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...
Christmas Light Architecture

Von Neumann Architecture
Consider Reprogramming

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

- 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.

Circuit, Visualized

- \[ A = \text{(A and not (B and C)) or (not A and (B and C))} \]
- Separate circuits for B and C.
- Ready for blinking!
**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

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

| acc = B     | acc now holds “B”     |
| acc = acc and C | acc now holds “B and C” |
| E = acc     | E now holds “B and C”  |
| acc = not A | acc now holds “not A”  |
| acc = acc and E | acc now holds “not A and (B and C)” |
| F = acc     | F now holds “not A and (B and C)” |
| acc = not E | acc now holds “not (B and C)” |
| acc = acc and A | acc now holds “A and not (B and C)” |
| acc = acc or F | acc now holds “(A and not (B and C)) or (not A and (B and C))” |
| A = acc     | A holds the new value of the equation |
### 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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>O</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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.

Memory Idea

• Like special numbered variables.
  ‣ mem[0] is a variable
  ‣ mem[1] is another
  ‣ mem[31] is the last one

• Useful, because we can use one variable to select another.
  ‣ if i is between 0 and 31, mem[i] refers to a specific variable, different depending on i.
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.

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 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)
- constant = 1000 = I
- So, “I = acc and I”
**Series of Instructions**

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents (decimal)</th>
<th>Contents (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>acc = not D</td>
<td>01100110</td>
</tr>
<tr>
<td>17</td>
<td>acc = acc and B</td>
<td>00101000</td>
</tr>
<tr>
<td>2</td>
<td>acc = P</td>
<td>11011100</td>
</tr>
<tr>
<td>49</td>
<td>acc = not B</td>
<td>01100001</td>
</tr>
<tr>
<td>19</td>
<td>acc = acc and D</td>
<td>00100111</td>
</tr>
<tr>
<td>15</td>
<td>acc = acc or P</td>
<td>00011111</td>
</tr>
<tr>
<td>104</td>
<td>I = acc</td>
<td>11101000</td>
</tr>
<tr>
<td>83</td>
<td>acc = B</td>
<td>01000011</td>
</tr>
<tr>
<td>19</td>
<td>acc = acc and D</td>
<td>00100111</td>
</tr>
</tbody>
</table>

Program counter: which address’s instruction to process next

Registers: Boolean variables and their values

Accumulator: Special register

---

**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

more of the same...

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:
### Opcode Mnemonic Description

<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 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

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:

```
ET =
True or False or not or and or or
```

```
ET
```

```
ET
```

```
ET
```

```
ET
```

```
ET
```
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.

```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.

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!

```plaintext
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: (EXPR₁ or EXPR₂)

answer in acc (temp N)
generate code for
EXPR₁, answer in N
generate code for
EXPR₂, answer in acc
acc = acc or N

answer in P
generate code for
EXPR₁, answer in P
generate code for
EXPR₂, 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

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

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

generate code for
B, answer in acc
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>32</td>
<td>32</td>
</tr>
<tr>
<td>N = not acc</td>
<td>P = not acc</td>
</tr>
<tr>
<td>125</td>
<td>127</td>
</tr>
<tr>
<td>acc = B</td>
<td>acc = B</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>acc = acc or N</td>
<td>P = acc or P</td>
</tr>
<tr>
<td>13</td>
<td>79</td>
</tr>
</tbody>
</table>

Complete Code

```python
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.)**

- \( E = ((A \text{ and } B) \text{ and } C) \text{ or } ((A \text{ and } B) \text{ and } D) \)
  - \( E = (A \text{ and } B) \text{ and } C \)
    - \( E = A \text{ and } B \)
      - acc = A
      - E = acc
      - acc = B
      - E = acc and E
    - acc = C
    - E = acc and E
  - \( N = A \text{ and } B \)
    - \( N = A \)
      - acc = A
      - \( N = acc \)
      - acc = B
      - \( N = acc \text{ and } acc \)
    - \( N = D \)
      - \( N = acc \text{ and } N \)
      - acc = D
      - acc = acc and O
      - \( E = acc \text{ or } E \)
  - acc = (A and B) and D
    - \( N = A \text{ and } B \)
      - acc = A
      - \( N = acc \)
      - acc = B
      - \( N = acc \text{ and } N \)
      - acc = D
      - acc = acc and O
      - \( E = acc \text{ or } E \)
  - 1 temps, 13 instructions

**Shared Subexpression**

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

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, robotics</td>
<td>networking, security, mathematics</td>
</tr>
<tr>
<td>high-level language</td>
<td>Python</td>
<td>C, Java, C++, Logo, LISP, 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 energy</td>
</tr>
</tbody>
</table>