IBM

By early 60’s, IBM had 4 incompatible lines of computers!

701 → 7094
650 → 7074
702 → 7080
1401 → 7010
Compatibility Problem at IBM

By early 60’s, *IBM had 4 incompatible lines of computers!*

<table>
<thead>
<tr>
<th>701</th>
<th>→</th>
<th>7094</th>
</tr>
</thead>
<tbody>
<tr>
<td>650</td>
<td>→</td>
<td>7074</td>
</tr>
<tr>
<td>702</td>
<td>→</td>
<td>7080</td>
</tr>
<tr>
<td>1401</td>
<td>→</td>
<td>7010</td>
</tr>
</tbody>
</table>

Each system had its own

- Instruction set
- I/O system and Secondary Storage: magnetic tapes, drums and disks
- assemblers, compilers, libraries,...
- market niche
  - business, scientific, real time, ...
Compatibility Problem at IBM

By early 60’s, *IBM had 4 incompatible lines of computers!*

701 → 7094  
650 → 7074  
702 → 7080  
1401 → 7010

Each system had its own
- Instruction set
- I/O system and Secondary Storage: magnetic tapes, drums and disks
- assemblers, compilers, libraries,...
- market niche  
  business, scientific, real time, ...

⇒ *IBM 360*
IBM 360 : Design Premises
Amdahl, Blaauw and Brooks, 1964

The design must lend itself to growth and successor machines

- General method for connecting I/O devices
- Total performance - answers per month rather than bits per microsecond \( \Rightarrow \) programming aids
- Machine must be capable of supervising itself without manual intervention
- Built-in hardware fault checking and locating aids to reduce down time
- Simple to assemble systems with redundant I/O devices, memories etc. for fault tolerance
- Some problems required floating point words larger than 36 bits
Processor State and Data Types

The information held in the processor at the end of an instruction to provide the processing context for the next instruction.
Processor State and Data Types

The information held in the processor at the end of an instruction to provide the processing context for the next instruction.

Program Counter, Accumulator, . . .
The information held in the processor at the end of an instruction to provide the processing context for the next instruction.

Program Counter, Accumulator, . . .

- The information held in the processor will be interpreted as having data types manipulated by the instructions.
Processor State and Data Types

The information held in the processor at the end of an instruction to provide the processing context for the next instruction.

Program Counter, Accumulator, . . .

- The information held in the processor will be interpreted as having data types manipulated by the instructions.

- If the processing of an instruction can be interrupted then the hardware must save and restore the state in a transparent manner.
Processor State and Data Types

The information held in the processor at the end of an instruction to provide the processing context for the next instruction.

Program Counter, Accumulator, . . .

- The information held in the processor will be interpreted as having data types manipulated by the instructions.

- If the processing of an instruction can be interrupted then the hardware must save and restore the state in a transparent manner.

*Programmer’s machine model* is a contract between the hardware and software.
Instruction set

The control for changing the information held in the processor are specified by the instructions available in the instruction set architecture or ISA.
Instruction set

The control for changing the information held in the processor are specified by the instructions available in the instruction set architecture or ISA.

Some things an ISA must specify:
Instruction set

The control for changing the information held in the processor are specified by the instructions available in the instruction set architecture or ISA.

Some things an ISA must specify:
- *A way to reference registers and memory*
Instruction set

The control for changing the information held in the processor are specified by the instructions available in the instruction set architecture or ISA.

Some things an ISA must specify:

- A way to reference registers and memory
- The computational operations available
Instruction set

The control for changing the information held in the processor are specified by the instructions available in the instruction set architecture or ISA.

Some things an ISA must specify:

- A way to reference registers and memory
- The computational operations available
- How to control the sequence of instructions
Instruction set

The control for changing the information held in the processor are specified by the instructions available in the instruction set architecture or ISA.

Some things an ISA must specify:

- A way to reference registers and memory
- The computational operations available
- How to control the sequence of instructions
- A binary representation for all of the above
Instruction set

The control for changing the information held in the processor are specified by the instructions available in the instruction set architecture or ISA.

Some things an ISA must specify:

- A way to reference registers and memory
- The computational operations available
- How to control the sequence of instructions

- A binary representation for all of the above

ISA must satisfy the needs of the software:
- assembler, compiler, OS, VM
IBM 360: A General-Purpose Register (GPR) Machine

- Processor State
  - 16 General-Purpose 32-bit Registers
    - may be used as index and base register
    - Register 0 has some special properties
  - 4 Floating Point 64-bit Registers
  - A Program Status Word (PSW)
    - PC, Condition codes, Control flags
IBM 360: A General-Purpose Register (GPR) Machine

- **Processor State**
  - 16 General-Purpose 32-bit Registers
    - *may be used as index and base register*
    - *Register 0 has some special properties*
  - 4 Floating Point 64-bit Registers
  - A Program Status Word (PSW)
    - *PC, Condition codes, Control flags*

- **Data Formats**
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words
  - 24-bit addresses
IBM 360: A General-Purpose Register (GPR) Machine

• Processor State
  – 16 General-Purpose 32-bit Registers
    • may be used as index and base register
    • Register 0 has some special properties
  – 4 Floating Point 64-bit Registers
  – A Program Status Word (PSW)
    • PC, Condition codes, Control flags

• Data Formats
  – 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words
  – 24-bit addresses

• A 32-bit machine with 24-bit addresses
IBM 360: A General-Purpose Register (GPR) Machine

• Processor State
  – 16 General-Purpose 32-bit Registers
    • may be used as index and base register
    • Register 0 has some special properties
  – 4 Floating Point 64-bit Registers
  – A Program Status Word (PSW)
    • PC, Condition codes, Control flags

• Data Formats
  – 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words
  – 24-bit addresses

• A 32-bit machine with 24-bit addresses
  – No instruction contains a 24-bit address!
IBM 360: Some Addressing Modes

RR

<table>
<thead>
<tr>
<th>8</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>R1</td>
<td>R2</td>
</tr>
</tbody>
</table>

R1 ← (R1) op (R2)
IBM 360: Some Addressing Modes

RR

<table>
<thead>
<tr>
<th>8</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>R1</td>
<td>R2</td>
</tr>
</tbody>
</table>

R1\to(R1) \text{ op } (R2)

RD

<table>
<thead>
<tr>
<th>8</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>R</td>
<td>X</td>
<td>B</td>
<td>D</td>
</tr>
</tbody>
</table>
**IBM 360: Some Addressing Modes**

A 24-bit address is formed by adding the 12-bit displacement (D) to a base register (B) and an Index register (X), if desired.

### RR
- **opcode**
- **R1**
- **R2**

\[ R1 \leftarrow (R1) \text{ op } (R2) \]

### RD
- **opcode**
- **R**
- **X**
- **B**
- **D**

\[ R \leftarrow (R) \text{ op } M[(X) + (B) + D] \]
IBM 360: Some Addressing Modes

<table>
<thead>
<tr>
<th>RR</th>
<th>8</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>R1</td>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>R1 ← (R1) op (R2)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RD</th>
<th>8</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>R</td>
<td>X</td>
<td>B</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

R ← (R) op M[(X) + (B) + D]  
a 24-bit address is formed by adding the  
12-bit displacement (D) to a base register (B)  
and an Index register (X), if desired

The most common formats for arithmetic & logic instructions, as well as Load and Store instructions
### IBM 360: Character String Operations

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>4</td>
<td>12</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>opcode</td>
<td>length</td>
<td>B1</td>
<td>D1</td>
<td>B2</td>
<td>D2</td>
</tr>
</tbody>
</table>

**SS format: store to store instructions**

\[
M[(B1) + D1] \leftarrow M[(B1) + D1] \text{ op } M[(B2) + D2]
\]

iterate “length” times
IBM 360: Character String Operations

SS format: store to store instructions

\[
M[(B1) + D1] \leftarrow M[(B1) + D1] \text{ op } M[(B2) + D2]
\]
iterate “length” times

Most operations on decimal and character strings use this format

- MVC move characters
- MP multiply two packed decimal strings
- CLC compare two character strings
- ...

February 10, 2014
**IBM 360: Character String Operations**

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>8</th>
<th>4</th>
<th>12</th>
<th>4</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SS format: store to store instructions**

\[
M[(B1) + D1] \leftarrow M[(B1) + D1] \text{ op } M[(B2) + D2]
\]

iterate “length” times

**Most operations on decimal and character strings use this format**

- MVC     move characters
- MP      multiply two packed decimal strings
- CLC     compare two character strings

... 

**Multiple memory operations per instruction complicates exception & interrupt handling**
IBM 360: Branches & Condition Codes

- Arithmetic and logic instructions set *condition codes*
  - equal to zero
  - greater than zero
  - overflow
  - carry...

- I/O instructions also set condition codes
  - channel busy

- Conditional branch instructions are based on testing condition code registers (CC’s)
  - RX and RR formats
    - BC_ branch conditionally
    - BAL_ branch and link, i.e., R15 ← (PC)+1
      *for subroutine calls*
      ⇒ CC’s must be part of the PSW
IBM 360: Precise Interrupts

- IBM 360 ISA (Instruction Set Architecture) preserves sequential execution model

- Programmers view of machine was that each instruction either completed or signaled a fault before the next instruction began execution

- Exception/interrupt behavior identical across family of implementations
## IBM 360: Initial Implementations (1964)

<table>
<thead>
<tr>
<th></th>
<th>Model 30</th>
<th>...</th>
<th>Model 70</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory Capacity</strong></td>
<td>8K - 64 KB</td>
<td></td>
<td>256K - 512 KB</td>
</tr>
<tr>
<td><strong>Memory Cycle</strong></td>
<td>2.0µs</td>
<td>...</td>
<td>1.0µs</td>
</tr>
<tr>
<td><strong>Datapath</strong></td>
<td>8-bit</td>
<td></td>
<td>64-bit</td>
</tr>
<tr>
<td><strong>Circuit Delay</strong></td>
<td>30 nsec/level</td>
<td></td>
<td>5 nsec/level</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>in Main Store</td>
<td></td>
<td>in Transistor</td>
</tr>
<tr>
<td><strong>Control Store</strong></td>
<td>Read only 1µsec</td>
<td></td>
<td>Dedicated circuits</td>
</tr>
</tbody>
</table>

- Six implementations (Models, 30, 40, 50, 60, 62, 70)
- 50X performance difference cross models
- *ISA completely hid the underlying technological differences between various models.*

With minor modifications, IBM 360 ISA is still in use
IBM 360: Forty-Six years later... zEnterprise196 Microprocessor

- 1.4 billion transistors, Quad core design
- Up to 96 cores (80 visible to OS) in one multichip module
- 5.2 GHz, IBM 45nm SOI CMOS technology
- 64-bit virtual addressing
  - original 360 was 24-bit; 370 was a 31-bit extension
- Superscalar, out-of-order
  - Up to 72 instructions in flight
- Variable length instruction pipeline: 15-17 stages
- Each core has 2 integer units, 2 load-store units and 2 floating point units
- 8K-entry Branch Target Buffer
  - Very large buffer to support commercial workload
- Four Levels of caches:
  - 64KB L1 I-cache, 128KB L1 D-cache
  - 1.5MB L2 cache per core
  - 24MB shared on-chip L3 cache
  - 192MB shared off-chip L4 cache

[ September 2010 ]

February 10, 2014
Sanchez & Emer
Instruction Set Architecture (ISA) versus Implementation

- ISA is the hardware/software interface
  - Defines set of programmer visible state
  - Defines data types
  - Defines instruction semantics (operations, sequencing)
  - Defines instruction format (bit encoding)
  - Examples: MIPS, Alpha, x86, IBM 360, VAX, ARM, JVM
Instruction Set Architecture (ISA) versus Implementation

• ISA is the hardware/software interface
  - Defines set of programmer visible state
  - Defines data types
  - Defines instruction semantics (operations, sequencing)
  - Defines instruction format (bit encoding)
  - Examples: MIPS, Alpha, x86, IBM 360, VAX, ARM, JVM

• Many possible implementations of one ISA
  - 360 implementations: model 30 (c. 1964), zEnterprise196 (c. 2010)
  - x86 implementations: 8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4, Core i7, AMD Athlon, AMD Opteron, Transmeta Crusoe, SoftPC
  - MIPS implementations: R2000, R4000, R10000, ...
  - JVM: HotSpot, PicoJava, ARM Jazelle, ...
Next lecture:
Implementing an ISA