1.) **DATA TRANSFER**
   * MOVE X, Y (set CON(Y) to CON(X))
   * LOAD X (copy CON(X) into register R)
   * STORE X (copy contents of register R into memory cell X)

2.) **ARITHMETIC**
   * ADD_1 X (set CON(R) to CON(X) + CON(R))
   * ADD_2 X, Y (set CON(Y) to CON(X) + CON(Y))
   * ADD_3 X, Y, Z (set CON(Z) to CON(X) + CON(Y))

3.) **COMPARE**
   The result of a compare operation is to set the values of special bits called condition codes:
   - **GT**, **EQ**, **LT**
   - **COMPARE X, Y** (compare CON(X) to CON(Y) and set condition codes accordingly)

<table>
<thead>
<tr>
<th>Condition</th>
<th>GT</th>
<th>EQ</th>
<th>LT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CON(X) &gt; CON(Y)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CON(X) = CON(Y)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>CON(X) &lt; CON(Y)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
4. **Branching**

These instructions alter the normal sequential flow of program execution.

- **JUMP** X  (take next instruction from cell X.)
- **JUMP GT** X  (if GT = 1 take next instruction from cell X, otherwise follow sequence.)
- **JUMP EQ** X
- **JUMP LT** X
- **JUMP GE** X
- **JUMP LE** X
- **JUMP NEQ** X
- **HALT**

**IN THE FOLLOWING EXAMPLES, ASSUME THAT THE VALUES A, B, C, D ARE STORED IN CELLS 200, 201, 202, 203 RESPECTIVELY.**

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>a</td>
</tr>
<tr>
<td>201</td>
<td>b</td>
</tr>
<tr>
<td>202</td>
<td>c</td>
</tr>
<tr>
<td>203</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Example:**

**Write machine language instructions to:**

Set A to 6 + C.

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>LOAD 201</td>
<td>put CON(201) into R</td>
</tr>
<tr>
<td>101</td>
<td>ADD 202</td>
<td>put CON(202) + CON(R) into R</td>
</tr>
<tr>
<td>102</td>
<td>STORE 200</td>
<td>put CON(R) into 200</td>
</tr>
</tbody>
</table>

**Or**

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>ADD2 201, 202</td>
<td>put CON(201) + CON(202) in 202</td>
</tr>
<tr>
<td>101</td>
<td>MOVE 202, 200</td>
<td>put CON(202) in 200 (Note: C is lost)</td>
</tr>
</tbody>
</table>

**Or**

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>ADD2 201, 202, 200</td>
<td>put CON(201) + CON(202) into 200 (Note: R not lost)</td>
</tr>
</tbody>
</table>

This illustrates the point that a complex instruction set leads to shorter programs, and conversely, a simpler instruction set leads to longer programs.
### Example 1

If \( a = b \) set \( c \) to \( d \)

```plaintext
\[
\begin{array}{ll}
100 & \text{COMPARE 200,201} \\
101 & \text{JUMP} 103 \\
102 & \text{JUMP} 104 \\
103 & \text{MOVE 203,202} \\
104 & \text{...}
\end{array}
\]
```

### Example 2

If \( a < b \) set \( c \) to \( d \)
Else set \( c \) to 20

```plaintext
\[
\begin{array}{ll}
100 & \text{COMPARE 200,201} \\
101 & \text{JUMP} 106 \\
102 & \text{LOAD} 203 \\
103 & \text{ADD} 203 \\
104 & \text{STORE} 202 \\
105 & \text{JUMP} 107 \\
106 & \text{MOVE 203,202} \\
107 & \text{...}
\end{array}
\]
```

```
\[
\text{Put CON(203) in R} \\
\text{Put CON(203)+CON(R) in R} \\
\text{Put CON(R) in 202}
\]
Ex.
Repeat until \( a > c \)
set \( a \) to \( a + b \)
end loop.

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>LOAD 201</td>
<td>Put ( \text{CON}(201) ) in R</td>
</tr>
<tr>
<td>101</td>
<td>ADD, 200</td>
<td>Put ( \text{CON}(200) + \text{CON}(R) ) in R</td>
</tr>
<tr>
<td>102</td>
<td>STORE 200</td>
<td>Put ( \text{CON}(R) ) in 200</td>
</tr>
<tr>
<td>103</td>
<td>COMPARE 200,202</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>JUMP GT 106</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>JUMP 100</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In addition to the instruction register, the control unit contains the instruction decoder and the program counter.

![Diagram of instruction register (IR) with K bits in IR_op, IR_addr, and 2^K output lines to ALU, I/O, memory, etc.]}
The instruction decoder takes the k bits in IR0 and interprets them as a binary number in the range 0 to $2^k - 1$. A 1 is placed on the corresponding output line, which activates the circuitry needed to perform the desired operation.

The program counter holds the address of the next instruction to be executed. It is incremented after each instruction step by the number of bytes in a single instruction.

Ex: 

Ex: 

IR0 \[\rightarrow\] IR012 \[\rightarrow\] IR

# bits: 8 24 24 24

Each instruction occupies 80 bits \(\Rightarrow 10\) bytes in memory.

Fig. 5.18 on p. 203 shows the overall organization of the subsystems in the Von Neumann architecture.

Although our discussion of the Von Neumann architecture has been somewhat superficial, we have come quite far.
WE HAVE SEEN HOW THE MOST ELEMENTARY DEVICES (TRANSISTORS) CAN BE USED TO BUILD COMPLEX SUBSYSTEMS (BASES, CONTROL CIRCUITS, ADDERS, ETC.) WHICH CAN THEN BE USED TO IMPLEMENT OUR ALGORITHM STEPS.

FIG. 5.19 P. 204 SHOWS THE INSTRUCTION SET FOR A VERY SIMPLE VON NEUMANN COMPUTER. WE WILL USE THIS INSTRUCTION SET IN THE NEXT PROGRAMMING ASSIGNMENT.

IN PSET YOU WILL WRITE SIMULATED MACHINE LANGUAGE PROGRAMS FOR A SIMULATED PROCESSOR.

READ: 5.3 HISTORICAL OVERVIEW P. 208-228

HW 6: CHAP 5 P. 228:
   3, 4, 5, 8, 11, 17, 18 abcde