Chapter 3: Programming

CS105: Great Insights in Computer Science
Administrative

• Homework 2 due Wednesday!
Clickers Strike Back
Today’s Goals

• Cover 2.5 ideas

• First, idea of machine language, which we alluded to last time

• Second, idea of expression trees. which is yet another way of thinking about Boolean expressions

• Half of idea, thinking about computer memory and how it works
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.
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.
• Start small with a two bit memory circuit

• Each one has an 2-bit name (0-3) called its “address”.

• Each memory box can store a specific amount of bits. For example, 7-bits
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 is are the instructions stored?
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>acc</th>
<th>acc now holds</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc = B</td>
<td>“B”</td>
</tr>
<tr>
<td>acc = acc and C</td>
<td>“B and C”</td>
</tr>
<tr>
<td>E = acc</td>
<td>“B and C”</td>
</tr>
<tr>
<td>acc = not A</td>
<td>“not A”</td>
</tr>
<tr>
<td>acc = acc and E</td>
<td>“not A and (B and C)”</td>
</tr>
<tr>
<td>F = acc</td>
<td>“not A and (B and C)”</td>
</tr>
<tr>
<td>acc = not E</td>
<td>“not (B and C)”</td>
</tr>
<tr>
<td>acc = acc and A</td>
<td>“A and not (B and C)”</td>
</tr>
<tr>
<td>acc = acc or F</td>
<td>“(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 Bit

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

<table>
<thead>
<tr>
<th>b6 b5 b4 b3 b2 b1 b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>load/store (1 bit)</td>
</tr>
</tbody>
</table>

- 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”
Series of Instructions

Michael Littman’s Mini Logic Machine Language (ML3)

Program counter: which address’s instruction to process next

Registers: Boolean variables and their values

Accumulator: Special register

contents (decimal)

address

contents (binary)

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 of CPU components: mem, acc, PC, reg with bit counts (7x32, 5, 1, 1x16) and total 246 bits, CPU = Central Processing Unit]
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.
Cycle: A Whole Computer

memlookup32x7

ir0, ir1, ir2, v

not
and
not
and3
and3
ifthenelse
ifthenelse
ifthenelse
memwrite16x1

or and
memread16x1

addbyte5

val

memlookup16x1

more of the same...

mem acc PC reg

Instruction Sets

• 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:
x86: Intel’s Old Set

IN: Input Byte or Word From Port

IMUL: Signed Multiply
IDIV: Signed Integer Division
HLT: Halt CPU
ESC: Escape
ENTER: Make Stack Frame (80188+)

MUL: Unsigned Multiply
NEG: Two’s Complement Negation
NOP: No Operation (90h)
NOT: One’s Complement Negation (Logical NOT)
OR: Inclusive Logical OR
OUT: Output Data to Port
OUTS: Output String to Port (80188+)
POP: Pop Word off Stack
POPAP: Pop All Registers onto Stack (80188+)
POPFD: Pop Flags off Stack
POPF: Pop Word onto Stack
POPFH: Push Flags onto Stack
POPF: Push Flags onto Stack

RDTS: Read Task Switched Flag (286+ privileged)
RTS: Return From Procedure
RETF: Return From Procedure
RES: Reserved
RETF: Return From Procedure
RET: Return From Procedure
RET: Return From Procedure

XC: Extend Carry
XLC: Extend Carry
XCHG: Exchange
XCHG: Exchange

XOR: Exclusive OR
XLAT/XLATB: Translate
XOR: Exclusive OR

WBINVD: Write-Back and Invalidate Cache
WAIT/FWAIT: Event Wait
WINVD: Write-Back and Invalidate Cache
XCHG: Exchange

XLAT/XLATB: Translate
XOR: Exclusive OR
EX: Execute
EXIT: Exit Current Task
EDX: Extended Data X
EDX: Extended Data X

XCHG: Exchange
XOR: Exclusive OR
EX: Execute
EXIT: Exit Current Task
EDX: Extended Data X
EDX: Extended Data X
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A</td>
<td>Add</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>ADD A,(HL)</td>
<td>Add with carry from (HL)</td>
</tr>
<tr>
<td>ADD A,(IX+N)</td>
<td>Add with carry from (IX+N)</td>
</tr>
<tr>
<td>ADD A,(IY+N)</td>
<td>Add with carry from (IY+N)</td>
</tr>
<tr>
<td>Opcode</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>0</td>
<td>INP</td>
</tr>
<tr>
<td>1</td>
<td>CLA</td>
</tr>
<tr>
<td>2</td>
<td>ADD</td>
</tr>
<tr>
<td>3</td>
<td>TAC</td>
</tr>
<tr>
<td>4</td>
<td>SFT</td>
</tr>
<tr>
<td>5</td>
<td>OUT</td>
</tr>
<tr>
<td>6</td>
<td>STO</td>
</tr>
<tr>
<td>7</td>
<td>SUB</td>
</tr>
<tr>
<td>8</td>
<td>JMP</td>
</tr>
<tr>
<td>9</td>
<td>HRS</td>
</tr>
</tbody>
</table>
I’m now going to switch gears into a topic that bridges the gap between programs and circuits.

It will also give us a different view of logical expressions.

› Expression Trees
Expression Trees: Small

- True
- False
- True and False
- not True

Diagram:

```
   True
  /   
 not  
 /     
 True  False
```

```
  and
 /   
 True  False
```

```
Expression Trees: Small
- True
- False
- True and False
- not True

Diagram:

```
   True
  /   
 not  
 /     
 True  False
```

```
  and
 /   
 True  False
```
Expression Trees: Bigger

- True and True

```
  and
 /    \
True   True
```

- not True or False

```
  or
 /    \
 not   False
 /     \
 True
```
Expression Trees: Combined

- \((\text{True and True}) \text{ and } (\text{not True or False})\)
Trees and Subtrees

- Just as we can build more complex expressions out of simpler ones, we can build more complex trees out of simpler ones.

- An expression tree is:

```
ET
```

```
= True
```

```
False
```

```
not
```

```
and
```

```
or
```

True or False or not or and or or
Circular Definition?

- In a sense, this definition looks broken because it is defining an expression tree in terms of expression trees.

- This circularity is **safe**, because the definition also provides us a way to stop (True/False).

- It’s also **necessary**, because there’s an infinitely large set of possible expressions.
Some Tree Terminology

- **root**: The top node of the tree ("True", "False", "not", "and", or "or").
- **subtree**: A tree underneath the root.
- **left subtree**: The subtree to the left.
- **right subtree**: The subtree to the right.
- **leaf**: A tree with no subtrees of its own.
- **depth**: Number of nodes between the root and the farthest leaf.
Evaluate Bottom Up

- Evaluate a tree whenever all its subtrees are evaluated.

```
      and
     /   \
  and   or
  True  True
       /  \
     not False
         /  \
        True
```
Evaluate Bottom Up

- Evaluate a tree whenever all its subtrees are evaluated.
Evaluate Bottom Up

- Evaluate a tree whenever all its subtrees are evaluated.
Evaluate a tree whenever all its subtrees are evaluated.
• Evaluate a tree whenever all its subtrees are evaluated.
Evaluate Top Down

- Start at the root.
- Ask a friend to evaluate the subtrees.
- Do the root.
Evaluate Top Down

- Start at the root.
- Ask a friend to evaluate the subtrees.
- Do the root.
Evaluate Top Down

- Start at the root.
- Ask a friend to evaluate the subtrees.
- Do the root.
• Start at the root.
• Ask a friend to evaluate the subtrees.
• Do the root.
Not That Different, Really

• In a sense, you have to start at the bottom.

• But, what recursion (self delegation?) does is let you focus on what happens at the top and the lower-down stuff just works itself out.

• Can make for much cleaner code.
evaluateTree Function

- Takes a tree as input, returns True/False.
- In some sense, very literal!
- But, uses recursion to handle the messy lower-level stuff.
- Somehow, extremely natural and extremely mind bending.

```python
def evaluateTree(tree):
    if root(tree) == 'True': return True
    if root(tree) == 'False': return False
    if root(tree) == 'not':
        val = evaluateTree(subtree(tree))
        return not val
    if root(tree) == 'and':
        v1 = evaluateTree(leftSubtree(tree))
        v2 = evaluateTree(rightSubtree(tree))
        return v1 and v2
    if root(tree) == 'or':
        v1 = evaluateTree(leftSubtree(tree))
        v2 = evaluateTree(rightSubtree(tree))
        return v1 or v2
```
Expressions With Variables

- evaluateTree: Took an expression tree (with Trues and Falses) as input and returned its value.

- What about an expression with variables: “not A or B”?

- If we know A and B’s values, can substitute them in and use evaluateTree! Interpreter.
Expressions With Variables

- evaluateTree: Took an expression tree (with Trues and Falses) as input and returned its value.

- What about an expression with variables: “not A or B”? 

- If we know A and B’s values, can substitute them in and use evaluateTree! Interpreter.
• If A and B are not known, but we still want to do something useful, we can convert the expression tree into a program that, given A and B, produces the value of the expression!

or

\[
\begin{align*}
\text{not} & \\
\text{B} & \\
A & \\
\end{align*}
\]

⇒ acc = not A
acc = acc or B

{ answer in acc }