Happy Friday!!

Number Systems (in Binary)

Lab 1 part 2 due Sunday
CATME survey due Sunday
No lab or discussion section next week
lab 2 part 1 is due Thursday
office hours during scheduled section times
Stored state bits can be interpreted with many different encoding schemes

Computer can do 2 things
1) Store state (How do we interpret stored bits?)
2) Manipulate state
The class consists roughly of 4 quarters: (Bolded words are the big ideas of the course, pay attention when you hear these words)

1. You will build a simple computer processor. **Build and create state machines with** **data**, **control**, and **indirection**
2. You will learn how high-level language code executes on a processor. **Time limitations create dependencies in the state of the processor**
3. You will learn why computers perform the way they do. **Physical limitations require locality and indirection in how we access state**
4. You will learn about hardware mechanisms for parallelism. **Locality, dependencies, and indirection** on performance enhancing drugs

- We will have a SPIMbot contest!
Today’s lecture

- Representing things with bits
  - $N$ bits gets you $2^N$ representations
- Unsigned binary number representation
  - Converting between binary and decimal
  - Hexadecimal notation
- Binary Addition & Bitwise Logical Operations
  - Every operation has a width
- Two’s complement signed binary representation
A code maps each fixed-width string of bits to a meaning

(like a secret decoder ring...)

<table>
<thead>
<tr>
<th>Bit pattern</th>
<th>Marine Mammal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100101</td>
<td>Humpback Whale</td>
</tr>
<tr>
<td>0100110</td>
<td>Leopard Seal</td>
</tr>
<tr>
<td>0100111</td>
<td>Sea Otter</td>
</tr>
<tr>
<td>0101000</td>
<td>West Indian Manatee</td>
</tr>
<tr>
<td>0101001</td>
<td>Bottlenose Dolphin</td>
</tr>
</tbody>
</table>

- This mapping however is rarely stored explicitly
  - Rather it is used when we interpret the bits.
How many bits to encode N possible things?

- 1 bit can encode 2 possibilities (0, 1)

\[ N \]

Bits = 1

\[ \frac{1}{N} \]

# encodings = 2

\[ 2 \]

\[ 2^2 \]

\[ 2^3 = 8 \]

\[ N \]

\[ 2 \]
What is the minimum # of bits to encode?

- One of the U.S.’s 50 states?

  a) $3 = 2^3 = 8$
  b) $4 = 2^4 = 16$
  c) $5 = 2^5 = 32$
  d) $6 = 2^6 = 64$
  e) 7
How many bits to encode?

- The list of Justin Bieber’s good songs?

  a) 0
  b) 0
  c) 0
  d) 0
  e) 0
Unsigned numbers are the set of non-negative numbers

- 0, 1, 2, 3, 4, 5, ...

- N bits \(\Rightarrow\) store \(2^N\) unsigned numbers \(\Rightarrow\) what range should the bits encode?
  - 3 bits \(\Rightarrow\) 8 representations \(\Rightarrow\) 0 to 7? 1 to 8? 32-40?
  - 8 bits \(\Rightarrow\) 256 representations \(\Rightarrow\) 0 to 255? 1 to 256? 1024 to 1280?
How does decimal representation work?

Numbers consist of a bunch of digits, each with a weight:

1  6  2  .  3  7  5  Digits
100 10 1  1/10  1/100  1/1000 Weights

All weights are powers of the base, which is 10:

1  6  2  .  3  7  5  Digits
10^2 10^1 10^0 = 1  10^-1  10^-2  10^-3 Weights

To find the decimal value of a number, multiply each digit by its weight and sum the products.

(1 x 10^2) + (6 x 10^1) + (2 x 10^0) + (3 x 10^-1) + (7 x 10^-2) + (5 x 10^-3) = 162.375

Consider 162.375
Unsigned binary number representation uses a position-weighted encoding scheme

- The weights are powers of 2.
- For example, here is 1101 in binary:

  \[
  \begin{array}{cccc}
  1 & 1 & 0 & 1 \\
  2^3 & 2^2 & 2^1 & 2^0 \\
  \end{array}
  \]

  Binary digits, or bits

  Weights (in base 10)

- The decimal value is:

  \[(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13\]

  Powers of 2:

  \[
  \begin{array}{ccc}
  2^0 = 1 & 2^4 = 16 & 2^8 = 256 \\
  2^1 = 2 & 2^5 = 32 & 2^9 = 512 \\
  2^2 = 4 & 2^6 = 64 & 2^{10} = 1024 \\
  2^3 = 8 & 2^7 = 128 & \phantom{2^{10}}
  \end{array}
  \]
Binary to Decimal

What is the 5-bit unsigned number 01010 in decimal?

a) 2
b) 5
c) 10
d) 12
e) 18

Powers of 2:

| $2^0$ = 1 | $2^4$ = 16 | $2^8$ = 256 |
| $2^1$ = 2 | $2^5$ = 32 | $2^9$ = 512 |
| $2^2$ = 4 | $2^6$ = 64 | $2^{10}$ = 1024 |
| $2^3$ = 8 | $2^7$ = 128 |
Fractional binary numbers use the same pattern as integer binary numbers

- For example, here is 1101.01 in binary:

  \[
  \begin{array}{ccccccccc}
  1 & 1 & 0 & 1 & . & 0 & 1 \\
  2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2}
  \end{array}
  \]

  Binary digits, or bits
  Weights (in base 10)

- The decimal value is:

  \[
  (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) = \\
  8 + 4 + 0 + 1 + 0.25 = 13.25
  \]
An algorithm for converting decimal to binary

- Decimal integer → binary: Keep dividing by 2 until the quotient is 0. Collect the remainders in reverse order.

- Example: 162:

\[
\begin{array}{lll}
162 / 2 &= 81 & \text{rem} 0 \\
/ 2 &= 40 & \text{rem} 1 \\
/ 2 &= 20 & \text{rem} 0 \\
/ 2 &= 10 & \text{rem} 0 \\
/ 2 &= 5 & \text{rem} 0 \\
/ 2 &= 2 & \text{rem} 1 \\
/ 2 &= 1 & \text{rem} 0 \\
/ 2 &= 0 & \text{rem} 1 \\
\end{array}
\]

\[
\begin{array}{c}
10100010 \\
\text{128 + 32 + 2 = 162}
\end{array}
\]
Converting decimal to binary

- Decimal integer ➔ binary: Keep dividing by 2 until the quotient is 0. Collect the remainders in reverse order.
- Example: 162.375:

  162 / 2 = 81  rem 0
  81 / 2 = 40  rem 1
  40 / 2 = 20  rem 0
  20 / 2 = 10  rem 0
  10 / 2 = 5   rem 0
  5 / 2 = 2   rem 1
  2 / 2 = 1   rem 0
  1 / 2 = 0   rem 1

- So, $162.375_{10} = 10100010.011_2$
Converting Decimal to Binary

- How do you represent 49 in 8-bit unsigned binary?
  a) 110001
  b) 10011
  c) 00110001
  d) 10001100

Powers of 2:

<table>
<thead>
<tr>
<th>$2^0$</th>
<th>$2^1$</th>
<th>$2^2$</th>
<th>$2^3$</th>
<th>$2^4$</th>
<th>$2^5$</th>
<th>$2^6$</th>
<th>$2^7$</th>
<th>$2^8$</th>
<th>$2^9$</th>
<th>$2^{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
</tbody>
</table>
Why does this work?

- This works for converting from decimal to any base
- Why? Think about converting 162.375 from decimal to decimal.
  
  $\frac{162}{10} = 16\text{ rem }2$
  $\frac{16}{10} = 1\text{ rem }6$
  $\frac{1}{10} = 0\text{ rem }1$

- Each division strips off the rightmost digit (the remainder). The quotient represents the remaining digits in the number.
- Similarly, to convert fractions, each multiplication strips off the leftmost digit (the integer part). The fraction represents the remaining digits.

  $0.375 \times 10 = 3.750$
  $0.750 \times 10 = 7.500$
  $0.500 \times 10 = 5.000$
Writing binary numbers is tedious and error prone

\[ \text{char} - 8 \text{ bits} \quad \text{int} - 32 \text{ bits} \]

Consider

\[ 1001101011100110101100011111101 \]

\[ \underline{32} \]
Use Hexadecimal (base-16) as a shorthand for binary numbers

- The hexadecimal system uses 16 digits:
  
  0 1 2 3 4 5 6 7 8 9 A B C D E F

- We can write our 32-bit number:
  
  \[ \begin{array}{cccccccccc}
  1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
  \end{array} \]

  as

  \[ \begin{array}{cccccccccc}
  0 & x & 9 & A & E & 6 & B & 1 & F & D \\
  \end{array} \]  \text{(C/Java style)}

  \[ \begin{array}{cccccccccc}
  32' & h & 9 & A & E & 6 & B & 1 & F & D \\
  \end{array} \]  \text{(Verilog style)}

Fun fact: Hex is frequently used to specify things like 32-bit IP addresses and 24-bit colors.
Hexadecimal to Binary

- What is B4₁₆ in binary?
  - A: 10110100
  - B: 1010100
  - C: 1011100
  - D: 11000100
Binary and hexadecimal conversions

- Converting from hexadecimal to binary is easy: just replace each hex digit with its equivalent 4-bit binary sequence.

  \[
  261.35_{16} = 2 \quad 6 \quad 1 \quad . \quad 3 \quad 5_{16}
  = 0010 \quad 0110 \quad 0001 \quad . \quad 0011 \quad 0101_2
  \]

- To convert from binary to hex, make groups of 4 bits, starting from the binary point. Add 0s to the ends of the number if needed. Then, just convert each bit group to its corresponding hex digit.

  \[
  10110100.001011_2 = 1011 \quad 0100 \quad . \quad 0010 \quad 1100_2
  = B \quad 4 \quad . \quad 2 \quad C_{16}
  \]
Add binary numbers just like how you do with decimal numbers

- But remember that it’s binary! For example, $1 + 1 = 10$ and you have to carry!

\[
\begin{array}{c}
1 \\
+ 5  \\
\hline
10
\end{array}
\]

\[
\begin{array}{ccccccc}
0 & 1 & 0 & 1 & 1 & \quad \text{Augend (11)} \\
+ & 0 & 1 & 1 & 1 & 0 & \quad \text{Addend (14)} \\
\hline
1 & 1 & 0 & 0 & 0 & \quad \text{Sum}
\end{array}
\]

$16 + 8 + 1 = 25 = \text{YAY}$
Add binary numbers just like how you do with decimal numbers

- But remember that it’s binary! For example, $1 + 1 = 10$ and you have to carry!

The initial carry in is implicitly 0

```
  1 1 1 0  \quad \text{Carry in}
+ 0 1 0 1 1  \quad \text{Augend}
+ 0 1 1 1 0  \quad \text{Addend}
\hline
  1 1 0 0 1  \quad \text{Sum}
```

most significant bit, or MSb

least significant bit, or LSb
Computers restrict all binary numbers to use the same number of bits (i.e., fixed-width)

- What if we do that same addition, using only 4-bit numbers
- (and where the result can only be 4 bits long...)

```
  1011  Augend  (11)
+ 1110  Addend  (14)
  1001  Sum
```

- $8 + 1 = 9 \neq 11$ (overflow)
- Interpretation: 8 + 1 = 9
The number wheel (4-bit unsigned #’s)
“Carry-out” is a procedure, “Overflow” is an interpretation

**Carry-out**
- Occurs at every bit-position
- The process of moving larger numbers to higher bit positions
- Focuses on bit-wise operations

**Overflow**
- Can only be seen after completing an entire mathematical operation
- When the interpretation of a set of bits does not match the expected value after a mathematical operation
- Focuses on representational range (i.e., 4 bits represent 0-15)
Bitwise Logical operations support logical operations on multi-bit \textbf{words}

- To apply a logical operation to two words $X$ and $Y$, apply the operation on each pair of bits $X_i$ and $Y_i$:

\[
\begin{array}{c}
1011 \\
\text{AND} \\
1110 \\
1010 \\
\end{array}
\quad
\begin{array}{c}
1011 \\
\text{OR} \\
1110 \\
1111 \\
\end{array}
\quad
\begin{array}{c}
1011 \\
\text{XOR} \\
1110 \\
0101 \\
\end{array}
\]
Languages like C, C++ and Java provide bitwise logical operations: \( \times \& \gamma \)

\& (AND) | (OR) \( ^\wedge \) (XOR) \( \sim \) (NOT)

- These operations treat each integer as a bunch of individual bits:
  
  \[
  13 \& 25 = 9 \quad \text{because} \quad 01101 \\
  \quad \text{&} \quad 11001 \\
  \quad \quad 01001
  \]

- Bitwise operators are often used in programs to set a bunch of Boolean options, or flags, with one argument.

- They are *not* the same as the operators \&\&, || and !, which treat each integer as a single logical value (0 is false, everything else is true):
  
  \[
  13 \&\& 25 = 1 \quad \text{because} \quad \text{true} \&\& \text{true} = \text{true}
  \]

"\( x=13 \)" "\( y=25 \)"
Bit-wise XOR

001011 XOR 110011

- A: 111001
- B: 111011
- C: 111000
- D: 000110
Bitwise operations are used to find network information

- IP addresses are actually 32-bit (or 128-bit) binary numbers
- For example, you can bitwise-AND an address 192.168.10.43 with a “subnet mask” to find the “network address,” or which network the machine is connected to.

  \[
  192.168.10.43 = 11000000.10101000.00001010.00101011 \\
  \& 255.255.255.224 = 11111111.11111111.11111111.11100000 \\
  \underline{192.168.10.32} = 11000000.10101000.00001010.00100000
  \]

- You can use bitwise-OR to generate a “broadcast address,” for sending data to all machines on the local network.

  \[
  192.168.10.43 = 11000000.10101000.00001010.00101011 \\
  \mid 0. 0. 0. 31 = 00000000.00000000.00000000.00011111 \\
  \underline{192.168.10.63} = 11000000.10101000.00001010.00111111
  \]
Preview for next time: How do we represent negative numbers

- It is useful to be able to represent negative numbers.

- What would be ideal is:
  - If we could use the **same algorithm to add signed numbers as we use for unsigned numbers**
  - Then our computers wouldn’t need 2 kinds of adders, just 1.

- This is achieved using the **2’s complement representation**.
The number wheel (4-bit unsigned #'s)
The number wheel (4-bit 2’s complement)