The Vector

- We can build (or at least imagine!) lots of circuits.
- We can even think about state machines that use circuits to do various things over time.
- We’re headed towards using these ideas to create a programmable computer.
- Hillis has us take a detour to talk about programming first.
Logo in Python

- Logo is a language invented to help people (kids?) learn to program.
- Python is the language we’ve been using for examples in this class.
- Hillis uses Logo for his examples in this chapter.
- I’ll translate them into Python so it fits better with our other examples.

Drawing Commands

```python
forward(10)
right(90)
forward(10)
```
def square():
    forward(10)
    right(90)
    forward(10)
    right(90)
    forward(10)
    right(90)
    forward(10)

Do It!
square()
**Command Using The New One**

```python
reset()
def window():
square()
square()
square()
square()
window()
reset()
square is a subroutine of window
```

**Command With Parameters**

```python
def square(size):
    forward(size)
    right(90)
    forward(size)
    right(90)
    forward(size)
    right(90)
    forward(size)
square(5)
```
def design():
square(5)
right(10)
design()
reset()
design()
...never returns

Recursion

def design(number):
square(5)
right(10)
if number == 1: return
else: design(number-1)
design(4)

Proper Recursion
def tree(size):
    if size < 1: return
    else:
        forward(size)
twoTrees(size/2.0)
        tree(size)
        back(size)
        left(45)

def twoTrees(size):
    left(45)
tree(size)
    right(90)
tree(size)
    left(45)
    left(90)
tree(6)

Recap

- Using logic gates, we know how to do a bunch of things with bits:
  - test equality
  - if-then-else gate
  - select one bit from a set (universal gate)
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...

- present von Neumann architecture
Consider Reprogramming

\[ A = (A \text{ and } \neg (B \text{ and } C)) \text{ or } (\neg A \text{ and } (B \text{ and } C)) \]

- Reprogramming the Christmas lights required that we create a bunch of circuits including the one above.
- It’s got 2 “not” gates, 4 “and” gates, and 1 “or” gate wired in a particular pattern.
- It would be nice if we didn’t have to play with gates and wires to change behavior.

Concrete Example: Adding

\[ \begin{array}{c}
10 \\
11 \\
+ 10 \\
101 \\
\end{array} \quad \begin{array}{c}
C_1C_0 \\
x_1x_0 \\
+y_1y_0 \\
z_2z_1z_0 \\
\end{array} \]

- 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.
- \( z_0 = (x_0 \text{ and } \neg y_0) \text{ or } (\neg x_0 \text{ and } y_0) \)
- \( c_0 = (x_0 \text{ and } y_0) \)
Concrete Example: Adding

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

- \( z_1 = (x_1 \text{ and not } y_1 \text{ and not } c_0) \text{ or } (\text{not } x_1 \text{ and } y_1 \text{ and not } c_0) \text{ or } (\text{not } x_1 \text{ and not } y_1 \text{ and } c_0) \text{ or } (x_1 \text{ and } y_1 \text{ and } c_0) \)
- \( c_1 = z_2 = (x_1 \text{ and } y_1 \text{ and not } c_0) \text{ or } (x_1 \text{ and not } y_1 \text{ and } c_0) \text{ or } (\text{not } x_1 \text{ and } y_1 \text{ and } c_0) \text{ or } (x_1 \text{ and } y_1 \text{ and } c_0) \)

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

- So, 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.
Programming an Adder

Circuit level:

Instruction level:

- \( H = A \oplus B = (A \text{ and } \neg B) \text{ or } (\neg A \text{ and } B) \)
- \( C_1 = (A \text{ and } B) \text{ or } (C \text{ and } H) \)
- \( S = H \oplus C = (H \text{ and } \neg C) \text{ or } (\neg H \text{ and } C) \)

• ug, need to simplify further... no accumulator
Simple Statements

• Still too many different statements.
• Break complex statements down into a set of simple statements.
• Instead of $E = (H \text{ and not } C) \text{ or (not } H \text{ and } C)$:
  • $\text{acc} = \text{not } C$
  • $\text{acc} = \text{acc and } H$
  • $P = \text{acc}$
  • $\text{acc} = \text{not } H$
  • $\text{acc} = \text{acc and } C$
  • $\text{acc} = \text{acc or } P$
  • $E = \text{acc}$

Instruction Set: 7 Bits

• $V$ in 0000...1111 (variables A- P)
  • 000V: $\text{acc} = \text{acc or } V$
  • 001V: $\text{acc} = \text{acc and } V$
  • 010V: $\text{acc} = V$
  • 011V: $\text{acc} = \text{not } V$
  • $\text{acc}$: special temporary variable
  • 100V: $V = \text{acc or } V$
  • 101V: $V = \text{acc and } V$
  • 110V: $V = \text{acc}$
  • 111V: $V = \text{not } \text{acc}$
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.

Memory Circuit

- We’ll have 32 memory locations, for instructions.
- Each one has a 5-bit name (0-31) called its “address”.
- Memory circuit takes the contents of memory (32 x 7 bits) and an address, 229 bits in all, & outputs the data stored at the corresponding address.
Memory Lookup

```python
def memlookup(add, mem):
    mem00 = mem[0]
    mem01 = ifthenelse7(equal5(intToByte(1), add), mem[1], mem00)
    mem31 = ifthenelse7(equal5(intToByte(31), add), mem[31], mem30)
    return mem31
```

Writing to Memory

- Similar circuit allows memory cells to be altered.
- `mem[address] = newval`
- If needed for future processing, copied back up at the end of the cycle.
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?

Python code:

```python
def memwrite(active, add, mem, val):
    return [if then else7 (active and equal5(intToByte(1), add), val, mem[7]),
            if then else7 (active and equal5(intToByte(2), add), val, mem[6]),
            if then else7 (active and equal5(intToByte(3), add), val, mem[5]),
            if then else7 (active and equal5(intToByte(4), add), val, mem[4]),
            if then else7 (active and equal5(intToByte(5), add), val, mem[3]),
            if then else7 (active and equal5(intToByte(6), add), val, mem[2]),
            if then else7 (active and equal5(intToByte(7), add), val, mem[1]),
            if then else7 (active and equal5(intToByte(8), add), val, mem[0])]
```
**One Instruction**

\[
\begin{array}{cccccccc}
\text{b6} & \text{b5} & \text{b4} & \text{b3} & \text{b2} & \text{b1} & \text{b0} \\
\end{array}
\]

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

- **variable name (4 bits)**
  - 1011000

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

- **store = 1**
- **instruction = 01**
- **constant = 1000 = I**
- **So, “I = acc and I”**

**Series of Instructions**

Program counter: which address’s instruction to process next

Accumulator: Special register

Michael Littman’s Mini Logic Machine Language (ML³)

Registers: Boolean variables and their values

Contents (binary)

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents (decimal)</th>
<th>Contents (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51</td>
<td>1110010010</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>0010010100</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>1010111111</td>
</tr>
<tr>
<td>3</td>
<td>49</td>
<td>1110001111</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>0010111111</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>1010011111</td>
</tr>
<tr>
<td>6</td>
<td>104</td>
<td>1100011111</td>
</tr>
<tr>
<td>7</td>
<td>83</td>
<td>1100011111</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>1101011111</td>
</tr>
</tbody>
</table>

acc = acc or D
acc = acc and B
P = acc
acc = not B
acc = acc or D
acc = acc and P
I = acc
acc = B
acc = acc or D
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.

Cycle: A Whole Computer

- **memlookup32x7**
- **addbyte5**
- **memlookup16x1**
- **memwrite16x1**
- **memacc PC reg**
- more of the same...

CPU = Central Processing Unit

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

```python
def cycle(input):
    [mem, pc, acc, reg] = input
    ir = memlookup32x7(pc, mem)
    pcnew = addbyte5(pc, intToByte5(1))
    v = ir[3:]
    val = memlookup16x1(v, reg)
    res = ifthenelse(not ir[1] and not ir[2], acc or val, False)
    res = ifthenelse(not ir[1] and ir[2], acc and val, res)
    res = ifthenelse(not ir[0] and ir[1] and not ir[2], acc, res)
    res = ifthenelse(not ir[0] and ir[1] and ir[2], not val, res)
    res = ifthenelse(ir[0] and ir[1] and ir[2], not acc, res)
    accnew = ifthenelse(ir[0], acc, res)
    regnew = memwrite16x1(ir[0], v, reg, res)
    return [mem, pcnew, accnew, regnew]
```

• DEMO

Instruction Sets

• ML³ 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>

---

### Evaluating Expression Trees

- I’m now going to switch gears into a topic that bridges the gap between subroutines and the hardware.
- It will also give us a different view of logical expressions.
- **Expression Trees**
Expression Trees: Small

- True
- False
- True and False
- not True

Expression Trees: Bigger

- True and True
- not True or False
Expression Trees: Combined

• (True and True) and (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:
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.

**Evaluate Top Down**

- 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.

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.

Tree ⇒ Program

• 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!

⇒ acc = not A
acc = acc or B

{ answer in acc }
Compiler

- An interpreter takes a program as input and makes it happen.
- A compiler takes a program as input and creates a machine-language program as output.
- A program that converts a program into a program—*twisted*, but useful!

Game Plan

- We’ll develop two schemes that compute the value of the expression.
  - One leaves the final value in “acc”.
  - The other leaves it in a variable in memory.
- Both are given a list of variables they can use for temporary storage.
- Mutually recursive (oooh).
**Code Generation: Var**

- Expression: A

  answer in acc
  acc = A

  answer in P
  acc = A
  P = acc

**Code Generation: not**

- Expression: not EXPR

  answer in acc (temp N)
  generate code for
  EXPR, answer in N
  acc = not N

  answer in P
  generate code for
  EXPR, answer in acc
  P = not acc

- Note: We keep a pool of temporary variables to use as needed (not just N).
Code Generation: or

- Expression: (EXPR$_1$ or EXPR$_2$)

answer in acc (temp N)
generate code for
EXPR$_1$, answer in N
generate code for
EXPR$_2$, answer in acc
acc = acc or N

answer in P
generate code for
EXPR$_1$, answer in P
generate code for
EXPR$_2$, answer in acc
P = acc or P

- Note: “and” is handled the same way.

Example Expression

- Expression: not A or B

answer in acc
generate code for
not A, answer in N

answer in P
generate code for
not A, answer in P

generate code for
B, answer in acc
acc = acc or N

P = acc or P
**Assembler**

- An assembler handles the last little step of translating the series of instructions to a series of numbers.

<table>
<thead>
<tr>
<th>answer in acc</th>
<th>answer in P</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc = A</td>
<td>acc = A</td>
</tr>
<tr>
<td>N = not acc</td>
<td>P = not acc</td>
</tr>
<tr>
<td>acc = B</td>
<td>acc = B</td>
</tr>
<tr>
<td>acc = acc or N</td>
<td>P = acc or P</td>
</tr>
</tbody>
</table>

```
def compileToVar(tree, temps, target):
    if root(tree) == 'V': return [LOAD(ACC(VAR(tree)))] + [STORE(ACC(target))]
    if root(tree) == 'not':
        return compileToAcc(subtree(tree), temps + [target]) + [STORE(NOT(target))]
    if root(tree) == 'or':
        finalCmd = STORE(OR(temps[0]))
    if root(tree) == 'and':
        finalCmd = STORE(AND(temps[0]))
    return compileToVar(leftSubtree(tree), temps, target) + compileToAcc(rightSubtree(tree), temps) + [finalCmd]

def compileToAcc(tree, temps):
    if root(tree) == 'V': return [LOAD(ACC(VAR(tree)))]
    assert(len(temps) > 0)
    if root(tree) == 'not':
        return compileToVar(subtree(tree), temps[1:], temps[0]) + [LOAD(NOT(temps[0]))]
    if root(tree) == 'or':
        finalCmd = LOAD(OR(temps[0]))
    if root(tree) == 'and':
        finalCmd = LOAD(AND(temps[0]))
    return compileToVar(leftSubtree(tree), temps[1:], temps[0]) + compileToAcc(rightSubtree(tree), temps) + [finalCmd]
```
Inefficiency

• (not A or B)

• Automatically generated machine code:

```
answer in acc
acc = A 32
N = not acc 125
acc = B 33
acc = acc or N 13
```

• By hand:

```
answer in acc
acc = not A 48
acc = acc or B 1
```

• Often more than one way to do it!

Optimizations

• Many ways of speeding up compiled code have been developed.

• Want to minimize steps, temporary variables.

• I’ll describe two important ones:
  - Shared subexpressions
  - Logical equivalence
Automatic Code (13 inst.)

- $E = ((A \land B) \land C) \lor ((A \land B) \land D)$
  - $E = (A \land B) \land C$
  - $E = A \land B$
  - acc = A
  - $E = acc$
  - acc = B
  - $E = acc \land E$
  - acc = C
  - $E = acc \land E$
  - acc = (A \land B) \land D$
  - $N = A \land B$
  - acc = A
  - $N = acc$
  - acc = B
  - $N = acc \land N$
  - acc = D
  - $N = acc \land N$
  - $E = acc \lor E$
  - 1 temps, 13 instructions

Shared Subexpression

- $E = ((A \land B) \land C) \lor ((A \land B) \land D)$
  - $N = A \land B$
  - $N = A$
  - acc = A
  - $N = acc$
  - acc = B
  - $N = acc \land N$
  - $E = (N \land C) \lor (N \land D)$
  - $E = N \land C$
  - acc = N
  - $E = acc$
  - acc = C
  - $E = acc \land N$
  - $E = E \land acc$
  - $acc = N \land D$
  - acc = N
  - O = acc
  - acc = D
  - $acc = acc \land O$
  - $E = E \lor acc$
  - 2 temps, 13 instructions
Logical Equivalence

- $E = ((A \text{ and } B) \text{ and } C) \text{ or } ((A \text{ and } B) \text{ and } D)$
- $E = (A \text{ and } B) \text{ and } (C \text{ or } D)$
- $E = A \text{ and } B$
- acc = A
- $E = acc$
- acc = B
- $E = E \text{ and } acc$
- acc = C or D
- acc = C
- acc = D
- N = acc
- acc = D
- acc = acc or N
- $E = E \text{ and } acc$

1 temps, 9 instructions

A Compiler

- A program that translates computer programs that people write into a machine language instructions for the computer to execute.
- (Adapted from notes by Barbara Ryder.)
Parser

- Programs are written in a high-level language such as Java or C++
  - A grammar description of the programming language describes a well-formed program
- Parsers check that a program adheres to the rules of the programming language’s grammar.
- If so, parser translates the program into an internal representation used by the compiler

Code Generator

- Translates the internal representation of a program into a specific machine language.
- Has all the info it needs in the internal representation and knows the program is correct according to the rules of the grammar.
- Can change to a different computer chip with a different instruction set by changing code generators, without other changes to the compiler.
Summing Up

- Parser uses grammar rules to check expressions for correct structure -- syntax
- If correct, then builds the expression graphs
- Optimizes the graphs to find repeated subexpressions and constants that can be evaluated at compile-time
- Then generates code from the graph

Interpreters

- Compiler translates program to machine lang.
- Interpreter translates statements by executing equivalent commands
  - No real translation step
- Interpretation requires programming language have a defined meaning for its statements---semantics
  - Sometimes defined mathematically, sometimes in English.
How Are They Useful?

• Allow prototyping of new langs.
  - Get to test out quickly (e.g., Scheme, Prolog, Java).

• Achieves portability/universality for a PL
  - Code to be interpreted by a Virtual Machine (VM)
  - Can install the PL on a different machine (i.e., chip) merely by rewriting the VM
  - As long as spec is careful (syntax/semantics), programs should work equivalently!
  - Model for Java (e.g., JVM - Java Virtual Machine)

Java

• Language definition ~mid-1990’s

• Used to write applications built out of pieces (e.g., libraries, components, middleware)
  - Built by different people, in different places, on different machines
  - Works because of VM mechanism

• Interpretation frees user from worries about machine-dependent translation details.
Some History of Langs.

- 1950’s
  - Machine language programming
  - Scientific computation in Fortran with first compilers
  - LISP for non-numerical computation
- 1960’s
  - First optimizing Fortran compiler (IBM)
- 1970’s
  - First program analyses enable complex optimizations
  - C language and UNIX (Linux is a form of UNIX)
  - Optimizing for space and time savings
Some History of Langs.

• 1980’s
  - First widely-used object-oriented langs. - Smalltalk, C++
  - Compilers translate for parallel computers (e.g., Connection Machine, Cray)
  - Langs. allowing explicit parallelism (i.e., use of multiple processors; Ada)

• 1990’s
  - Birth of the Internet
  - PLs for explicitly distributed computation (e.g., across machines in an network)
  - Object-oriented langs. - Java (VMs)

• 2000’s
  - Compiling for low power
  - Special purpose (domain specific) langs
  - Scalability, distributed computation, ubiquity
Recap: Reduction

<table>
<thead>
<tr>
<th>Level</th>
<th>Examples</th>
<th>Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>software libraries</td>
<td>graphics, animation,</td>
<td>networking, security,</td>
</tr>
<tr>
<td></td>
<td>robotics</td>
<td>mathematics</td>
</tr>
<tr>
<td>high-level language</td>
<td>Python</td>
<td>C, Java, C++, Logo, LISP,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fortran, ML</td>
</tr>
<tr>
<td>machine language</td>
<td>ML³</td>
<td>x86, CARDIAC, Z80</td>
</tr>
<tr>
<td>logic gates</td>
<td>equal, ifthenelse, add</td>
<td>memlookup, memwrite</td>
</tr>
<tr>
<td>basic logic gates</td>
<td>and, or, not</td>
<td>nor, nand, xor</td>
</tr>
<tr>
<td>physical bits</td>
<td>0,1 via high/low voltage</td>
<td>water pressure, kinetic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>energy</td>
</tr>
</tbody>
</table>

Does It End There?

- Of course, not.
- We can continue to build sophisticated programs out of simpler programs.
- The idea of **subroutines** (procedures, functions) makes this work.
Subroutines

- A lot of research in computer science is about designing and creating just the right set of subroutines, sometimes called libraries.
- You don’t have enough background yet to weigh in on these problems.
- But, there is an analogous set of problems where you are already an expert...

Sing-A-Long Programs

Gilligan’s Island Theme

Just sit right back and you'll hear a tale, a tale of a fateful trip.
That started from this tropic port, aboard this tiny ship.
The mate was a mighty sailin' man, the skipper brave and sure.
Five passengers set sail that day, for a three hour tour, a three hour tour...
The weather started getting rough, the tiny ship was tossed.
If not for the courage of the fearless crew, the Minnow would be lost; the Minnow would be lost.
The ship took ground on the shore of this uncharted desert isle, with Gilligan, the Skipper too, the Millionaire, and his Wife, the Movie Star, the Professor and Mary Ann, here on Gilligan's Isle.
def GilligansIsland():
    print "Just sit right back and you'll hear a tale, a tale of a fateful trip."
    print "That started from this tropic port, aboard this tiny ship."
    print "The mate was a mighty sailin' man, the skipper brave and sure."
    print "Five passengers set sail that day, for a three hour tour, a three hour tour."
    print "The weather started getting rough, the tiny ship was tossed."
    print "If not for the courage of the fearless crew,"
    print "The ship took ground on the shore of this uncharted desert isle,"
    print "with Gilligan, the Skipper too, the Millionaire, and his Wife,"
    print "the Movie Star, the Professor and Mary Ann, here on Gilligan's Isle."

GilligansIsland()
Using A Subroutine

- Now, “ClementineChorus()” is its own little program, or a new statement in the language, that we can call whenever we need it.
- Calling the subroutine does all the steps, then the program resumes from where the call took place.
- Simplifies the program, easier to maintain, extend, fix.
Shared Structure

**All My Loving**
Lennon/McCartney

Close your eyes and I'll kiss you,
Tomorrow I'll miss you,
Remember I'll always be true,
And then while I'm away,
I'll write home every day,
And I'll send all my loving to you.

I'll pretend I am kissing,
The lips I am missing,
And hope that my dreams will come true,
And then while I'm away,
I'll write home every day,
All my loving, I will send to you,
All my loving, darling, I'll be true.

Close your eyes and I'll kiss you,
Tomorrow I'll miss you,
Remember I'll always be true,
And then while I'm away,
I'll write home every day,
And I'll send all my loving to you.

All my loving, I will send to you,
All my loving, darling, I'll be true,
All my loving. All my loving

```
def closeYourEyes():
  print "Close your eyes and I'll kiss you"
  print "Tomorrow I'll miss you"
  print "Remember I'll always be true"
  print "And then while I'm away"
  print "I'll write home every day"
  print "And I'll send all my loving to you"
```

```
def allMyLoving():
  closeYourEyes()
  print "All my loving, I will send to you"
  print "All my loving, All my loving"
  print "Woo, all my loving, I will send to you"
```

Deeper Nesting?

This subroutine called its own subroutine.

```
def Away():
  print "And then while I'm away"
  print "I'll write home every day"
  print "And I'll send all my loving to you"
```

```
def closeYourEyes():
  print "Close your eyes and I'll kiss you"
  print "Tomorrow I'll miss you"
  print "Remember I'll always be true"
  print "And then while I'm away"
  print "I'll write home every day"
  print "And I'll send all my loving to you"
```

```
def allMyLoving():
  closeYourEyes()
  print "All my loving, I will send to you"
  print "All my loving, darling I'll be true"
  print "Woo, all my loving. All my loving"
  print "Close your eyes and I'll kiss you"
```

What subroutines would you define?
**More Structure**

**Eight Days a Week (Beatles)**

- **B**
  - Ooh I need your love babe, guess you know it's true, Hope you need my love babe just like I need you,

- **R**
  - Hold me, love me, Hold me, love, Ain't got nothin' but love babe, Eight days a week,

- **Y**
  - Love you ev'ry day girl, always on my mind, One thing I can say girl, love you all the time,

- **G**
  - Eight days a week I love you, Eight days a week is not enough to show I care,

**Python Code**

```python
def B():
    print "Ooh I need your love babe"
    print "Guess you know it's true"
    print "Hope you need my love babe" print "Just like I need you"

def R():
    print "Hold me, love me"
    print "Hold me, love me"
    print "Ain't got nothin' but love babe"
    print "Eight days a week"

def Y():
    print "Love you every day girl"
    print "Always on my mind"
    print "One thing I can say girl"
    print "Love you all the time"

def G():
    print "Eight days a week"
    print "I love you"
    print "Eight days a week"
    print "Is not enough to show I care"

def eightDaysAWeek():
    B(); R(); Y(); R(); G()
    B(); R(); G(); Y(); R()
    print "Eight days a week"
    print "Eight days a week"
```
Old Macdonald had a farm, E-I-E-I-O
And on his farm he had a cow, E-I-E-I-O
With a "moo-moo" here and a "moo-moo" there
Here a "moo" there a "moo"
Everywhere a "moo-moo"
Old Macdonald had a farm, E-I-E-I-O
Old Macdonald had a farm, E-I-E-I-O
And on his farm he had a pig, E-I-E-I-O
With a (snort) here and a (snort) there
Here a (snort) there a (snort)
Everywhere a (snort-snort)
With a "moo-moo" here and a "moo-moo" there
Here a "moo" there a "moo"
Everywhere a "moo-moo"
Old Macdonald had a farm, E-I-E-I-O
Old Macdonald had a farm, E-I-E-I-O
And on his farm he had a horse, E-I-E-I-O
With a "neigh, neigh" here and a "neigh, neigh" there
Here a "neigh" there a "neigh"
Everywhere a "neigh-neigh"
With a (snort) here and a (snort) there
Here a (snort) there a (snort)
Everywhere a (snort-snort)
With a "moo-moo" here and a "moo-moo" there
Here a "moo" there a "moo"
Everywhere a "moo-moo"
Old Macdonald had a farm, E-I-E-I-O
Old Macdonald had a farm, E-I-E-I-O
And on his farm he had a chick, E-I-E-I-O
With a "cluck, cluck" here and a "cluck, cluck" there
Here a "cluck" there a "cluck"
Everywhere a "cluck-cluck"
With a "neigh, neigh" here and a "neigh, neigh" there
Here a "neigh" there a "neigh"
Everywhere a "neigh-neigh"
With a (snort) here and a (snort) there
Here a (snort) there a (snort)
Everywhere a (snort-snort)
With a "moo-moo" here and a "moo-moo" there
Here a "moo" there a "moo"
Everywhere a "moo-moo"
Old Macdonald had a farm, E-I-E-I-O
But Notice...

• Each time we “snort”, we follow it with “moo”.
• Each time we “neigh”, we follow it with “snort”.
• Each time we “cluck”, we follow it with “neigh”.
• Let’s move the earlier noises into the later subroutines...

Macdonald #2

def OldMacdonald():
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print 'And on his farm he had a cow, E-I-E-I-O'
    moo()
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print ''
    print 'Old Macdonald had a farm, E-I-E-I-O'
    neigh()
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print 'And on his farm he had a pig, E-I-E-I-O'
    snort()
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print ''
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print 'And on his farm he had a horse, E-I-E-I-O'
    neigh()
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print 'And on his farm he had a chick, E-I-E-I-O'
    cluck()
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print ''

def moo():
    print 'With a “moo-moo” here and a “moo-moo” there'
    print 'Here a “moo” there a “moo”'
    print 'Everywhere a “moo-moo”'

def snort():
    print 'With a (snort) here and a (snort) there'
    print 'Here a (snort) there a (snort)'
    print 'Everywhere a (snort-snort)'
    moo()

def neigh():
    print 'With a “neigh, neigh” here and a “neigh, neigh” there'
    print 'Here a “neigh” there a “neigh”'
    print 'Everywhere a “neigh-neigh”'
    snort()

def cluck():
    print 'With a “cluck, cluck” here and a “cluck, cluck” there'
    print 'Here a “cluck” there a “cluck”'
    print 'Everywhere a “cluck-cluck”'
    neigh()
def OldMacdonald():
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print 'And on his farm he had a cow, E-I-E-I-O''
    cluck()
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print ''

def neigh():
    print 'With a "neigh-neigh" there'
    print 'Here a "neigh" there a "neigh"'
    print 'Everywhere a "neigh-neigh"'
    snort()

def snort():
    print 'With a (snort) here and a (snort) there'
    print 'Here a (snort) there a (snort)'
    print 'Everywhere a (snort-snort)'

def moo():
    print 'With a "moo-moo" here'
    print '  and a "moo-moo" there'
    print 'Here a "moo" there a "moo"'
    print 'Everywhere a "moo-moo"'

    def moo():
        print 'With a "moo-moo" here'
        print '  and a "moo-moo" there'
        print 'Here a "moo" there a "moo"'
        print 'Everywhere a "moo-moo"'
Subroutine Stack

def OldMacdonald():
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print 'And on his farm he had a cow, E-I-E-I-O'
    moo()
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print ''
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print 'And on his farm he had a pig, E-I-E-I-O'
    snort()
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print ''
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print 'And on his farm he had a horse, E-I-E-I-O'
    neigh()
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print ''
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print 'And on his farm he had a chick, E-I-E-I-O'
    cluck()
    print 'Old Macdonald had a farm, E-I-E-I-O'
    print ''

Subroutines So Far

• Can be used to gather up repeated code in one place.
• Can be deeply nested to capture structure when necessary.
• Next, parameters greatly increase the power of the subroutine idea...
Near Miss...

Magical Mystery Tour (Beatles)

Roll up, roll up for the mystery tour.
Roll up, roll up for the mystery tour.
Roll up, roll up for the mystery tour.
Roll up, roll up for the mystery tour.
The magical mystery tour is ______ to
take you away, ______ to take you away.

Roll up, roll up for the mystery tour.
Roll up, roll up for the mystery tour.
Roll up, roll up for the mystery tour.
Roll up, roll up for the mystery tour.
The magical mystery tour is ______ to
take you away, ______ to take you away.

• Verse structure is nearly repetitive. Can’t quite define a single subroutine that covers all three.

• If we could fill in the blank, we could reuse the same routine...

Naive Python Code

def mysteryVerseWaiting():
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "The magical mystery tour is ______ to take you away, ______ to take you away."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "The magical mystery tour is ______ to take you away, ______ to take you away."
    print

def mysteryVerseComing():
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "The magical mystery tour is ______ to take you away, ______ to take you away."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "The magical mystery tour is ______ to take you away, ______ to take you away."
    print

def mysteryVerseHoping():
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "The magical mystery tour is ______ to take you away, ______ to take you away."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "The magical mystery tour is ______ to take you away, ______ to take you away."
    print

def magicalMysteryTour():
    mysteryVerseWaiting()
    mysteryVerseHoping()
    mysteryVerseComing()
Formal Parameters

```python
def mysteryVerse(x):
    print "Roll up, roll up for the mystery tour."
    print "Roll up, roll up for the mystery tour."
    print "The magical mystery tour is "
    print " + x + " to take you away, "
    print "Take you today."
```

- When `mysteryVerse("waiting")` is executed, the caller temporarily assigns the variable "x" the value "waiting".
- The subroutine does its thing, using the value “waiting” whenever “x” appears. When finished, it returns to the caller, and “x” is forgotten.
- x is called the formal parameter, “waiting” is the actual parameter.

Macdonald #3

```python
def OldMacdonald():
    moo(True)
    snort(True)
    neigh(True)
    cluck(True)

def doIntro(animal, noise, intro):
    if intro:
        print 'Old Macdonald had a farm, E-I-E-I-O'
        print 'And on his farm he had a ' + animal + ', E-I-E-I-O'
        makeNoise(noise)
    if intro:
        print 'Old Macdonald had a farm, E-I-E-I-O'
        print '
def makeNoise(noise):
    print 'With a ' + noise + ' here and a '
    print ' + noise + ' there'
    print 'Here a ' + noise + ' there a ' + noise + ' Everywhere a ' + noise + ' there'
```
def makeNoise(noise):
    print 'With a '' + noise + '-' + noise + ' here and a '
    print 'Here a '' + noise + '-' + noise + ' there'
    print 'Here a '' + noise + ' there a '' + noise + '
    print 'Everywhere a '' + noise + '-' + noise + '"

def doIntro(animal, noise, intro):
    if intro:
        print 'Old Macdonald had a farm, E-I-E-I-O'
        print 'And on his farm he had a ' + animal + ', E-I-E-I-O'
        makeNoise(noise)
    if intro:
        print 'Old Macdonald had a farm, E-I-E-I-O'
        print "

The Verses

def moo(intro):
    dolIntro("cow", "moo", intro)

def snort(intro):
    dolIntro("pig", "(snort)", intro)
    moo(False)

def neigh(intro):
    dolIntro("horse", "neigh", intro)
    snort(False)

def cluck(intro):
    dolIntro("chick", "cluck", intro)
    neigh(False)

• Writes out the “here-a-there-a” part with the given noise.

• If “intro” is False, just makes the given animal noise. If “intro” is True, also includes that animal’s introduction (“And on this farm...”).
The Song

def OldMacdonald():
    moo(True)
    snort(True)
    neigh(True)
    cluck(True)

• Sings all four complete verses.

• Important thing to notice:
  • `snort(True)` means “set intro to True and begin executing the
    snort subroutine”.
  • While executing `snort(True)`, it makes a call to `moo(False)`
  • `moo(False)` means “set intro to False and begin executing
    the moo subroutine.
  • No confusion---each subroutine has its own copy of “intro”!

Subroutine Stack

def moo(intro = False):
    doIntro("cow", "moo", intro)

def snort:intro = True):
    doIntro("pig", "snort", intro)

    moo(False)
Local Variables

• Unless otherwise specified, variable names refer to their values within the subroutine itself only.
• Changes to these local variables do not impact any of the calling subroutines.
  • “What happens in LV stays in LV.”

Recursion...

• Since each subroutine call has its own context (local variables, “red arrow”), a subroutine can call itself.
• This is called recursion.
• First, some song examples, then more general...
Recursive Algorithm

def bottlesOfBeer(n):
    if n > 0:
        print str(n) + " bottles of beer on the wall."
        print str(n) + " bottles of beer."
        print "If one of those bottles should happen to fall."  
        print str(n-1) + " bottles of beer on the wall."
        print 
bottlesOfBeer(n-1)

• bottlesOfBeer(99) prints a message, then calls
• bottlesOfBeer(98), which prints a message, then calls
• bottlesOfBeer(97), which prints a message, then calls...
• bottlesOfBeer(0), which does nothing.

Everyday Recursion

• Some people think recursion is scary, but it’s all around us: language, nature, toys, music, family trees, mathematical expressions.

• If you like self-reference, you’ll love recursion!
def songNeverEnds():
    print "This is 'The Song That Doesn't End"
    print "Yes it goes on, and on my friend"
    print "Some people, started singing it"
    print "not knowing what it was"
    print "And they’ll just keep on singing it"
    print "forever just because…"
    songNeverEnds()

• Featured in “Lambchop’s Play-along”
• Canceled.