Lecture 2: Representing Integers

CS 105
Abstraction
Memory: A (very large) array of bytes

- **Memory** is an array of **bits**.
- A **byte** is a unit of eight bits.
- An index into the array is an **address**, **location**, or **pointer**.
  - Often expressed in hexadecimal.
- We speak of the **value** in memory at an address.
  - The value may be a single byte …
  - … or a multi-byte quantity starting at that address.
Representing Integers

• Arabic Numerals: 47

• Roman Numerals: XLVII

• Brahmi Numerals:

• Tally Marks: IIII IIII IIII IIII IIII IIII IIII IIII IIII II
Base-10 Integers

1000 (10^3)

100 (10^2)

10 (10^1)

1 (10^0)

0 0 0 5

0 0 4 7

1 8 8 7
Storing bits

- Static random access memory (SRAM): stores each bit of data in a flip-flop, a circuit with two stable states.
- Dynamic Memory (DRAM): stores each bit of data in a capacitor, which stores energy in an electric field (or not).
- Magnetic Disk: regions of the platter are magnetized with either N-S polarity or S-N polarity.
- Optical Disk: stores bits as tiny indentations (pits) or not (lands) that reflect light differently.
- Flash Disk: electrons are stored in one of two gates separated by oxide layers.
Base-2 Integers (aka Binary Numbers)

128 \( (2^7) \)  64 \( (2^6) \)  32 \( (2^5) \)  16 \( (2^4) \)  8 \( (2^3) \)  4 \( (2^2) \)  2 \( (2^1) \)  1 \( (2^0) \)

0  0  0  0  0  0  1  0  0  1
0  0  1  0  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1
Binary Numbers

• Decimal (Base-10):

4211

= 4 \cdot 10^3 + 2 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0

= 4211

• Binary (Base-2):

1011

= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0

= 11
Exercise 1: Binary Numbers

- Consider the following four-bit binary values. What is the (base-10) integer interpretation of these values?

1. 0001  = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1
2. 1010  = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 2 = 10
3. 0111  = 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 4 + 2 + 1 = 7
4. 1111  = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15
Exercise 2: Binary Number Range

• What are the max number and min number that can be represented by a w-bit binary number?

1. \( w = 3 \)  
   \[
   \text{min} = 000_2 = 0_{10} \quad \text{max} = 111_2 = 2^2 + 2^1 + 2^0 = 7_{10}
   \]

2. \( w = 4 \)  
   \[
   \text{min} = 0000_2 = 0_{10} \quad \text{max} = 1111_2 = 2^3 + 2^2 + 2^1 + 2^0 = 15_{10}
   \]

3. \( w = 8 \)  
   \[
   \text{min} = 00000000_2 = 0_{10} \quad \text{max} = 11111111_2 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 255_{10}
   \]

• What is the general equation?
  
• \( 2^w - 1 \)
## Unsigned Integers in C

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Size (bytes)</th>
<th>For x86_64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>unsigned long</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Unsigned -> Cannot represent negative numbers.
# ASCII characters

<table>
<thead>
<tr>
<th>Char</th>
<th>Dec</th>
<th>Binary</th>
<th>Char</th>
<th>Dec</th>
<th>Binary</th>
<th>Char</th>
<th>Dec</th>
<th>Binary</th>
<th>Char</th>
<th>Dec</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>33</td>
<td>00100001</td>
<td>1</td>
<td>49</td>
<td>00110001</td>
<td>A</td>
<td>65</td>
<td>01000001</td>
<td>Q</td>
<td>81</td>
<td>01010001</td>
</tr>
<tr>
<td>&quot;</td>
<td>34</td>
<td>00100010</td>
<td>2</td>
<td>50</td>
<td>00110010</td>
<td>B</td>
<td>66</td>
<td>01000010</td>
<td>R</td>
<td>82</td>
<td>01010010</td>
</tr>
<tr>
<td>#</td>
<td>35</td>
<td>00100011</td>
<td>3</td>
<td>51</td>
<td>00110011</td>
<td>C</td>
<td>67</td>
<td>01000011</td>
<td>S</td>
<td>83</td>
<td>01010011</td>
</tr>
<tr>
<td>$</td>
<td>36</td>
<td>00100100</td>
<td>4</td>
<td>52</td>
<td>00110100</td>
<td>D</td>
<td>68</td>
<td>01000100</td>
<td>T</td>
<td>84</td>
<td>01010100</td>
</tr>
<tr>
<td>%</td>
<td>37</td>
<td>00100101</td>
<td>5</td>
<td>53</td>
<td>00110101</td>
<td>E</td>
<td>69</td>
<td>01000101</td>
<td>U</td>
<td>85</td>
<td>01010101</td>
</tr>
<tr>
<td>&amp;</td>
<td>38</td>
<td>00100110</td>
<td>6</td>
<td>54</td>
<td>00110110</td>
<td>F</td>
<td>70</td>
<td>01000110</td>
<td>V</td>
<td>86</td>
<td>01010110</td>
</tr>
<tr>
<td>'</td>
<td>39</td>
<td>00100111</td>
<td>7</td>
<td>55</td>
<td>00110111</td>
<td>G</td>
<td>71</td>
<td>01000111</td>
<td>W</td>
<td>87</td>
<td>01010111</td>
</tr>
<tr>
<td>(</td>
<td>40</td>
<td>00101000</td>
<td>8</td>
<td>56</td>
<td>00111000</td>
<td>H</td>
<td>72</td>
<td>01001000</td>
<td>X</td>
<td>88</td>
<td>01011000</td>
</tr>
<tr>
<td>)</td>
<td>41</td>
<td>00101001</td>
<td>9</td>
<td>57</td>
<td>00111001</td>
<td>I</td>
<td>73</td>
<td>01001001</td>
<td>Y</td>
<td>89</td>
<td>01011001</td>
</tr>
<tr>
<td>*</td>
<td>42</td>
<td>00101010</td>
<td>:</td>
<td>58</td>
<td>00111010</td>
<td>J</td>
<td>74</td>
<td>01001010</td>
<td>Z</td>
<td>90</td>
<td>01011010</td>
</tr>
<tr>
<td>+</td>
<td>43</td>
<td>00101011</td>
<td>;</td>
<td>59</td>
<td>00111100</td>
<td>K</td>
<td>75</td>
<td>01001011</td>
<td>[</td>
<td>91</td>
<td>01011011</td>
</tr>
<tr>
<td>,</td>
<td>44</td>
<td>00101100</td>
<td>&lt;</td>
<td>60</td>
<td>00111100</td>
<td>L</td>
<td>76</td>
<td>01001100</td>
<td>\</td>
<td>92</td>
<td>01011100</td>
</tr>
<tr>
<td>-</td>
<td>45</td>
<td>00101101</td>
<td>=</td>
<td>61</td>
<td>00111101</td>
<td>M</td>
<td>77</td>
<td>01001101</td>
<td>]</td>
<td>93</td>
<td>01011101</td>
</tr>
<tr>
<td>.</td>
<td>46</td>
<td>00101110</td>
<td>&gt;</td>
<td>62</td>
<td>00111110</td>
<td>N</td>
<td>78</td>
<td>01001110</td>
<td>^</td>
<td>94</td>
<td>01011110</td>
</tr>
<tr>
<td>/</td>
<td>47</td>
<td>00101111</td>
<td>?</td>
<td>63</td>
<td>00111111</td>
<td>O</td>
<td>79</td>
<td>01001111</td>
<td>_</td>
<td>95</td>
<td>01011111</td>
</tr>
<tr>
<td>0</td>
<td>48</td>
<td>00110000</td>
<td>@</td>
<td>64</td>
<td>01000000</td>
<td>P</td>
<td>80</td>
<td>01010000</td>
<td>\</td>
<td>96</td>
<td>01100000</td>
</tr>
</tbody>
</table>

https://www.ascii-code.com/
Hexadecimal Numbers (Base 16)

0x2c3530e1

How many binary digits can you represent with a single hexadecimal (base 16) digit?
Exercise 3: Hexadecimal Numbers

- Consider the following hexadecimal values. What is the representation of each value in (1) binary and (2) decimal?

1. $0xA = 1010_2 = 10_{10}$
2. $0x11 = 00010001_2 = 17_{10}$
3. $0x2F = 00101111_2 = 47_{10}$
Endianness

• **Big Endian:** low-order bits go on the right (47)
  - I tend to think in big endian numbers, so examples in class will generally use this representation
  - Networks generally use big endian (aka network byte order)

• **Little Endian:** low-order bits go on the left (74)
  - Most modern machines use this representation

• I will try to always be clear about whether I'm using a big endian or little endian representation

• When in doubt, ask!
Arithmetic Logic Unit (ALU)

- A circuit that performs bitwise operations and arithmetic on integer binary types

Status examples:
- Carry-out
- Zero
- Negative
- Overflow
- parity

Opcode examples:
- Add, Subtract
- Increment, Decrement
- AND, OR, XOR
- Shift, Rotate
Bitwise vs Logical Operations in C

- **Bitwise Operators** &amp;, |, ~, ^
  - View arguments as bit vectors
  - operations applied bit-wise in parallel

- **Logical Operators** &&, ||, !
  - View 0 as “False”
  - View anything nonzero as “True”
  - Always return 0 or 1
  - Short-circuit termination

- **Shift operators** &lt;&lt;, &gt;&gt;
  - Left shift fills with zeros
  - For unsigned integers, right shift is logical (fills with zeros)
Exercise 4: Bitwise vs Logical Operations

Assume unsigned char data type (one byte). What do each of the following expressions evaluate to (interpreted as unsigned integers and expressed base-10)?

1. \(~226\) = \(~11100010\) = 00011101 = 29
2. \(!226\) = \(!11100010\) = 00000000 = 0

3. \(120 \& 85\) = 01111000 \& 01010101 = 01010000 = 80
4. \(120 \mid 85\) = 01111000 \mid 01010101 = 01111101 = 125
5. \(120 \&\& 85\) = 01111000 \&\& 01010101 = 00000001 = 1
6. \(120 \mid\mid 85\) = 01111000 \mid\mid 01010101 = 00000001 = 1

7. \(81 \ll 4\) = 01010001 \ll 4 = 00010000 = 16
8. \(81 \ll 2\) = 01010001 \ll 2 = 01000100 = 68
9. \(81 \gg 4\) = 01010001 \gg 4 = 00000101 = 5
10. \(81 \gg 2\) = 01010001 \gg 2 = 00010100 = 20
Example: Using Bitwise Operations

What do these operations do?

\( x \ll 2 \)

- "multiply by 4"

\( x \& 1 \)

- “\( x \) is odd”

\((x + 7) \& 0xFFFFFFFF8\)

- “round up to a multiple of 8”
Addition Example

- Compute $5 + 6$ assuming all ints are stored as **eight-bit** (1 byte) unsigned values.

\[
\begin{array}{cccccccc}
& & & & & 1 & & \\
& & & & 0 & 0 & 0 & 0 \\
& & + & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
& & & & & 1 & & \\
& & & & 0 & 0 & 0 & 0 \\
& & + & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\]

Like you learned in grade school, only binary!

… and with a finite number of digits
Addition Example with Overflow

- Compute $200 + 100$ assuming all ints are stored as eight-bit (1 byte) unsigned values.

\[
\begin{array}{c}
1 1 \\
1 1 0 0 1 0 0 0 \\
+ 0 1 1 0 0 1 0 0 \\
0 0 1 0 1 1 0 0 \\
\end{array}
\] = 44 (Base-10)

Like you learned in grade school, only binary!

... and with a finite number of digits
Error Cases

- Assume $w$-bit unsigned values

\[
x + y = \begin{cases} 
  x + y & \text{(normal)} \\
  x + y - 2^w & \text{(overflow)} 
\end{cases}
\]

- Overflow occurred if and only if $x + \frac{u}{w} y < x$
Exercise 5: Binary Addition

- Given the following 5-bit unsigned values, compute their sum and indicate whether an overflow occurred

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x+y</th>
<th>overflow?</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010</td>
<td>00101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td>00100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>10001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise 5: Binary Addition

• Given the following 5-bit unsigned values, compute their sum and indicate whether an overflow occurred.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x+y</th>
<th>overflow?</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010</td>
<td>00101</td>
<td>00111</td>
<td>no</td>
</tr>
<tr>
<td>01100</td>
<td>00100</td>
<td>10000</td>
<td>no</td>
</tr>
<tr>
<td>10100</td>
<td>10001</td>
<td>00101</td>
<td>yes</td>
</tr>
</tbody>
</table>
Multiplication Example

- Compute 5 \times 6 assuming all ints are stored as eight-bit (1 byte) unsigned values

\[
\begin{array}{c}
0 0 0 0 0 1 0 1 \\
\times 0 0 0 0 0 0 1 1 0 \\
\hline
0 0 0 0 0 0 0 0 0 0 \\
0 0 0 0 0 1 0 1 0 \\
+ 0 0 0 0 0 1 0 1 0 0 \\
\hline
0 0 0 1 1 1 1 1 0
\end{array}
\]

= 30 (Base-10)

Like you learned in grade school, only binary!

\ldots \text{and with a finite number of digits}
Multiplication Example

• Compute $200 \times 3$ assuming all ints are stored as **eight-bit** (1 byte) unsigned values

\[
\begin{array}{c}
11001000 \\
\times 000000011 \\
\hline
11001000 \\
+ 1100100000 \\
\hline
1001011000 = 88 \text{ (Base-10)}
\end{array}
\]

Like you learned in grade school, only binary!

… and with a finite number of digits
Error Cases

- Assume $w$-bit unsigned values

\[
x \cdot y = (x \cdot y) \mod 2^w
\]
Exercise 6: Binary Multiplication

- Given the following 3-bit unsigned values, compute their product and indicate whether an overflow occurred.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x*y</th>
<th>overflow?</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise 6: Binary Multiplication

Given the following 3-bit unsigned values, compute their product and indicate whether an overflow occurred.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x*y</th>
<th>overflow?</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>101</td>
<td>100</td>
<td>yes</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td>110</td>
<td>no</td>
</tr>
<tr>
<td>111</td>
<td>010</td>
<td>110</td>
<td>yes</td>
</tr>
</tbody>
</table>
Multiplying with Shifts

- Multiplication is slow
- Bit shifting is kind of like multiplication, and is often faster

- What is “x << 3”?
  - x * 8 = x << 3

- How could you perform “x * 10” with shifts and addition?
  - x * 10 = x << 3 + x << 1

- Most compilers will automatically replace multiplications with shifts where possible