CS 126 Lecture A1: TOY Machine

Outline

• Introduction
• Toy machine
• Machine language instructions
• Example machine language programs
• Conclusions
Brief History Leading to the Dominance of von Neumann Architecture

• 1940s, Atanasoff, Iowa State, first special-purpose electronic computer, binary representation of numbers
• ~1946, ENIAC, Eckert and Mauchly, UPenn, first general-purpose electronic computer
  - 100 ft long, 8.5 ft high, several ft wide, 18000 vacuum tubes
  - conditional jumps, programmable
  - code: setting switches, data: punch cards
  - Used to compute artillery firing tables
• 1944, von Neumann, visited ENIAC, the “von Neumann Memo”, concept of a “stored-program” computer
• 1949, Wilkes, EDSAC, first stored-program computer
• 1946, von Neumann, Goldstine, Burks, IAS machine, Princeton, the report pioneered most modern computer architecture concepts

Why Study Machine Language Programming Today

• Learn how computers really work
• There are still (a few) situations where machine language programming is necessary
• The first step towards understanding how to build better computers
Outline

• Introduction
• Toy machine
• Machine language instructions
• Example machine language programs
• Conclusions

Toy Machine

An imaginary machine, similar to ancient early computers
* today's microprocessors

Box with switches and lights, maybe TTY
Inside the Box

- ALU (arithmetic logic unit) -- executes instructions to manipulate data
- 8 registers -- the fastest form of storage, on-chip in modern computers, used as scratch space during computation
- PC (program counter) -- a register with special meaning, keeps track of the next instruction to be executed
- 256 16-bit words of memory -- stores both code and data

Binary Numbers

- Machine consists of two-state ("ON-OFF") switches and lights
- Use binary encoding to represent values

Ex:

\[
0.6375 = 0001100011100111
\]

\[
\begin{array}{cccccccccccc}
. & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
. & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
. & 12 & 11 & & & 7 & 6 & 5 & & 2 & 1 & 0 \\
. & 2 + 2 & & 2 + 2 & & 2 + 2 & & 2 + 2 & & 2 + 2 \\
.6375 = 4096 + 2048 + 128 + 64 + 32 + 4 + 2 + 1
\end{array}
\]
Hexadecimal Numbers

Hexadecimal (base-16) notation provides shorthand binary code four bits at a time.

<table>
<thead>
<tr>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

Ex:

1. $6375 = \overline{0001100011100111}$
2. $\overline{18E7}$
3. $6375 = 1 \times 16^3 + 8 \times 16^2 + 14 \times 16 + 7 \times 16$
4. $= 4096 + 2048 + 224 + 7$

TOY machine memory

Contents of machine in hexadecimal ("dump")

PC: 0010

R0: R1: R2: R3: R4: R5: R6: R7:
0000 0000 0000 0000 0000 0000 0000 0000
08: 0000 0000 0000 0000 0000 0000 0000 0000
10: 0000 0000 0000 0000 0000 0000 0000 0000
18: 0000 0000 0000 0000 0000 0000 0000 0000
20: 0000 0000 0000 0000 0000 0000 0000 0000
28: 0000 0000 0000 0000 0000 0000 0000 0000

Programmers still look at dumps, even today

Contents of memory

- record of what program has done
- determines (with PC) what machine will do
Program and Data

Program: sequence of instructions

Instruction:
16-bit word (interpreted one way)

Data:
16-bit word (interpreted other ways)

How to Use the TOY Machine

To run a program
* load the program and data
  (set switches, press LOAD for each word)
* set switches to address of first instruction
* press GO
How to Use the TOY Machine

GO button
  * loads PC from address switches
  * initiates FETCH-INCREMENT-EXECUTE cycle
  * machine runs until halt instruction hit

FETCH (get instruction from memory into CPU)
INCREMENT program counter (PC)
EXECUTE (may require data from or to memory)

Output:
  * read contents of memory word in lights
  * system call can write output to an output device (tty)

Outline

• Introduction
• Toy machine
• Machine language instructions
• Example machine language programs
• Conclusions
TOY Instructions

- Encode each of these instructions using 16 bits
- Need to divide up the 16 bits to denote components of each type of instructions
- Instruction formats - different ways of dividing up the 16 bits

Instruction Format 1

**FORMAT 1: register-register**

<table>
<thead>
<tr>
<th>4 bits</th>
<th>4 bits</th>
<th>4 bits</th>
<th>4 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>dest</td>
<td>regA</td>
<td>regB</td>
</tr>
</tbody>
</table>

Ex: 1234 means
- add register R3 and R4
- put the result in R2
  \[ R2 \leftarrow R3 + R4 \]
- Other instrs: sub, mult, xor, and
Instruction Format 2

**FORMAT 2**: register-memory, register-immediate

<table>
<thead>
<tr>
<th>4 bits</th>
<th>4 bits</th>
<th>8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>dest</td>
<td>addr/const</td>
</tr>
</tbody>
</table>

Ex: 9234 means "load memory loc 34 (hex) into R2"
R2 <- mem[34]

Ex: A234 means "store R2 into memory loc 34"
mem[34] <- R2

Ex: B234 means "load the value 0034 into R2"
R2 <- 0034

Other instrs: shifts, halt, system call, jumps

Logical Instructions

**opcode**

- C: xor
- D: and
- E: shift right
- F: shift left

xor, and: bit-by-bit operations
shift: move bits
**Right-Shift**

\[ x \]

\[
\begin{array}{c}
1001010110000011 \\
\downarrow \text{discarded} \\
0000000000100101 \\
x >> 10
\end{array}
\]

**Bit-by-Bit-And**

\[
\begin{array}{c}
1001010110000010 \\
0011001010110011 \\
\downarrow a \land b \\
0001001010000010
\end{array}
\]

**MASKing with \textit{and} instruction**

<table>
<thead>
<tr>
<th>a</th>
<th>1010010x0111010</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>000000010000000</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>0000000x0000000</td>
</tr>
</tbody>
</table>
Other Logical Operations

- Can implement other logical operations

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>&amp;</th>
<th>^</th>
<th>(a &amp; b)</th>
<th>^ (a ^ b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Outline

- Introduction
- Toy machine
- Machine language instructions
- **Example machine language programs**
- Conclusions
Sample TOY program o: arithmetic

Ex: TOY code for C expression t = b*b - 4*a*c
memory loc D0 is used for storing a
D1 for b
D2 for c
D3 for t

Suppose memory locations 10-19 contain
10: 91D1 3111 B204 93D0 94D2 3223 3224 2112
18: A1D3 0000

Set PC to 10; Press GO. TOY computes the value.
Step-by-step trace:
10: 91D1 R1 <- b
11: 3111 R1 <- b*b
12: B204 R2 <- d
13: 93D0 R3 <- a
14: 94D2 R4 <- c
15: 3223 R2 <- 4*a
16: 3224 R2 <- (4*a)*c
17: 2112 R2 <- (b*b) - (4*a*c)
18: A1D3 t <- (b*b - 4*a*c)
19: 0000 halt

(C compiler produces code like this)

TOY Demo
Sample TOY program 1: more arithmetic

Ex: Suppose memory locations 10-1F contain
10: B001 B200 B101 1221 1110 1221 1110 1221 1110 1221 1110 1221 1110 0000 0000
18: 1110 1221 1110 1221 1110 1221 1110 1221 0000 0000

* Set PC to 10. Press GO. What happens?

* Step-by-step trace:

```
10: B001  R0 <- 0001
11: B200  R2 <- 0000
12: B101  R1 <- 0001
13: 1221  R2 <- R2 + R1
14: 1110  R1 <- R1 + R0
15: 1221  R2 <- R2 + R1
16: 1110  R1 <- R1 + R0
17: 1221  R2 <- R2 + R1
18: 1110  R1 <- R1 + R0
19: 1221  R2 <- R2 + R1
1A: 1110  R1 <- R1 + R0
1B: 1221  R2 <- R2 + R1
1C: 1110  R1 <- R1 + R0
1D: 1221  R2 <- R2 + R1
1E: 0000  halt
```

Computes 1 + 2 + 3 + 4 + 5 + 6 = 21

Sample TOY program 2: loop

* Suppose memory locations 10-1F contain
10: B106 B200 B001 1221 1110 6113 0000 0000
18: 1110 1221 1110 1221 6113 0000 0000

* Set PC to 10. Press GO. What happens?

* Step-by-step trace:

```
10: B106  R0 <- 0006
11: B200  R2 <- 0000
12: B001  R0 <- 0001
13: 1221  R2 <- R2 + R1
14: 1110  R1 <- R1 - R0
15: 6113  jump if (R1 > 0)
16: 1221  R2 <- R2 + R1
17: 2110  R1 <- R1 - R0
18: 6113  jump if (R1 > 0)
19: 1221  R2 <- R2 + R1
1A: 2110  R1 <- R1 - R0
1B: 6113  jump if (R1 > 0)
1C: 1221  R2 <- R2 + R1
1D: 2110  R1 <- R1 - R0
1E: 6113  jump if (R1 > 0)
1F: 1221  R2 <- R2 + R1
20: 2110  R1 <- R1 - R0
21: 6113  jump if (R1 > 0)
22: 1221  R2 <- R2 + R1
23: 2110  R1 <- R1 - R0
24: 6113  jump if (R1 > 0)
25: 1221  R2 <- R2 + R1
26: 2110  R1 <- R1 - R0
27: 6113  jump if (R1 > 0)
28: 1221  R2 <- R2 + R1
29: 2110  R1 <- R1 - R0
2A: 6113  jump if (R1 > 0)
2B: 1221  R2 <- R2 + R1
2C: 2110  R1 <- R1 - R0
2D: 6113  jump if (R1 > 0)
2E: 1221  R2 <- R2 + R1
2F: 2110  R1 <- R1 - R0
30: 6113  jump if (R1 > 0)
31: 1221  R2 <- R2 + R1
32: 2110  R1 <- R1 - R0
33: 6113  jump if (R1 > 0)
34: 1221  R2 <- R2 + R1
35: 2110  R1 <- R1 - R0
36: 6113  jump if (R1 > 0)
37: 1221  R2 <- R2 + R1
38: 2110  R1 <- R1 - R0
39: 6113  jump if (R1 > 0)
3A: 1221  R2 <- R2 + R1
3B: 2110  R1 <- R1 - R0
3C: 6113  jump if (R1 > 0)
3D: 1221  R2 <- R2 + R1
3E: 2110  R1 <- R1 - R0
3F: 6113  jump if (R1 > 0)
```

Computes \( N + (N-1) + \ldots + 3 + 2 + 1 = N(N+1)/2 \)

for any value \( N \) loaded into Ri.
Horner’s Method

**Problem:**
- evaluate $2x^3 + 3x^2 + 9x + 7$ at $x = 10$
- assume “data” stored in locations 30--34
  
  x a b c d

  30: 000A 0002 0003 0009 0007 0000 0000

**First try:**
- compute $x^3$, mult. by $a$; compute $x^2$, ...
  (cumbersome, inefficient)

**Efficient algorithm (Horner’s method):**
- rewrite $ax^3+bx^2+cx+d$ as $((ax+b)x+c)x+d$

---

Sample TOY Program 3: Horner’s Method

**Efficient algorithm (Horner’s method):**
- rewrite $ax^3+bx^2+cx+d$ as $((ax+b)x+c)x+d$

10: 9430  R4 <- M[30]  000A  x
11: 9531  R5 <- M[31]  0002  a
12: 3554  R5 <- R5 * R4  0014  a*x
14: 1556  R5 <- R5 + R6  0017  a*x+b
15: 3554  R5 <- R5 * R4  00DC  (a*x+b)*x
16: 9633  R6 <- M[33]  0009  c
17: 1556  R5 <- R5 + R6  00E5  (a*x+b)*x + c
18: 3554  R5 <- R5 * R4  0956  ((a*x+b)*x+c)*x
19: 9634  R6 <- M[34]  0007  d
1A: 1556  R5 <- R5 + R6  095D  ((a*x+b)*x+c*x)+d
1B: 4502  write R5 to tty
Linear feedback shift register (LFBSR)

- R1 is LFBSR content
- R2 is a copy of R1
- So is R3
- Get 3rd bit to the right end
- Get 10th bit to the right end
- Only right-most bit of xor
- Left shift LFBSR
- Put in the new right-most bit

Sample TOY program 4: bit manipulation

Ex: suppose that memory locations 10-15 contain
10: 911F B000 1210 1310 E203 E30A C323 B401
18: D334 F101 C113 0000 0000 0000 0000 0684
* Set PC to 10. Press GO. What happens?

Step-by-step:

10: 911F  R1 <- 0684 0000011010000100
11: B000  R0 <- 0000
12: 1210  R2 <- R1 + R0 0000011010000100
13: 1310  R3 <- R1 + R0 0000011010000100
14: E203  R2 <- R2 >> 3 0000000011010000
15: E30A  R3 <- R3 >> 10 0000000000000000
16: C323  R3 <- R2 ^ R3 0000000011010001
17: B401  R4 <- 0001 0000000000000000
18: D334  R3 <- R3 & R4 0000000000000000
19: F101  R1 <- R1 << 1 0000110100001000
1A: C113  R1 <- R1 ^ R3 0000110100001001
1B: 0000  halt

*Simulates one step of LFBSR of Lecture 1
Outline

• Introduction
• Toy machine
• Machine language instructions
• Example machine language programs
• Conclusions

Basic Characteristics of TOY Machine

TOY is a “general purpose” computer

• “von Neumann” machine
  • instructions and data in same memory
  • can change program (control) w/o rewiring immediate applications
    profound implications

• sufficient power to perform any computation
  • limited only by amount of memory (and time)
    [stay tuned]

• similar to real machines
**“Computer Architecture”**

| Compilers | Machine language programmers |

---

**Instruction Set Architecture:** instruction set, registers, memory

**Implementation:** “Organization” and “Hardware”

---

- **Interface:** “instruction set architecture” (ISA)
  - visible to machine language programmers
  - boundary between software and hardware

- **Implementation**
  - “Organization”: interaction of high-level components
  - “Hardware”: low level specifics such as detailed logic design

- **Abstractions**
  - Can change hardware without changing organization
  - Can change implementation without changing ISA