“I think there is a world market for maybe five computers”
Thomas Watson, chairman of IBM, 1943

1. **Indexed** addressing mode
   - fixed offset with the 16-bit registers: X, Y, SP, or PC
   - 5-bit (-16 to +15). -> write as 5-bit 2’s complement integer
   - 9-bit (-256 to +127), -> write as 9-bit 2’s complement integer
   - 16-bit unsigned (0 to 65535) or signed (-32768 to +32767.)

<table>
<thead>
<tr>
<th>postbyte, xb</th>
<th>syntax</th>
<th>mode</th>
<th>explanations</th>
</tr>
</thead>
<tbody>
<tr>
<td>rr000000</td>
<td>,r</td>
<td>IDX</td>
<td>5-bit constant, n=0</td>
</tr>
<tr>
<td>rr00nnnn</td>
<td>n,r</td>
<td>IDX</td>
<td>5-bit constant, n=0 to +15</td>
</tr>
<tr>
<td>rr01nnnn</td>
<td>-n,r</td>
<td>IDX</td>
<td>5-bit constant, n=-16 to -1</td>
</tr>
<tr>
<td>rr100nnn</td>
<td>n,+r</td>
<td>IDX</td>
<td>pre-increment, n=1 to 8</td>
</tr>
<tr>
<td>rr101nnn</td>
<td>n,-r</td>
<td>IDX</td>
<td>pre-decrement, n=1 to 8</td>
</tr>
<tr>
<td>rr110nnn</td>
<td>n,r+</td>
<td>IDX</td>
<td>post-increment, n=1 to 8</td>
</tr>
<tr>
<td>rr111nnn</td>
<td>n,r-</td>
<td>IDX</td>
<td>post-decrement, n=1 to 8</td>
</tr>
<tr>
<td>111rr000 fff</td>
<td>n,r</td>
<td>IDX1</td>
<td>9-bit cons, n 16 to 255</td>
</tr>
<tr>
<td>111rr001 fff</td>
<td>-n,r</td>
<td>IDX1</td>
<td>9-bit const, n -256 to -16</td>
</tr>
<tr>
<td>111rr010 ffee</td>
<td>n,r</td>
<td>IDX2</td>
<td>16-bit const, any 16-bit n</td>
</tr>
<tr>
<td>111rr111</td>
<td>[D,r]</td>
<td>[D,IDX]</td>
<td>Reg D offset, indirect</td>
</tr>
<tr>
<td>111rr011 ffee</td>
<td>[n,r]</td>
<td>[IDX2]</td>
<td>16-bit constant, indirect</td>
</tr>
</tbody>
</table>
Table 3.24. Postbyte values for the indexed-addressing modes.

Table A-3 Indexed Addressing Mode Postbyte Encoding (xb)

<table>
<thead>
<tr>
<th>Machine Code</th>
<th>Opcode</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A6E99C</td>
<td>ldaa</td>
<td>$-100,Y</td>
<td>$[Y-100] = RegA</td>
</tr>
</tbody>
</table>

3.3. Introduction to programming

3.3.1. Memory and register transfer operations

The 8-bit load instructions

```
ldaa #w RegA=w
ldaa U RegA=[U]
```
ldab  #w  RegB=w
ldab  U   RegB=[U]

Condition code bits are set
N: result is negative N=R7
Z: result is zero
V: signed overflow V=0

The 16-bit load instructions
ingga  #W   RegD=W
ingga  U   RegD={U}
ingga  #W   RegSP=W
ingga  U   RegSP={U}
ingga  #W   RegX=W
ingga  U   RegX={U}
ingga  #W   RegY=W
ingga  U   RegY={U}

Condition code bits are set
N: result is negative N=R15
Z: result is zero
V: signed overflow V=0

The memory to memory move instructions, set no flags.
movb  #w,addr        [addr]=w
movb  addr1,addr2   [addr2]=[addr1]
movw  #W,addr       {addr}=W
movw  addr1,addr2   {addr2}={addr1}

The 8-bit store instructions
staa  U       [U]=RegA
\textbf{stab $U$ \quad $[U]=\text{RegB}$}

Condition code bits are set
\begin{itemize}
\item N: result is negative $N=R_7$
\item Z: result is zero
\item V: signed overflow $V=0$
\end{itemize}

\textbf{The 16-bit store instructions}
\begin{itemize}
\item \textbf{std} $U$ \quad \{U\}=\text{RegD}
\item \textbf{sts} $U$ \quad \{U\}=\text{RegSP}
\item \textbf{stx} $U$ \quad \{U\}=\text{RegX}
\item \textbf{sty} $U$ \quad \{U\}=\text{RegY}
\end{itemize}

Condition code bits are set
\begin{itemize}
\item N: result is negative $N=R_{15}$
\item Z: result is zero
\item V: signed overflow $V=0$
\end{itemize}

\textbf{The transfer operations}
\begin{itemize}
\item \textbf{xgdx} \quad \text{swap RegD and RegX}
\item \textbf{xgdy} \quad \text{swap RegD and RegY}
\item \textbf{clc} \quad \text{clear carry bit, } C=0
\item \textbf{cli} \quad \text{enable interrupts, } I=0
\item \textbf{clv} \quad \text{clear overflow bit, } V=0
\item \textbf{sec} \quad \text{set carry bit, } C=1
\item \textbf{sei} \quad \text{disable interrupts, } I=1
\item \textbf{sev} \quad \text{set overflow bit, } V=1
\item \textbf{tap} \quad \text{transfer A to CC}
\item \textbf{tpa} \quad \text{transfer CC to A}
\end{itemize}

\textbf{3.3.2. Arithmetic operations}

\textbf{8-bit addition}
These instructions work for both signed and unsigned data.
adda #w \quad \text{RegA=RegA+w}
adda U \quad \text{RegA=RegA+[U]}
addb #w \quad \text{RegB=RegB+w}
addb U \quad \text{RegB=RegB+[U]}

Condition code bits are set after R=X+M,
N: result is negative N=R7
Z: result is zero
V: signed overflow
V=X7\cdot M7\cdot \text{not}(R7)+\text{not}(X7)\cdot \text{not}(M7)\cdot R7
C: unsigned overflow
C=X7\cdot M7+M7\cdot \text{not}(R7)+\text{not}(R7)\cdot X7

16-bit addition
These instructions work for both signed and unsigned data.
addd #W \quad \text{RegD=RegD+W}
addd U \quad \text{RegD=RegD+[U]}

Condition code bits are set after R=D+M.
N: result is negative N=R15
Z: result is zero
V: signed overflow
V=D15\cdot M15\cdot \text{not}(R15)+\text{not}(D15)\cdot \text{not}(M15)\cdot R15
C: unsigned overflow
C=D15\cdot M15+M15\cdot \text{not}(R15)+\text{not}(R15)\cdot D15

8-bit subtraction
These instructions work for both signed and unsigned data.
cmpa #w \quad \text{RegA=w}
cmpa U \quad \text{RegA=[U]}
cmpb #w \quad \text{RegB=w}
cmpb U \quad \text{RegB=[U]}
suba #w      RegA=RegA−w
suba U       RegA=RegA−[U]
subb #w      RegB=RegB−w
subb U       RegB=RegB−[U]
tsta         RegA=0
 tstb         RegB=0

Condition code bits are set after R=X-M
N: result is negative N=R7
Z: result is zero
V: signed overflow
V=X7•not(M7)•not(R7)+not(X7)•M7•R7
C: unsigned overflow
C=not(X7)•M7+M7•R7+R7•not(X7)

16-bit subtraction
These instructions work for both signed and unsigned data.
cpd  #W      RegD−W
cpd  U       RegD−{U}
cpx  #W      RegX−W
cpx  U       RegX−{U}
cpy  #W      RegY−W
cpy  U       RegY−{U}
subd #W      RegD=RegD−W
subd U       RegD=RegD−{U}

Condition code bits are set after R=X-M
N: result is negative N=R15
Z: result is zero
V: signed overflow
V=X15•not(M15)•not(R15)+not(X15)•M15•R15
C: unsigned overflow
\[ C = \neg(X_{15}) \cdot M_{15} + M_{15} \cdot R_{15} + R_{15} \cdot \neg(X_{15}) \]

The increment and decrement instructions,
These instructions work for both signed and unsigned data.
The Z bit is set if the result is zero.

\[
\begin{align*}
\text{deca} & \quad \text{RegA} = \text{RegA} - 1 \\
\text{decb} & \quad \text{RegA} = \text{RegA} - 1 \\
\text{dex} & \quad \text{RegX} = \text{RegX} - 1 \\
\text{dey} & \quad \text{RegY} = \text{RegY} - 1 \\
\text{inca} & \quad \text{RegA} = \text{RegA} + 1 \\
\text{incb} & \quad \text{RegB} = \text{RegB} + 1 \\
\text{inx} & \quad \text{RegX} = \text{RegX} + 1 \\
\text{iny} & \quad \text{RegY} = \text{RegY} + 1
\end{align*}
\]

\textbf{mul} an 8-bit by 8-bit into 16-bit unsigned multiply

\[
\begin{array}{c}
\text{Register A} \times \text{Register B} = \text{Register D} \\
\text{8 bits} \times \text{8 bits} = \text{16 bits}
\end{array}
\]

Figure 3.13. The \texttt{mul} instruction takes two 8-bit inputs and generates a 16-bit product.

Condition code bits are set after \( R = A \times B \).

C: R7, set if bit 7 of 16-bit result is one

\textbf{Checkpoint 3.13:} Prove the \texttt{mul} instruction can’t overflow.

Let \( N \) and \( M \) be 8-bit unsigned locations. \( M = 3 \times N \).
ldaa N
ldab #3
mul RegD=3*N
cmpa #0
beq ok
ldab #255 overflow
OK stab M

'idiv' performs 16-bit by 16-bit unsigned divide

\[
\begin{array}{ccc}
\text{Register D} & \div & \text{Register X} \\
\end{array} = \begin{array}{c}
\text{Register X} \\
\text{remainder} = \text{Register D} \\
\end{array}
\]

Figure 3.14. The 'idiv' instruction takes two 16-bit inputs and generates a 16-bit quotient and a 16-bit remainder.

Condition code bits are set after quotient=dividend/divisor.
- Z: result is zero,
- V: 0
- C: divide by zero,
  \[C = \text{not}(X_{15}) \cdot \text{not}(X_{14}) \cdot \ldots \cdot \text{not}(X_{2}) \cdot \text{not}(X_{1}) \cdot \text{not}(X_{0})\]

Let N and M be 16-bit unsigned locations. M=N/15.

\[
\begin{array}{c}
\text{ldd } N \\
\text{ldx } #15 \\
idiv X = N/15 \\
\text{stx } M
\end{array} \quad \text{(between 0 and 65535)} \quad \text{(between 0 and 4369)}
\]

Checkpoint 3.14: Give a single mathematical equation relating the dividend, divisor, quotient, and remainder.
This equation gives a unique solution as long as you assume the remainder is strictly less than the divisor.

\[ \frac{\text{Register D}}{0} = \text{Register X} \]

\[ \text{remainder} = \text{Register D} \]

**Figure 3.15.** The \texttt{fdiv} instruction takes two 16-bit inputs and generates a 16-bit quotient and a 16-bit remainder.

Condition code bits are set after \( R = \frac{(65536 \times D)}{X} \).
- **Z:** result is zero,
- **V:** overflow if \( \text{RegX} = \text{RegD} \), result >$FFFF$
- **C:** divide by zero,

Let \( N \) and \( M \) be 16-bit unsigned locations. \( M = 12.34 \times N \).
We approximate 12.34 by \( \frac{65536}{5311} \).

\[
\begin{align*}
\text{ldd} & \quad N \\
\text{ldx} & \quad \#5311 \\
\text{fdiv} & \quad \text{RegX} = \frac{(65536 \times N)}{5311} \\
\text{stx} & \quad M
\end{align*}
\]

**Checkpoint 3.15:** Let \( N \) and \( M \) be 8-bit unsigned locations. Write assembly code to implement \( M = \frac{(7 \times N)}{31} \).

### 3.3.3. Shift operations

The N bit is set if the result is negative.
The Z bit is set if the result is zero.
The V bit is set on a signed overflow, change in the sign bit.
The C bit is the carry out after the shift.

\[
\text{asla} \quad \text{RegA} = \text{RegA} \times 2 \quad \text{(same as lsla)}
\]
3.3.4. Logical operations

The N bit will be set if the result is negative.
The Z bit will be set if the result is zero.
Clear V=0 bit.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anda #w</td>
<td>RegA = RegA &amp; w</td>
</tr>
<tr>
<td>anda U</td>
<td>RegA = RegA &amp; [U]</td>
</tr>
<tr>
<td>andb #w</td>
<td>RegB = RegB &amp; w</td>
</tr>
<tr>
<td>andb U</td>
<td>RegB = RegB &amp; [U]</td>
</tr>
<tr>
<td>bita #w</td>
<td>RegA &amp; w</td>
</tr>
<tr>
<td>bita U</td>
<td>RegA &amp; [U]</td>
</tr>
<tr>
<td>bitb #w</td>
<td>RegB &amp; w</td>
</tr>
<tr>
<td>bitb U</td>
<td>RegB &amp; [U]</td>
</tr>
<tr>
<td>coma</td>
<td>RegA = $FF - RegA, RegA = ~RegA</td>
</tr>
<tr>
<td>comb</td>
<td>RegB = $FF - RegB, RegB = ~RegB</td>
</tr>
</tbody>
</table>
### Subroutines and the stack

**Classical definition of the stack**
- Push saves data on the top of the stack,
- Pull removes data from the top of the stack
- Stack implements last in first out (LIFO) behavior

**Many uses of the stack**
- Temporary calculations
- Subroutine (function) return addresses
- Subroutine (function) parameters
- Local variables

The following code pushes the numbers 1, 2, and 3 in that order.
```
ldaa #1
psha  ; push 1 on the stack
ldaa #2
psha  ; push 2 on the stack
ldaa #3
psha  ; push 3 on the stack
```
Figure 3.16. stack as three elements are pushed.

At this point if one were to pull from the stack execute \texttt{pula}, the 3 would be returned, the 2 would now be on the top of the stack.

Figure 3.17. The stack holding two elements, with the 2 on top.