12. von Neumann Machines

TOY vs. your laptop

Two different computing machines

- Both implement basic data types, conditionals, loops, and other low-level constructs.
- Both can have arrays, functions, libraries, and other high-level constructs.
- Both have infinite input and output streams.

Q. Is 256 words enough to do anything useful?
A. Yes! (Stay tuned.)

OK, we definitely want a faster version with more memory when we can afford it...

Is 4096 bits of memory enough to do anything useful?

- Perspective
- A note of caution
- Practical implications
- Simulation
Is thousands of bits of memory enough to do anything useful?

LINC computer, MIT
12x2048 = 24576 bits of memory
Used for many biomedical and other experiments

Prof. Clark’s father, 1963

Prof. Clark and his father, 2013

Is 4096 bits enough to do anything useful?

Contents of memory, registers, and PC at a particular time
- Provide a record of what a program has done.
- Completely determines what the machine will do.

Total number of bits in the state of the machine
- 256 x 16 (memory)
- 16 x 16 (registers)
- 8 (PC)

Total number of different states: \(2^{4360}\) (!!!)

Total number of different states that could be observed if the universe were fully packed with laptops examining states for its entire lifetime: \(<2^{400}\).

Bottom line: We will never know what a 256-word machine can do.

An early computer

**ENIAC.** Electronic Numerical Integrator and Calculator
- First widely known general purpose electronic computer.
- Conditional jumps, programmable, but no memory.
- Programming: Change switches and cable connections.
- Data: Enter numbers using punch cards.

**Facts and figures**
- 30 tons
- 30 x 50 x 8.5 ft
- 17,468 vacuum tubes
- 300 multiply/sec

**ENIAC.** 1946

A famous memo

**First Draft of a report to the EDVAC, 1945**
- Written by John von Neumann, Princeton mathematician
- EDVAC; second computer proposed by Eckert and Mauchly.
- Memo written on a train trip to Los Alamos.
- A brilliant summation of the stored program concept.
- Influenced by theories of Alan Turing.
- Has influenced the design of every computer since.

Who invented the stored program computer?
- Fascinating controversy.
- Eckert-Mauchly discussed the idea before von Neumann arrived on the scene.
- Goldstine circulated von Neumann's first draft because of intense interest in the idea.
- Memo placed the idea in the public domain and prevented it from being patented.
- von Neumann never took credit for the idea, but never gave credit to others, either.
Another early computer

EDSAC. Electronic Delay Storage Automatic Calculator
- Second *stored program* computer (after EDVAC).
- Data and instructions encoded in binary.
- Could load programs, not just data, into memory.
- Could change program without rewiring.

**Facts and figures**
- 512 17-bit words (8074 bits)
- 2 registers
- 16 instructions
- Input: paper tape
- Output: teleprinter

**Implications**

Stored-program (*von Neumann*) architecture is the basis of nearly all computers since the 1950s.

**Practical implications**
- Can load programs, not just data, into memory (download apps).
- Can write programs that produce programs as output (compilers).
- Can write programs that take programs as input (simulators).

**Profound implications (stay tuned for theory lectures)**
- TOY can solve *any* problem that *any* other computer can solve (!)
- Some problems *cannot* be solved by any computer at all (!)

**An instructive scenario**

Alice, a scientist, develops a procedure for her experiments.
- Uses a scientific instrument connected to a paper tape punch.
- Takes the paper tape to a computer to process her data.
- Uses array code from last lecture to load her data.
- Writes array-processing code that analyzes her data.
- Punches out the results on paper tape to save them.
Arrays example: Read an array from standard input (continued from last lecture)

Register trace

| PC | 10 | 7101 | R1 ← 1 |
| 11 | 8AFF | RA ← N |
| 12 | 7680 | R6 ← 80 |
| 13 | 7800 | RB ← 0 |
| 14 | C21B | if (RA ← 0) PC ← 1B |
| 15 | 8CFF | read RC from stdin |
| 16 | 156B | RS ← R6 + RB |
| 17 | BC05 | new[R5] ← RC |
| 18 | 1BB1 | RB ← RB + 1 |
| 19 | 2A1A | RA ← RA - 1 |
| 2A | C014 | PC ← 14 |

[begin array processing code]

Memory

|    | 80 | 0001 |
|    | 81 | 0002 |
|    | 82 | 0003 |
|    | 83 | 0004 |
|    | 84 | 0005 |
|    | 85 | 0006 |

An instructive scenario (continued)

Alice, a scientist, develops a procedure for her experiments.

- Uses a scientific instrument connected to a paper tape punch.
- Takes the paper tape to a computer to process her data.
- Uses array code from last lecture to load her data.
- Writes array-processing code that analyzes her data.

Eve, a fellow scientist, runs some experiments, too.

What happens with Eve’s tape

Not what Alice expects!

- Memory 80–FE fills with 8888.
- 8888 appears on output.
- Address overflow from FF to 00.
- Memory 00–0F is overwritten.

And then things get worse...
What happens with Eve's tape when things get worse

```plaintext
<table>
<thead>
<tr>
<th>PC</th>
<th>Register trace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data is overwriting code?</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[begin array processing code]</td>
</tr>
</tbody>
</table>

```plaintext

Memory

```plaintext
int a = StdIn.read();
int b = 0;
while (a != 0) {
  int e = StdIn.read();
  arr[b] = e;
  b++; a--;
}
```

Data is overwriting code!

Or is it code overwriting data?

What happens with Eve's tape when things get worse: Eve OWNS Alice's computer

```plaintext
[maniacal laugh]
```

Remember me?

She could have loaded any program at all...
Programs that process programs on TOY

von Neumann architecture
- No difference between data and instructions.
- Same word can be data one moment, an instruction the next.

Early programmers immediately realized the advantages
- Can save programs on physical media (dump).
- Can load programs at another time (boot).
- Can develop higher-level languages (assembly language).

Q. What does the following program leave in R2?

\[
\begin{array}{c}
10 & 7 & C & 0 & A & R_C \leftarrow 10_{10} \\
11 & C & 0 & 1 & 6 & R_C \leftarrow 12 \\
12 & 1 & 2 & 2 & 2 & R_2 \leftarrow R_2 + R_2 \\
13 & 2 & C & 1 & 2 & R_C \leftarrow R_C - 1 \\
14 & D & C & 1 & 2 & \text{IF } (R_C > 0) \text{ PC } \leftarrow 12 \\
15 & 0 & 0 & 0 & 0 & \text{HALT} \\
16 & 7 & 1 & 0 & 1 & R_1 \leftarrow 1 \\
17 & 7 & 2 & 0 & 1 & R_2 \leftarrow 1 \\
18 & C & 0 & 1 & 2 & \text{PC } \leftarrow 12
\end{array}
\]

A. \(2^{16} = 1024_{10} = 0400_{16}\). Same as TEQ 2.

Example of a patch—very common in early programming.

TEQ 3 on TOY

Q. How to save a program for another day?
- Day's work represents patches and other code entered via switches.
- Must power off (vacuum tubes can't take the heat).

A. Write a short program to dump contents of memory to tape.
- key in program via switches in memory locations 00–08.
- Run it to save data/instructions in memory 10–FE.

DUMP code

\[
\begin{array}{c}
00 & 7 & 1 & 0 & 1 & R_1 \leftarrow 1 \\
01 & 7 & 2 & 1 & 0 & R_2 \leftarrow 10 \\
02 & 7 & 3 & F & F & R_1 \leftarrow 00_{16} \\
03 & A & A & 0 & 2 & \text{RA } \leftarrow \text{mem[R2]} \\
04 & 9 & A & F & \text{write } \text{RA to stdout} \\
05 & 1 & 2 & 2 & 1 & R_2 \leftarrow R_2 + 1 \\
06 & 2 & 4 & 3 & 2 & R_4 \leftarrow 00_{16} - R_2 \\
07 & D & 4 & 0 & 3 & \text{if } (R_4 > 0) \text{ PC } \leftarrow 03 \\
08 & 0 & 0 & 0 & 0 & \text{halt}
\end{array}
\]

TEQ 3 on TOY

Q. What does the following program leave in R2?

\[
\begin{array}{c}
10 & 7 & C & 0 & A & R_C \leftarrow 10_{10} \\
11 & C & 0 & 1 & 6 & R_C \leftarrow 12 \\
12 & 1 & 2 & 2 & 2 & R_2 \leftarrow R_2 + R_2 \\
13 & 2 & C & 1 & 2 & R_C \leftarrow R_C - 1 \\
14 & D & C & 1 & 2 & \text{IF } (R_C > 0) \text{ PC } \leftarrow 12 \\
15 & 0 & 0 & 0 & 0 & \text{HALT} \\
16 & 7 & 1 & 0 & 1 & R_1 \leftarrow 1 \\
17 & 7 & 2 & 0 & 1 & R_2 \leftarrow 1 \\
18 & C & 0 & 1 & 2 & \text{PC } \leftarrow 12
\end{array}
\]

TEQ 3 on TOY

Q. What does the following program leave in R2?

\[
\begin{array}{c}
10 & 7 & C & 0 & A & R_C \leftarrow 10_{10} \\
11 & C & 0 & 1 & 6 & R_C \leftarrow 12 \\
12 & 1 & 2 & 2 & 2 & R_2 \leftarrow R_2 + R_2 \\
13 & 2 & C & 1 & 2 & R_C \leftarrow R_C - 1 \\
14 & D & C & 1 & 2 & \text{IF } (R_C > 0) \text{ PC } \leftarrow 12 \\
15 & 0 & 0 & 0 & 0 & \text{HALT} \\
16 & 7 & 1 & 0 & 1 & R_1 \leftarrow 1 \\
17 & 7 & 2 & 0 & 1 & R_2 \leftarrow 1 \\
18 & C & 0 & 1 & 2 & \text{PC } \leftarrow 12
\end{array}
\]

Example of a patch—very common in early programming.
**Booting**

Q. How to load a program on another day?

A. Reboot the computer.

- Turn it on.
- Key in boot code via switches in memory locations 00–08.
- Run it to load data/instructions in memory 10–FE.

Why not 00–0F? Would overwrite program?

---

**BOOT code**

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>RA</th>
<th>RB</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>70 1</td>
<td>72 10</td>
<td>73 FF</td>
<td>8 A FF</td>
<td>1 2 2 1</td>
<td>2 4 3 2</td>
<td>D 4 0 3</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

int i = 0x10;

```c
int i = 0x10;
do {
  i++;
} while (i < 255)
```

Early programmers would pride themselves in the speed they could enter such code.

---

**Assembly language**

**Assembly language**

- Program in a higher-level language.
- Write a machine-language program to translate.
- Used widely from early days through the 1990s.
- Still used today.

**TOY machine code**

<table>
<thead>
<tr>
<th>00</th>
<th>07 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>72 10</td>
</tr>
<tr>
<td>02</td>
<td>73 FF</td>
</tr>
<tr>
<td>03</td>
<td>8 A FF</td>
</tr>
<tr>
<td>04</td>
<td>B A 0 2</td>
</tr>
<tr>
<td>05</td>
<td>1 2 2 1</td>
</tr>
<tr>
<td>06</td>
<td>2 4 3 2</td>
</tr>
<tr>
<td>07</td>
<td>D 4 0 3</td>
</tr>
<tr>
<td>08</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

TOY assembly code

<table>
<thead>
<tr>
<th>00</th>
<th>LA R1,01</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>LA R2,10</td>
</tr>
<tr>
<td>02</td>
<td>LA R3,FF</td>
</tr>
<tr>
<td>03</td>
<td>LOOP RD RA</td>
</tr>
<tr>
<td>04</td>
<td>SI RA,R2</td>
</tr>
<tr>
<td>05</td>
<td>A R2,R1</td>
</tr>
<tr>
<td>06</td>
<td>S R4,R3,R2</td>
</tr>
<tr>
<td>07</td>
<td>BP R4, LOOP</td>
</tr>
<tr>
<td>08</td>
<td>H</td>
</tr>
</tbody>
</table>

**Advantages**

- Mnemonics, not numbers, for opcodes.
- Symbols, not numbers, for addresses.
- Relocatable.

---

**Tip of the iceberg**

Practical implications of von Neumann architecture

- Installers that download applications.
- Compilers that translate Java into machine language.
- Simulators that make one machine behave like another (stay tuned).
- Cross-compilers that make code for one machine on another.
- Dumping and booting.
- Viruses.
- Virus detection.
- Virtual machines.
- Thousands of high-level languages.
- [an extremely long list]
Is TOY real?

Q. How did we debug all our TOY programs?
A. We wrote a Java program to simulate TOY.

Comments
• YOU could write this program (stay tuned).
• We designed TOY by refining this code.
• All computers are designed in this way.

Provocative questions
• Is Android real?
• Is Java real?
• Suppose we run our TOY simulator on Android. Is TOY real?

Toy simulator in Java

A Java program that simulates the TOY machine.
• Take program from a file named in the command line.
• Take TOY Stdin/StdOut from Java StdIn/Stdout.

```java
class TOYlecture {
    public static void main(String[] args) {
        int pc = 0x10; // program counter
        int[] r = new int[16]; // registers
        int mem = new int[256]; // main memory

        for (int i = 0x10; i < 0x1FF; i++)
            mem[i] = Integer.parseInt(in.readLine(), 16);

        while (true) {
            int inst = mem[pc++]; // fetch and increment
            // decode (next slide)
            // execute (second slide following)
        }
    }
}
```

Toy simulator: decoding instructions

Bitwacking is the same in Java as in TOY
• Extract fields for both instruction formats.
• Use shift and mask technique.

Decode
```java
int inst = mem[pc++]; // fetch and increment
int op = (inst >> 2) & 15; // opcode (bits 12-15)
int d = (inst >> 8) & 15; // dest d (bits 9-12)
int s = (inst >> 12) & 15; // source s (bits 04-07)
int t = (inst >> 16) & 15; // source t (bits 00-03)
int addr = (inst >> 20) & 255; // addr (bits 00-07)
```

Example: Extract destination d from 1CAB

<table>
<thead>
<tr>
<th>inst</th>
<th>C</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>000011110010101011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
n >> 8
0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0
```

```
15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
(n >> 8) & 15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Bitwise AND of data and “mask” result is 0 where mask is 0 data bit where mask is 1

Toy simulator: executing instructions

Use Java switch statement to implement the simple state changes for each instruction.

execute
```java
if (op == 0) break; // halt
switch (op) {
    case 1: R[d] = R[s] + R[t]; break;
    case 2: R[d] = R[s] - R[t]; break;
    case 3: R[d] = R[s] & R[t]; break;
    case 4: R[d] = R[s] ^ R[t]; break;
    case 5: R[d] = R[s] | R[t]; break;
    case 6: R[d] = R[s] >> R[t]; break;
    case 7: R[d] = addr; break;
    case 8: R[d] = mem[addr]; break;
    case 9: mem[addr] = R[d]; break;
    case 10: R[d] = mem[R[t]]; break;
    case 11: mem[R[s]] = R[d]; break;
    case 12: if (R[d] == 0) pc = addr; break;
    case 13: if (R[d] > 0) pc = addr; break;
    case 14: pc = R[d]; break;
    case 15: R[d] = pc; pc = addr; break;
}
```
Toy development environment

Another Java program that simulates the TOY machine
- Includes graphical simulator.
- Includes single stepping, full display of state of machine, and many other features.
- Includes many simple programs.
- Written by a COS 126 graduate.
- Available on the bookstore.
- YOU CAN develop TOY software.

Same approach used for all new systems nowadays
- Build simulator and development environment.
- Develop and test software.
- Build and sell hardware.

Toy simulator in Java

```
public class TOYlecture
public static void main(String[] args) {
    int pc = 0x80; // program counter
    int mem[] = new int[16]; // main memory
    int i = 0x00; // new string()...
    int inst = mem[i++]; // fetch/increment
    while (true) {
        if (inst == 0x00) // halt
            break;
        int addr = mem[i] & 0XFF; // addr
        mem[i] = mem[i] & 0X0F; // mask off lower bits
        inst = mem[i++]; // fetch
        while (true) {
            switch (inst) {
                case 0x00: // add
                    mem[addr] += pc; pc += addr; break;
                case 0x01: // sub
                    mem[addr] -= pc; pc += addr; break;
                case 0x02: // mov
                    pc = mem[addr]; break;
                case 0x03: // inc
                    pc++; break;
                case 0x04: // dec
                    pc--; break;
                case 0x05: // xor
                    mem[addr] ^= pc; pc += addr; break;
                case 0x06: // or
                    mem[addr] |= pc; pc += addr; break;
                case 0x07: // and
                    mem[addr] &= pc; pc += addr; break;
                case 0x08: // not
                    mem[addr] = ~pc; pc += addr; break;
                case 0x09: // left
                    mem[addr] = (mem[addr] << pc) & 0XFF; pc += addr; break;
                case 0x0a: // right
                    mem[addr] = (mem[addr] >> pc) & 0XFF; pc += addr; break;
                case 0x0b: // jump
                    pc = mem[addr]; break;
                case 0x0c: // return
                    mem[addr] = (mem[addr] >> 8) & 0X00; pc += addr; break;
                case 0x0d: // print
                    System.out.println(mem[addr]); pc += addr; break;
                case 0x0e: // load
                    mem[addr] += pc; pc += addr; break;
                case 0x0f: // choice
                 Mem[addr] = (Mem[addr] ^ mem[addr]) & 0XFF; pc += addr; break;
            }
        }
        if (inst == 0x00) // next word
            inst = mem[i++]; // fetch/increment
    }
}
```

Backward compatibility

Q. Time to build a new computer. What to do with old software?

Approach 1: Rewrite it all
- Costly and time-consuming.
- Error-prone.
- Boring.

Approach 2: Simulate the old computer on the new one.
- Not very difficult.
- Still likely more efficient.
- Succeeds for all old software.

Result. Old software remains available.

Disturbing thought: Does anyone know how it works?
Another note of caution

An urban legend about backward compatibility.

- Space shuttle solid rocket booster needed to be transported by rail.
- US railroads were built by English expats, so the standard rail gauge is 4 feet 8.5 inches.
- English rail gauge was designed to match ruts on old country roads.
- Ruts on old country roads were first made by Roman war chariots.
- Wheel spacing on Roman war chariots was determined by the width of a horse's back end.

End result. Key space shuttle dimension determined by the width of a war horse's back end.

Worthwhile takeaway. Backwards compatibility is Not Necessarily Always a Good Thing.

Backward compatibility is pervasive in today’s world

Much of our infrastructure was built in the 1970s on machines not so different from TOY.

Time to design and build something suited for today’s world? Go for it! That means YOU!

Virtual machines

Building a new rocket? Simulate it to test it.
- Issue 1: Simulation may not reflect reality.
- Issue 2: Simulation may be too expensive.

Building a new computer? Simulate it to test it.
- Advantage 1: Simulation is reality (it defines the new machine).
- Advantage 2: Can develop software without having machine.
- Advantage 3: Can simulate machines that may never be built.

Examples in today’s world.
- Virtual memory.
- Java virtual machine.
- Amazon cloud.

Virtual machines of many, many types (old and new) are available for use on the web. Internet commerce is moving to such machines.

Layers of abstraction

Computer systems are built by accumulating layers of abstraction.

Is TOY real?

A machine that may never be built

Is your computer real?

Examples in today’s world.
- Virtual memory.
- Java virtual machine.
- Amazon cloud.

Virtual machines of many, many types (old and new) are available for use on the web.

Approaching a new problem?
- Build an (abstract) language for expressing solutions.
- Design an (abstract) machine to run programs written in the language.
- Food for thought: Why build the machine? Just simulate it instead!
12. von Neumann Machines

Sections 5.1–4