Chapter 3: Programming

CS105: Great Insights in Computer Science
Rise of Clicker
(Infinite) Looping

Never never never stops!

(bug in book)

Never never never never stops!
Kinds of Loops

• Infinite loop

• “For” loop: Repeat a set number of times.
  ▶ Flexible.

• “Repeat until” loop: Until a condition holds.
  ▶ More Flexible.

• Recursion: Repeat substructure.
  ▶ Most Flexible.
Just Four, Thanks

- Sets a repeat count
- Stops after that number of repetitions
Unroll The Loop

- It’s as if the statements inside the loop are repeated four times.
More Power

• “For” (repeat) loops are great if you know how many times you will be repeating.

• Sometimes need something more powerful.

• “While” (repeat-until) loops keep going until a condition becomes true.

• Can behave like for loops...
Four Square Countdown

clear

set size to 50

set counter to 4

repeat until counter = 0

broadcast squareSize and wait

turn 5 degrees

change counter by -1
Unroll, With Conditions

clear
set size to 50
set counter to 4
if not counter = 0
  broadcast squareSize and wait
  turn 5 degrees
  change counter by -1
if not counter = 0
  broadcast squareSize and wait
  turn 5 degrees
  change counter by -1
if not counter = 0
  broadcast squareSize and wait
  turn 5 degrees
  change counter by -1
if not counter = 0
  broadcast squareSize and wait
  turn 5 degrees
  change counter by -1
Even More Power

- “While” (repeat-until) loops are great if repetitions are sequential.
- Sometimes need something more powerful.
- “Recursion” can allow control to proceed in multiple directions at once!
- But, can also behave like for loops...
Infinite Recursion

• Each “pattern” message spawns another.
• Ad infinitum
Proper Recursion

• “Base case” says what to do when the counter runs down.

• In this case, it stops when the counter reaches zero.
Computer Hierarchy

- Bits
- Logic (and, or, not)
- Boolean Algebra
- Programming Languages (Scratch, Java, etc...)
- English

Low Complexity → High Complexity
What Can We Do?

• Lots: Any function of bits, we can specify with logic gates.

• But, creating dedicated circuitry for every new problem is daunting and inefficient.

• Would like a way of using a fixed set of circuits to act like any circuitry we might want.

• We can use the state-machine idea to trade gates for time...
Christmas Light Architecture

(input) X → A → A
A → B → B
B → C

light output circuitry → light bulbs
Christmas Light Architecture

(input) X → state update circuitry → light output circuitry
A → A
B → B
C → C

→ (output)
Christmas Light Architecture

(input) X
A
B
C

state update circuitry

light output circuitry

A
B
C

copy back with 1 second delay
Christmas Light Architecture

(input) X
A
B
C

state update circuitry

A
B
C

light output circuitry

copy back with 1 second delay
Christmas Light Architecture

State update circuitry

Light output circuitry

(input) X
A
B
C

copy back with 1 second delay
Simple Computer

Copy back on a clock
Concrete Example: Adding

- We want to compute the sum of $x$ and $y$ (2-bit numbers). $z$ (3 bits) is the answer and $c$ (2 bits) is the carry.

- $z_0 = (x_0 \text{ and not } y_0) \text{ or } (\text{not } x_0 \text{ and } y_0)$

- $c_0 = (x_0 \text{ and } y_0)$
Truth Table That Adds Bits

• Basic step:
  ‣ 3 bits in
  ‣ 2 bits out
• sum bit
• new carry bit

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>c</th>
<th>c'</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Truth Table That Adds Bits

- Basic step:
  - 3 bits in
  - 2 bits out
- sum bit
- new carry bit

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>c</td>
<td>c'</td>
<td>z</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Concrete Example: Adding

\[
\begin{array}{c}
10 \\
11 \\
+ 10 \\
\hline
101
\end{array}
\]  
\[
\begin{array}{c}
c_1c_0 \\
x_1x_0 \\
+y_1y_0 \\
z_2z_1z_0
\end{array}
\]

- \( z_1 = (x_1 \text{ and not } y_1 \text{ and not } c_0) \text{ or } (\text{not } x_1 \text{ and } y_1 \text{ and not } c_0) \text{ or } (\text{not } x_1 \text{ and not } y_1 \text{ and } c_0) \text{ or } (x_1 \text{ and } y_1 \text{ and } c_0) \)

- \( c_1 = z_2 = (x_1 \text{ and } y_1 \text{ and not } c_0) \text{ or } (x_1 \text{ and not } y_1 \text{ and } c_0) \text{ or } (\text{not } x_1 \text{ and } y_1 \text{ and } c_0) \text{ or } (x_1 \text{ and } y_1 \text{ and } c_0) \)
Adding Bytes

- Computing $z_i$ and $c_i$ from $x_i$, $y_i$, and $c_{i-1}$ can be carried out with 6 ands, 3 ors, 4 nots.
- The previous slide uses 16 ands, 6 ors, and 9 nots (not as good).
- This operation is called a “full adder”.
- By chaining together one full adder per bit, we can make a circuit that adds any number of bits (4, 8, 16, 32, 64, etc.).
Hardware

- Any function we want to implement from bits to bits can be carried out by constructing the right circuit of and/or/nots.
- Creating a circuit solves the problem “in hardware”.
- The advantage of hardware solutions are that they are fast.
- The disadvantage is that they are inflexible.
Software

• The lovely thing about a computer is that the hardware does not have to change for the computer to change its behavior.

• A fixed set of circuits can actually change its behavior to represent any desired function!

• Build one, reprogram into anything.

• Disadvantage of the software approach: Can be much slower.
Key Insight

- Make a language for expressing operations.
- Complex enough to capture the important functions.
- Simple enough to be implementable in hardware.

Machine Language
Break it Down

- \( A = (A \text{ and not } (B \text{ and } C)) \text{ or } (\text{not } A \text{ and } (B \text{ and } C)) \)

<table>
<thead>
<tr>
<th>Equation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( acc = B )</td>
<td>( acc ) now holds “B and C”</td>
</tr>
<tr>
<td>( acc = acc \text{ and } C )</td>
<td>( acc ) now holds “B and C”</td>
</tr>
<tr>
<td>( E = acc )</td>
<td>( E ) now holds “B and C”</td>
</tr>
<tr>
<td>( acc = \text{not } A )</td>
<td>( acc ) now holds “not A and (B and C)”</td>
</tr>
<tr>
<td>( acc = acc \text{ and } E )</td>
<td>( acc ) now holds “not A and (B and C)”</td>
</tr>
<tr>
<td>( F = acc )</td>
<td>( F ) now holds “not A and (B and C)”</td>
</tr>
<tr>
<td>( acc = \text{not } E )</td>
<td>( acc ) now holds “not (B and C)”</td>
</tr>
<tr>
<td>( acc = acc \text{ and } A )</td>
<td>( acc ) now holds “A and not (B and C)”</td>
</tr>
<tr>
<td>( acc = acc \text{ or } F )</td>
<td>( acc ) now holds “(A and not (B and C)) or (not A and (B and C))”</td>
</tr>
<tr>
<td>( A = acc )</td>
<td>( A ) holds the new value of the equation</td>
</tr>
</tbody>
</table>
Instruction Set: 7 Bits

- V in 0000...1111 (variables A-P)
  - 000V: acc = acc or V
  - 001V: acc = acc and V
  - 010V: acc = V
  - 011V: acc = not V

- acc: special temporary variable
  - 100V: V = acc or V
  - 101V: V = acc and V
  - 110V: V = acc
  - 111V: V = not acc

0000 A 0010 C 0100 E 0110 G 1000 I 1010 K 1100 M 1110 O
0001 B 0011 D 0101 F 0111 H 1001 J 1011 L 1101 N 1111 P
Idea

- We write a little program that would perform the same function as the circuit.
- We make a circuit that can execute any program.
- Reduction!
Memory

• Need a place to store the various quantities we’re working with.

• Main memory is like a giant filing cabinet, where each drawer is numbered consecutively and can store one value.

• Need to be able to store and retrieve values.
Variables

• Let’s say we need to store 100 numbers.

• Can name them:
  - apple, asparagus, artichoke, apricot, banana, blueberry, blackberry, cantaloupe, ..., zucchini

• Tedious to assign names to them all.

```lua
set apple to 4
set blueberry to 21
say apple + blueberry for 2 secs
```
A List of Variables

- For convenience, if nothing else, use numbers to name the variables.
  - item 1 of var, item 2 of var, ..., item 100 of var.
Indirection

- Naming the variables with numbers gives us some additional power!

- Can use a variable to name another variable.
Persistence of Memory

- We can use this memory idea to store the Boolean variables (A-P).
- We can also use another set of memory locations to store the series of instructions to be executed (program).
- How are the instructions stored?
**Bits For One Instruction**

- **load/store (1 bit)**
  - 0: load; 1: store

- **instruction (2 bits)**
  - 00: acc or V
  - 01: acc and V
  - 10: acc (load)/V (store)
  - 11: not acc (load) / not V (store)

- **variable name (4 bits)**

  1011000
  - store = 1
  - instruction = 01
  - constant = 1000 = I
  - So, “I = acc and I”
Michael Littman’s Mini Logic Machine Language (ML³)

Program counter: which address’s instruction to process next

Contents (decimal):

Address                  Contents (binary)
0  51                     acc = acc and D
1  17                     acc = acc and B
2  111                     P = acc
3  49                     acc = not B
4  19                     acc = acc and D
5  15                     acc = acc or P
6 104                     l = acc
7  33                     acc = B
8  19                     acc = acc and D

Registers: Boolean variables and their values

Accumulator: Special register

Contents (instruction)
von Neumann Architecture

- A computer is just a big state machine.
- **Input**: registers, memory, input devices
- **Output**: new values for registers, memory, output devices
- **PC** = Program counter, the address of the statement to be executed.

Diagram:

```
<table>
<thead>
<tr>
<th>mem</th>
<th>acc</th>
<th>PC</th>
<th>reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>7x32 bits</td>
<td>5 bits</td>
<td>1 bit</td>
<td>1x16 bits</td>
</tr>
</tbody>
</table>
246 bits total
```

**CPU** = Central Processing Unit
von Neumann Architecture

- A computer is just a big state machine.
- **Input**: registers, memory, input devices
- **Output**: new values for registers, memory, output devices
- **PC** = Program counter, the address of the statement to be executed.

```
<table>
<thead>
<tr>
<th>mem</th>
<th>acc</th>
<th>PC</th>
<th>reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>7x32 bits</td>
<td>5 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 bit</td>
<td>1x16 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

CPU = Central Processing Unit

246 bits total
Cycle: A Whole Computer

memlookup32x7 → memaccPCreg

ir0 ir1 ir2 v → or and

not and

not and3

and3

ifthenelse

addbyte5

memlookup16x1 → val

memwrite16x1

more of the same...
Cycle: A Whole Computer

memlookup32x7

ir0, ir1, ir2, v

not, and

or, and

ifthenelse

memwrite16x1

addbyte5

memlookup16x1

ir0, ir1, ir2, v

not, and

ifthenelse

memwrite16x1

val

more of the same...
• ML$^3$ used a particular design that made it relatively easy to fit in a lecture slide while handling 2-bit addition.

• Computer manufacturers have different goals in mind: cost, speed, ease of running modern programs.

• Some quick examples:
## CARDiac (1968)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INP</td>
<td>Input – take a number from the input card and put it in a specified memory cell.</td>
</tr>
<tr>
<td>1</td>
<td>CLA</td>
<td>Clear and add – clear the accumulator and add the contents of a memory cell to the accumulator.</td>
</tr>
<tr>
<td>2</td>
<td>ADD</td>
<td>Add - add the contents of a memory cell to the accumulator.</td>
</tr>
<tr>
<td>3</td>
<td>TAC</td>
<td>Test accumulator contents – performs a sign test on the contents of the accumulator.</td>
</tr>
<tr>
<td>4</td>
<td>SFT</td>
<td>Shift – shifts the accumulator x places left, then y places right.</td>
</tr>
<tr>
<td>5</td>
<td>OUT</td>
<td>Output – take a number from the specified memory cell and write it on an output card.</td>
</tr>
<tr>
<td>6</td>
<td>STO</td>
<td>Store – copy the contents of the accumulator into a specified memory cell.</td>
</tr>
<tr>
<td>7</td>
<td>SUB</td>
<td>Subtract – subtract the contents of a specified memory cell from the accumulator.</td>
</tr>
<tr>
<td>8</td>
<td>JMP</td>
<td>Jump - jump to a specified memory cell.</td>
</tr>
<tr>
<td>9</td>
<td>HRS</td>
<td>Halt and reset – stop program execution, move bug to cell 00.</td>
</tr>
</tbody>
</table>