Consider the following type definition and variable declarations:

```
struct personT {                  struct personT p1;
  char name[32];
  int  age;                       struct personT people[40]
  float heart_rate;
};
```

(1) What type is each of the following expressions?

```
p1                          p1.name             people

p1.heart_rate               people.name         people[0]

people[0].name              people[0].name[3]
```

(2) Show the C staments to set the 3rd person's age to 18, heart_rate to 66, and name to "Ralph"

```
void mystery(int x[], int y);                    // STACK:
int main() {
  int i, arr[5];
  for(i=0; i < 5; i++) {
    arr[i] = i;
  }
  mystery(arr,4);
  for(i=0; i < 5; i++) {
     printf("arr[%d]=%d\n",I,arr[i]);
  }
}
/***************************/
void mystery(int x[], int y) {
  int i;
  for(i=0; i < y; i++) {
     if((x[i]%2) == 0) {
        x[i] = x[i] + 1;
     }
  }
  return;
}
```

Binary Representation and Operations on Binary Data

Week 2, CS31 Fall 2013

Tia Newhall

How a computer runs a program

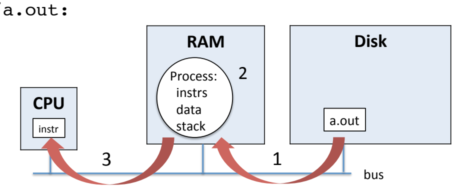
Program
Operating System
Computer Hardware

- Compiler: translates C source to binary executable file (OS/Arch specific)

  foo.c ---> gcc ---> a.out
- a.out: binary executable: all 0's and 1's
  - Instructions (e.g. add the value of x and 6)
  - Some data (e.g. 6, maybe initial value of x)
  - Information so that OS can load and start running

Operating System

`./a.out:`



1. Loads a.out from Disk into RAM
2. Creates a Process: running program w/own address space: (like an array of addressable contents: data, instructions)
3. Initializes the CPU to start running 1st instruction

You can view binary file contents

xxd (or hexdump –C) to veiw binary file values:

```
xxd a.out     # a binary executable file

Address:  value of the next 16 bytes in memory
0000000: 7f45 4c46 0201 0100 0000 0000 0000 0000
0000010: 0200 3e00 0100 0000 3007 4000 0000 0000
0000020: 4000 0000 0000 0000 084d 0000 0000 0000
  …
```

(these weird numbers (f,c,e, …), are hexidecimal digits)

```
xxd myprog.c     # binary ascii encoding of C source:

0000000: 2369 6e63 6c75 6465 3c73 7464 696f 2e68
             #i   nc   lu   de   <s   td   io   .h
0000010: 3e0a 696e 7420 6d61 696e 2829 207b 0a20
             >\n  in   t    ma   in   ()   {  \n
…
```

This Week:

- Binary Representation of different data types:
  6, -4.6, ‘a’
  bit, byte, word
  signed and unsigned
- How operations on binary data work
  6 + 12, 15 – 5, -9 + 12, ..
- Operations on bits
  Logical vs. bit-wise operators

Bits and Bytes

- Bit: a 0 or 1 values
  - HW represents as two different voltages
    - 1: the presence of voltage (high voltage)
    - 0: the absence of voltage (low voltage)
- Byte: 8 bits, the smallest addressable unit
  0: 01010101
  1: 10101010
  2: 00001111
  …
- Word: some number of bytes, depends on architecture (4 bytes is common)

How many values?

- The number of bits determines the range of values
  - 2 values with 1 bit
  - 4 values with 2 bits
  - 8 values with 3 bits
  - 16 values with 4 bits … 2^n values with n bits

```
1 bit:            0                                  1
            /          \                         /        \
2:        00             01                   10             11
       /     \         /    \              /     \         /    \
3:  000      001      010      011        100      101   110     111
   / \      / \     / \      / \        /  \     / \   / \    / \
0000 0001 0010 0011 0100 0101 0110 0111  1000 1001  …
```

C types and their sizes

```
• 1 byte:  char, unsigned char
• 2 bytes: short, unsigned short
• 4 bytes: int, unsigned int, float
• 8 bytes: long long, unsigned long long, double
• 4 or 8 bytes: long, unsigned long
```

```
unsigned long v1;
short s2;
unsigned int u1;
long long ll;
double d1;

printf(“%lu %u %d %lld %g\n”, v1, u1, s2, ll, d1);
printf(“%lu %lu %lu\n”, sizeof(v1), sizeof(s2),
        sizeof(ll));  // prints out number of bytes
```

Unsigned numbers

With N bits, can represent values: 0 to 2^n-1

4 bits: 0000	0		
0001	1	$= 1*2^0$	
0010	2	$= 1*2^1$	
0011	3	$= 1*2^1 + 1*2^0$	$= 2 + 1$
0100	4	$= 1*2^2$	
. . .			
1111	15	$= 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$	
		$= 8 + 4 + 2 + 1$	

Converting binary to decimal:
low order, 0th bit, counts the number of 2^0 (0 or 1)
1st bit is number of 2^1(0 or 1)
2nd bit is number of 2^2’s …

Binary: base 2 numbers

- Decimal, base 10, digits {0,1,2, …, 9}:

  1703: $1*10^3 + 7*10^2 + 0*10^1 + 3*10^0$
  $= 1000 + 700 + 0 + 3 = 1703$

- Binary, base 2, digits {0,1}:

  10101: $1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$
  $= 16 + 0 + 4 + 0 + 1 = 21$

Converting binary to decimal: just follow this pattern

Try

`unsigned char ch = 'm';`

in binary, ch's value is: `01101101`

- Convert to decimal (leave as expression):
- `ch+1` (add in binary, then convert to decimal):

Adding Binary Values

- Add Corresponding digits to get either 0 or 1 with a possible carry bit to next place
- Example adding two 4-bit values (result is 4-bits):

```
   1
   0110     6          1100     12
+  0100   + 4        + 1010    +10
   1010    10        1 0110      6
                     ^carry out bit
```

Unsigned overflow: result requires more bits than have

Representing Signed Integers

`int, short, char, long, long long`

- Use 2's complement encoding
  - High-order bit is **sign bit** (0:positive, 1: negative)
    1xxxxxxx : some negative value
    0xxxxxxx: some positive value
  - Positive 2's compliment encodings are same as their unsigned encodings
    - 0000 is zero signed and unsigned
    - 0110 is six signed and unsigned
  - With N bits, can represent: -2^{N-1} to $2^{N-1}-1$
    4 bit value can represent: -8, -7, …, -1, 0, 1, …, 7

2's Complement

2's complement of N bit value x is: $2^N - x$

4-bit value: 0010 (2)

its 2's complement is: $2^4 - 2$

```
                 10 borrow bit
  10000     011^0        2^4
- 0010    - 0010       - 2
  1110      1110        -2
          (borrow minus 1:10-1=1)
```

(there is a much easier way to negate and to subtract)

2's Complement to Decimal

High order bit is the sign bit, otherwise just like unsigned conversion. 4-bit examples:

```
0110: 0*-2^3 + 1*2^2 + 1*2^1 + 0*2^0
         0   +   4   +   2   +   0 = 6
1110: 1*-2^3 + 1*2^2 + 1*2^1 + 0*2^0
        -8   +   4   +   2   +   0 = -2
```

Try: 1010

1111

2's Complement Negation

Flip the bits and then add 1 (~x + 1):

```
6: 0110: 1001        -3: 1101:  0010
       + 0001                  +0001
         1010                   0011
   = -8+2 = -6             = 2+1 = 3
```

Try: negate 1   negate 7

2's Complement Subtraction

Negate and add: much easier than borrowing

6 – 3 == 6 + ~3 + 1

```
 6     0110     0110
-3   - 0011     1100
              + 0001
              1 0011  = 2 + 1 = 3
              ^  what about carry out bit?
```

It looks like overflow, but the result works out fine if we ignore the carry-out bit (0011 is the correct result)

**we can also do unsigned subtraction in this way

Subtraction

Negate and add 1 to second operand:

Can use the same circuit for add and subtract:

6 - 7 == 6 + ~7 + 1

```
input 1 ------------------------------> ┌───────────┐
input 2 -->[possible bit flipper]-->    │ADD CIRCUIT│---> result
       possible +1 input-------->       └───────────┘
```

Arithmetic Operation Overflow

Overflow: ~running out of enough bits to store result

Signed addition (and subtraction):

```
 2+-1=1     2+-2=0     2+-4=-2     2+7=-7     -2+-7=7
  0010       0010       0010       0010        1110
 +1111      +1110      +1100      +0111       +1001
1 0001     1 0000       1110       1001      1 0111
```

```
 0     1     2        7     -8     -7   ...   -2     -1
0000  0001  0010 ...  0111  1000  1001 ...   1110   1111
      add pos --------->              <------- add neg
<--------- add neg                       add pos ------>

 0     1     2        7     -8     -7   ...          -1
0000  0001  0010 ...  0111  1000  1001 ...          1111
                         ^ overflow when operation crosses here
```

Try Out: Signed Overflow Rules?

4 bit signed values (a-b is a + ~b + 1):

```
                                   carry-in  carry-out
 3 +  7 =    0011 + 0111        =   0    1010 = -6
-3 + -6 =    1101 + 1010        =   1    0111 =  7
-3 +  6 =    1101 + 0110        =   1    0011 =  3
 3 -  6 =    0011 + 1001 + 1    =   0    1101 = -3
 3 + -6 =    0011 + 1010        =   0    1101 = -3
```

Rule for detecting overflow in signed arithmetic?

is the carry-out bit meaningful?

if values are different signs, can we ever get overflow?

Arithmetic Operation Overflow

Overflow: ~running out of enough bits to store result

Unsigned addition (and subtraction):

```
 2+1=3      2-1=1      2+14=0     2-3=15
 0010       0010        0010      0010
+0001      +1111       +1110      1100
 0011     1 0001     1 0000      +0001
                                  1111
```

```
     0     1     2       7     8     9   ...   15
   0000  0001  0010 ...  0111  1000  1001 ...  1111
             add --------->
        <---------- sub
^ subtraction overflow                 addition overflow ^
```

Try Out: Unsigned Overflow Rules?

4 bit unsigned values (a – b is a + ~b +1, a+b is a+b+0):

```
                                  carry-in  carry-out
9 + 11  =    1001 + 1011 + 0 =    1    0100 =  4
9 +  6  =    1001 + 0110 + 0 =    0    1111 = 15
3 +  6  =    0011 + 0110 + 0 =    0    1001 =  9

6 -  3  =    0110 + 1100 + 1  = 1    0011 =  3
3 -  6  =    0011 + 1001 + 1  = 0    1101 = 13
```

Rule for detecting overflow?

is the carry-out bit meaningful? When?

Overflow Rules

- Signed: can only occur when adding two values of the same sign:
  - When sign bits of operands are the same, but the sign bit of result is different
- Unsigned: can occur when adding or when subtracting larger from smaller:
  - When carry-in bit is different than carry-out bit

C_{in}	C_{out}	C_{in} XOR C_{out}
0	0	0
0	1	1
1	0	1
1	1	0

Try out some 4-bit examples:

(1) signed result? (2) unsigned result? (3)overflow?

0110 + 0010   1001 – 1010   1001 + 1110

During Execution what Happens if Overflow?

- HW: sets flags as side-effect of arithmetic computations, these can be tested for error conditions
  - OF: overflow flag: set based on signed overflow
  - CF: set if carry-out is 1, can be used to test for unsigned overflow with carry-in bit
- What does C do?
  - Nothing:

  unsigned char s = 255;

  s = s + 4; // 3, maybe that is what you want?

Sign Extension

- When combining signed values of different num bytes, expanded smaller to equivalent larger size:

```
char y=2, x=-13;
short z = 10;

   z = z + y;                 z = z + x;

0000000000001010           0000000000000101
        00000010                   11110011
0000000000000010           1111111111110011
```

Fill in high-order bits with sign-bit value to get same numeric value in larger number of bytes

Let's verify that this works

4-bit signed value, sign extend to 8-bits, is it the same value?

0111 ---> 0000 0111 obviously still 7

1010 ----> 1111 1010 is this still -6?

-128 + 64 + 32 + 16 + 8 + 0 + 2 + 0 = -6 yes!

Different Number Representations

- Binary: base 2 digits {0,1}
- Decimal: base 10 digits {0, 1, …, 9}
- Hexidecimal: base 16 digits {0, …,9,a,b,c,d,e,f}
  - 2^4 is 16, so 4 binary digits to represent 1 hex: 0101:5, 1100:c

Binary to hex: group into 4 bin digits, convert each group:

```
0011  1010  1100  0101      0011101011000101
  3     a     c     5        = 0x3ac5
```

Hex to binary: expand each hex digit into its 4 binary digits:

```
  a     1     2     f       0xa12f
1010  0001  0010  1111      = 1010000100101111
```

hex is easier to read than binary: 0x3efa vs. 0011111011111010

Decimal to Binary (or to hex)

- 543876 in binary?   -34252 in binary?

if D negative: convert the positive to b, then negate (~b +1)

if D positive: need to find 0 and 1 digits for a_7-a_0 such that:

$a_7*2^7 + a_6*2^6 + a_5*2^5 + a_4*2^4 + a_3*2^3 + a_2*2^2 + a_1*2^1 + a_0*2^0 = D$

idea: build up binary value from low to high-order bit
- if the number D is odd then a_0 1, if even then a_0 is 0
- consider the next bit a_1: its value determined by whether or not D/2 is odd
- ... continue until D/2/2 .../2 is zero

Algorithm: decimal value D, binary result b (b_i is ith digit):

```
i = 0
while (D > 0)
  if D is odd
     set b_i to 1
  if D is even
     set b_i to 0
  i++
  D = D/2
```

```
idea:                    example: D = 105        a0 = 1
        D = b                     D =  52        a1 = 0
      D/2 = b/2                   D =  26        a2 = 0
      D/2 = b/2                   D =  13        a3 = 1
      D/2 = b/2                   D =   6        a4 = 0
      D/2 = b/2                   D =   3        a5 = 1
        0 =  0                    D =   1        a6 = 1
                                  D =   0        a7 = 0

                                      105   =   01101001
```

Try 74   Try -115

Operations on Bits

- Bit-wise operators: bit operands, bit result

& (AND) | (OR)   ~(NOT)   ^(XOR)

A	B	A & B	A | B	~A	A ^ B
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

```
  01010101     01101010     10101010    ~10101111
| 00100001   & 10111011   ^ 01101001     01010000
  01110101     00101010     11000011
```

More Operations on Bits

- Bit-shift operators: << left shift,   >> right shift

```
01010101 << 2  is 01010100
                     2 high-order bits shifted out
                     2 low-order bits filled with 0
01101010 << 4  is 10100000
01010101 >> 2  is 00010101
01101010 >> 4  is 00000110

10101100 >> 2  is 00101011 (logical shift)
            or 11101011 (arithmetic shift)
```

Arithmetic right shift: fills high-order bits w/sign bit

How Used?

Bit vectors: encode yes/no values in individual bits:

(ex) file permissions: `ls –l`

```
   directory  owner  group  world
       d       rwx    rwx    rwx

 -rw-------    …   foo.c
 -rwx------    …   a.out*
 drwx------    …   cs31/
```

Encode in 10 bits: (need a short variable, 2 bytes, to store):

```
 -  r w -  r w -  r w -
 0  1 1 0  1 1 0  1 1 0     permission: 666

chmod 620 foo.c:  0  1 1 0  0 1 0  0 0 0
                       6      2      0
```

Try using bit operators

```
short f = 281; // 0000 0001 0001 1001
```

(1) C code to see if file is readable by group? drwxrwxrwx

```
000000 0 100 011 001   (this value)
```

(2) C code to set perms. so that owner can write?

printf to print diff types and reps:

```
%x: hex
%u: unsigned
%ld: long signed
%llu: unsigned long long

printf("%c %d %x", 'a', 'a', 'a');

printf("%d %x", 1234, 1234);
```

Floating Point Representation

1 bit for sign   sign | exponent | fraction |
8 bits for exponent
23 bits for precision

value = $(-1)^{sign}$ + 1.fraction * $2^{(exponent-127)}$

let's just plug in some values and try it out

```
0xc080015a: 1 10000001   00000000000000101011010
     sign = 1 exp = 129    fraction = 346
```

= -1 + 1.346*2^2 = 5.384

I don't expect you know how to do this

Summary

- Know how binary data represented and manipulated:
  - Different sizes depend on C type:
    - 1 byte, 2 bytes, 4 bytes, 8 bytes
  - Unsigned and Signed Representations
  - Arithmetic operations: + and –
    - Same rules for performing signed & unsigned ops
    - Different rules for determining if result overflowed
  - Bit-wise operations: &, |, ^, ~, <<, >>
  - Different representations: hex, binary, decimal
    - Converting values between these