ARITHMETIC OPERATIONS

WE HAVE THE DATA, WHAT NOW?

Operations in C

- Bit-wise boolean operations
- Logical operation
- Arithmetic operations

BOOLEAN ALGEBRA

Algebraic representation of logic

- Encode “True” as 1 and “False” as 0
- Operators **&** **|** **~** **^**

AND (&)

TRUE when both A = 1 AND B = 1

&	0	1
0	0	0
1	0	1

OR (|)

TRUE when either A = 1 OR B = 1

|	0	1
0	0	1
1	1	1

BOOLEAN ALGEBRA

Algebraic representation of logic

- Encode "True" as 1 and "False" as 0
- Operators **&** **|** **~** **^**

NOT (~)
Inverts the truth value

~	
0	1
1	0

XOR "Exclusive OR" (^)
TRUE when A or B = "1", but not both.

^	0	1
0	0	1
1	1	0

BOOLEAN ALGEBRA

Applies to any "Integer" data type

- That is… `long, int, short, char`
- View arguments as bit vectors
- Operations applied in a "bit-wise" manner

Examples:

```
   01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
   01000001      01111101      00111100      10101010
```

PARTNER ACTIVITY

```
0x69  &  0x55

  01101001
& 01010101
  01000001
    = 0x41
```

```
0x69  ^  0x55

  01101001
^ 01010101
  00111100
    = 0x3C
```

```
0x69  |  0x55

  01101001
| 01010101
  01111101
    = 0x7D
```

```
~ 0x55

~ 01010101
  10101010
    = 0xAA
```

SHIFT OPERATIONS

Left Shift: `x << y`

- Shift bit-vector x left y positions
  - Throw away extra bits on left
  - Fill with 0's on right

Argument x	01100010
x << 3	00010000

Right Shift: `x >> y`

- Shift bit-vector x right y positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0's on left
- Arithmetic shift
  - Replicate most significant bit on left
  - Recall two's complement integer representation
  - Perform division by 2 via shift

Argument x	10100010
Logical >> 2	00101000
Arith. >> 2	11101000

PARTNER ACTIVITY

x	x << 3	x >> 2 (Logical)	x >> 2 (Arithmetic)
0xF0	0x80	0x3C	0xFC
0x0F	0x78	0x03	0x03
0xCC	0x60	0x33	0xF3
0x55	0xA8	0x15	0x15

LOGIC OPERATIONS IN C

Operations always return 0 or 1

Comparison operators

- `>` `>=` `<` `<=` `==` `!=`

Logical Operators

- `&&` `||` `!`
- Logical AND, Logical OR, Logical negation
- In C (and most languages)
  - 0 is “False”
  - Anything non-zero is “True”

LOGIC OPERATIONS IN C

Examples (char data type)

- `!0x41  ==  0x00`
- `!0x00  ==  0x01`
- `!!0x41 ==  0x01`

What are the values of

- `0x69 || 0x55`
- `0x69 |  0x55`

What does this expression do?

- `(p && *p)`

Watch out!

- Logical operators versus bitwise

```
&& versus &
|| versus |
== versus =
```

USING
BITWISE AND LOGICAL OPERATIONS

Two integers x and y

For any processor, independent of the size of an integer, write C expressions without any "=" signs that are true if:

- x and y have any non-zero bits in common in their low order byte
  - `0xFF & (x & y)`
- x has any 1 bits at higher positions than the low order 8 bits
  - `~0xFF & x`
  - `(x & 0xFF) ^ x`
  - `(x >> 8)`
- x is zero
  - `!x`
- x == y
  - `!(x ^ y)`

ARITHMETIC OPERATIONS

Signed / Unsigned

- Addition and subtraction
- Multiplication
- Division

UNSIGNED ADDITION WALKTHROUGH

Binary (and hexadecimal) addition similar to decimal

Assuming arbitrary number of bits, use binary addition to calculate 7 + 7

```
  0111
+ 0111
------
```

Assuming arbitrary number of bits, use hexadecimal addition to calculate 168+123 (A8+7B)

```
  A8
+ 7B
----
```

UNSIGNED SUBTRACTION WALKTHROUGH

Binary subtraction similar to decimal
Assuming 4 bits, use subtraction to calculate 6 - 3

```
  0110
- 0011
------
```

In hardware, done via 2s complement negation followed by addition,
(2s complement negation of 3 = ~3 + 1) `0011 => 1100 => 1101   (-3)`

```
  0110
+ 1101
------
```

UNSIGNED SUBTRACTION WALKTHROUGH

Hexadecimal subtraction similar to decimal
Use subtraction to calculate 266-59 (0x10A – 0x3B)

```
   10A
-  03B
-----
```

UNSIGNED ADDITION AND OVERFLOW

Suppose we have a computer with 4-bit words

What is 9 + 9?

- 1001 + 1001 **= 0010 (2 or 18 % 2^4)**

With w bits, unsigned addition is regular addition, modulo 2^w

- Bits beyond **w** are discarded

PARTNER ACTIVITY

Assuming an arbitrary number of bits, calculate

```
  0x693A
+ 0xA359
--------
```

What would the result be if a 16-bit representation was used instead?

UNSIGNED ADDITION

With 32-bits, unsigned addition is modulo 2^{32}

What is the value of

```
  0xc0000000
+ 0x70004444
------------
```

```
#include <stdio.h>
unsigned int sum(unsigned int a, unsigned int b)
{
   return a+b;
}
int main () {
   unsigned int i=0xc0000000;
   unsigned int j=0x70004444;
   printf("%x\n",sum(i,j));
   return 0;
}
```

Output: 30004444

PARTNER ACTIVITY

Assuming 5 bit 2s complement representation, what is the decimal value of the following sums: (7 + 11), (-14 + 5), and (-11 + -2)

```
Recall:   -16   8   4   2   1

  00111              10010                  10101
+ 01101            + 00101                + 11110
-------            -------                -------
```

What would the result be if a 16-bit representation was used instead?

POINTER ARITHMETIC

Always unsigned

Based on size of the type being pointed to

- Incrementing a `(int *)` adds 4 to pointer
- Incrementing a `(char *)` adds 1 to pointer

PARTNER ACTIVITY

Consider the following declaration on

```
char*   cp = 0x100;
int*    ip = 0x200;
float*  fp = 0x300;
double* dp = 0x400;
int     i  = 0x500;
```

What are the hexadecimal values of each after execution of these commands?

`cp++;`	0x101
`ip++;`	0x204
`fp++;`	0x304
`dp++;`	0x408
`i++;`	0x501

DATA SIZES IN C

C Data Type	Typical 32-bit	x86-64
char	1	1
short	2	2
int	4	4
long	4	8
float	4	4
double	8	8
pointer	4	8

TWO'S COMPLEMENT MULTIPLICATION

Same problem as unsigned

The bit-level representation for two's-complement and unsigned is identical

- This simplifies the integer multiplier

As before, the interpretation of this value is based on signed vs. unsigned

Maintaining exact results

- Need to keep expanding word size with each product computed
- Must be done in software, if needed
  - e.g., by "arbitrary precision" arithmetic packages

MULTIPLICATION BY SHIFTING

What happens if you shift a decimal number left one place?

- 30_{10} => 300_{10}
  - Multiplies number by base (10)

What happens if you shift a binary number left one place?

- 00011_2 => 00110_2
  - Multiplies number by base (2)

MULTIPLICATION BY SHIFTING

What if you shift a decimal number left N positions?

- (N = 3) 31_{10} => 31000_{10}
- Multiplies number by $(\text{base})^N$ or 10^N (1000 for N = 3)

What if you shift a binary number left N positions?

- 00001000_2 << 2 = 00100000_2
- (8_{10}) << 2 = (32_{10})

- Multiplies number by $(\text{base})^N$ or 2^N

MULTIPLICATION BY SHIFTS AND ADDS

CPUs shift and add faster than multiply

```
u << 3   ==   u * 8
```

- Compiler may automatically generate code to implement multiplication via shifts and adds
  - Dependent upon multiplication factor

- Examples
  - K = 24

    ```
    (u << 5) - (u << 3) ==  u * 32 - u * 8 == u * 24
    ```
  - K = 18

    ```
    (u << 4) + (u << 1) ==  u * 16 + u * 2 == u * 18
    ```

DIVISION BY SHIFTING

What happens if you shift a decimal number right one digit?

31_{10} => 3_{10}

Divides number by base (10), rounds down towards 0

What happens if you shift an unsigned binary number right one bit?

00000111_2 => 00000011_2 (7 >> 1 = 3)

Divides number by base (2), rounds down towards 0

DIVISION BY SHIFTING

Question:

If:

```
7 >> 1 == 3
```

What would you expect the following to give you?

```
-7 >> 1 == ?
```

Try using a byte

```
 7 == 00000111
-7 == 11111001 (flip bits, add 1)
-7 >> 1 == 11111100 (-4)!
```

What happens if you shift a negative signed binary number right one bit?

- Divides number by base (2), rounds away from 0!

WHY ROUNDING MATTERS

German parliament (1992)

- 5% law before vote allowed to count for a party
- Rounding of 4.97% to 5% allows Green party vote to count
- “Rounding error changes Parliament makeup” Debora Weber-Wulff, The Risks Digest, Volume 13, Issue 37, 1992

Vancouver stock exchange (1982)

- In January 1982 the index was initialized at 1000 and iteratively updated with each subsequent trade.
- After each update, the index was truncated to three decimal places. The truncated value was used to calculate the next value of the index. Updates occurred approximately 3000 times each day.
- The accumulated truncations led to an erroneous loss of around 25 points per month. Over the weekend of November 25–28, 1983, the error was corrected, raising the value of the index from its Friday closing figure of 524.811 to 1098.892.

OPERATOR PRECEDENCE

What is the output of this code?

```
#include <stdio.h>
int main () {
   int i = 3;
   printf("%d\n", i*8 - i*2);
   printf("%d\n", i<<3 - i<<1);
}
```

```
$./a.out
18
6
```

C OPERATOR PRECEDENCE

<table>
<tr><th>Precedence</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td rowspan="6">1</td><td>++ --</td><td>Suffix/postfix increment and decrement</td><td rowspan="6">Left-to-right</td></tr>
<tr><td>()</td><td>Function call</td></tr>
<tr><td>[]</td><td>Array subscripting</td></tr>
<tr><td>.</td><td>Structure and union member access</td></tr>
<tr><td>-></td><td>Structure and union member access through pointer</td></tr>
<tr><td>(type){list}</td><td>Compound literal(C99)</td></tr>
<tr><td rowspan="8">2</td><td>++ --</td><td>Prefix increment and decrement</td><td rowspan="8">Right-to-left</td></tr>
<tr><td>+ -</td><td>Unary plus and minus</td></tr>
<tr><td>! ~</td><td>Logical NOT and bitwise NOT</td></tr>
<tr><td>(type)</td><td>Type cast</td></tr>
<tr><td>*</td><td>Indirection (dereference)</td></tr>
<tr><td>&</td><td>Address-of</td></tr>
<tr><td>sizeof</td><td>Size-of</td></tr>
<tr><td>_Alignof</td><td>Alignment requirement(C11)</td></tr>
<tr><td>3</td><td>* / %</td><td>Multiplication, division, and remainder</td><td rowspan="11">Left-to-right</td></tr>
<tr><td>4</td><td>+ -</td><td>Addition and subtraction</td></tr>
<tr><td>5</td><td><< >></td><td>Bitwise left shift and right shift</td></tr>
<tr><td rowspan="2">6</td><td>< <=</td><td>For relational operators < and ≤ respectively</td></tr>
<tr><td>> >=</td><td>For relational operators > and ≥ respectively</td></tr>
<tr><td>7</td><td>== !=</td><td>For relational = and ≠ respectively</td></tr>
<tr><td>8</td><td>&</td><td>Bitwise AND</td></tr>
<tr><td>9</td><td>^</td><td>Bitwise XOR (exclusive or)</td></tr>
<tr><td>10</td><td>|</td><td>Bitwise OR (inclusive or)</td></tr>
<tr><td>11</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>12</td><td>||</td><td>Logical OR</td></tr>
<tr><td>13</td><td>?:</td><td>Ternary conditional</td><td rowspan="6">Right-to-Left</td></tr>
<tr><td rowspan="5">14</td><td>=</td><td>Simple assignment</td></tr>
<tr><td>+= -=</td><td>Assignment by sum and difference</td></tr>
<tr><td>*= /= %=</td><td>Assignment by product, quotient, and remainder</td></tr>
<tr><td><<= >>=</td><td>Assignment by bitwise left shift and right shift</td></tr>
<tr><td>&= ^= |=</td><td>Assignment by bitwise AND, XOR, and OR</td></tr>
<tr><td>15</td><td>,</td><td>Comma</td><td>Left-to-right</td></tr>
</table>