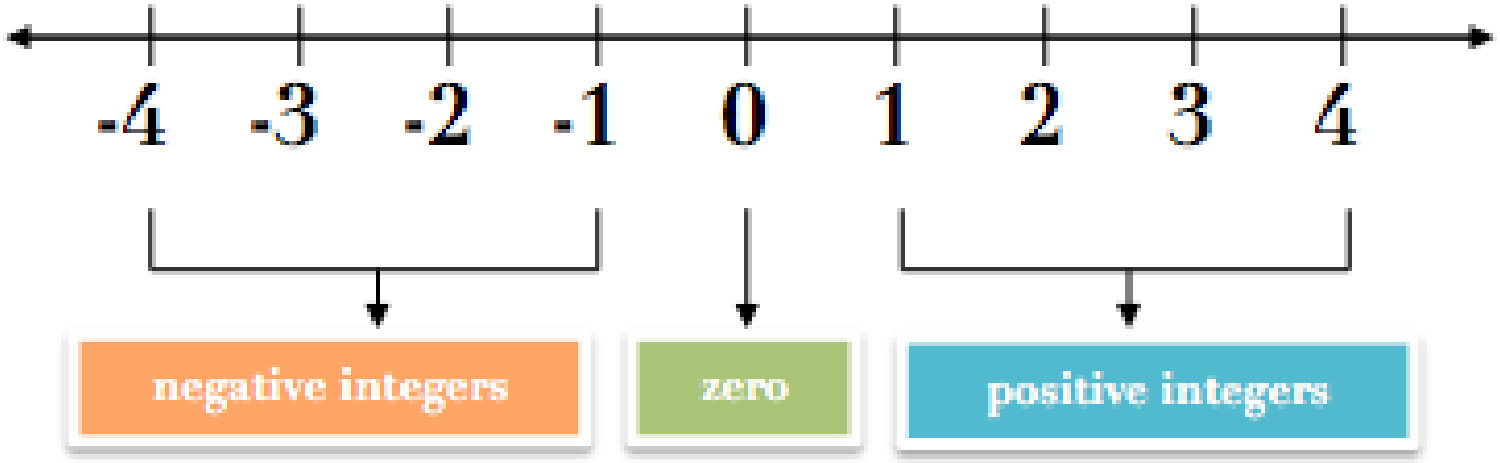
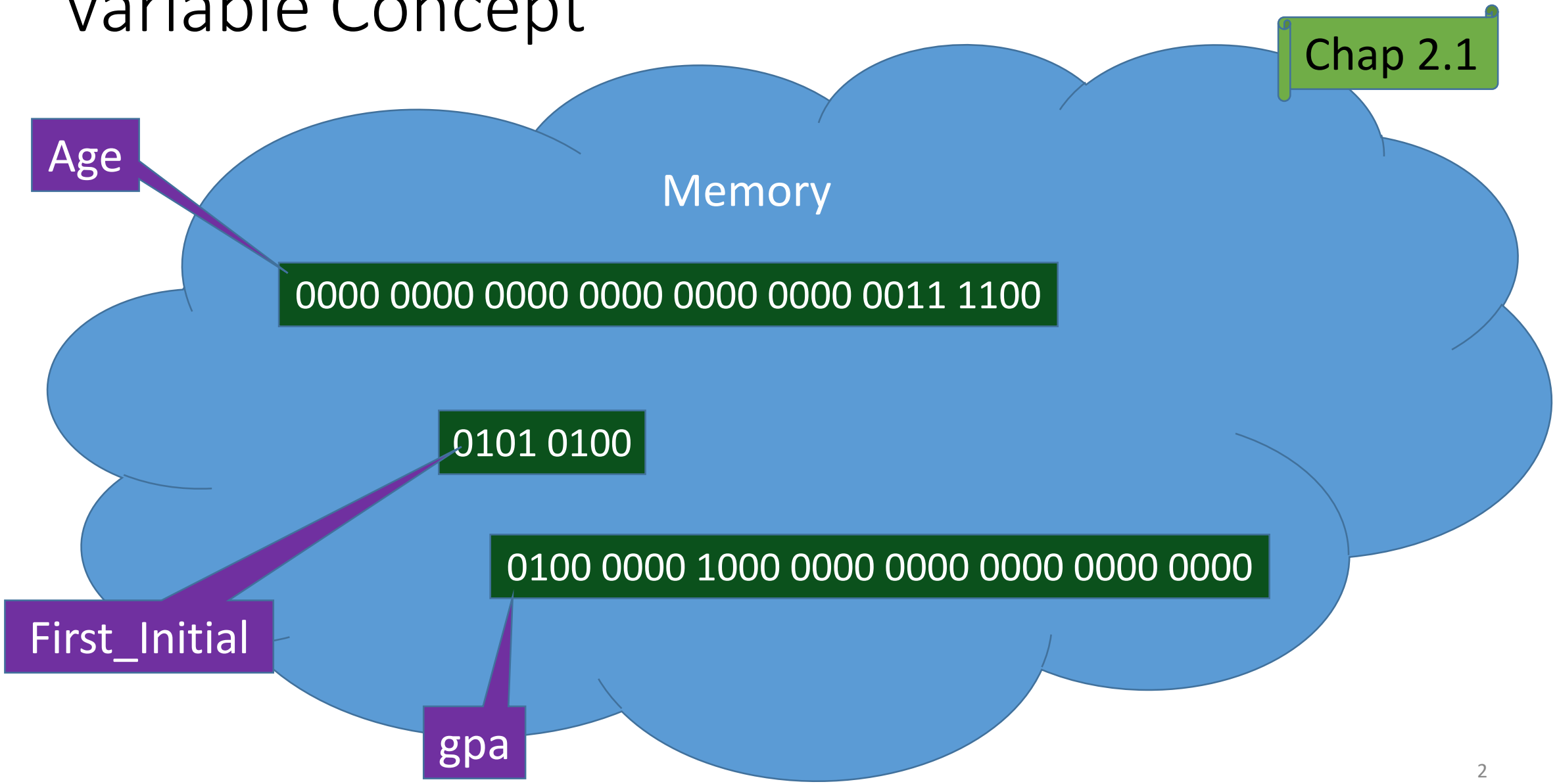


Data in C: Integers

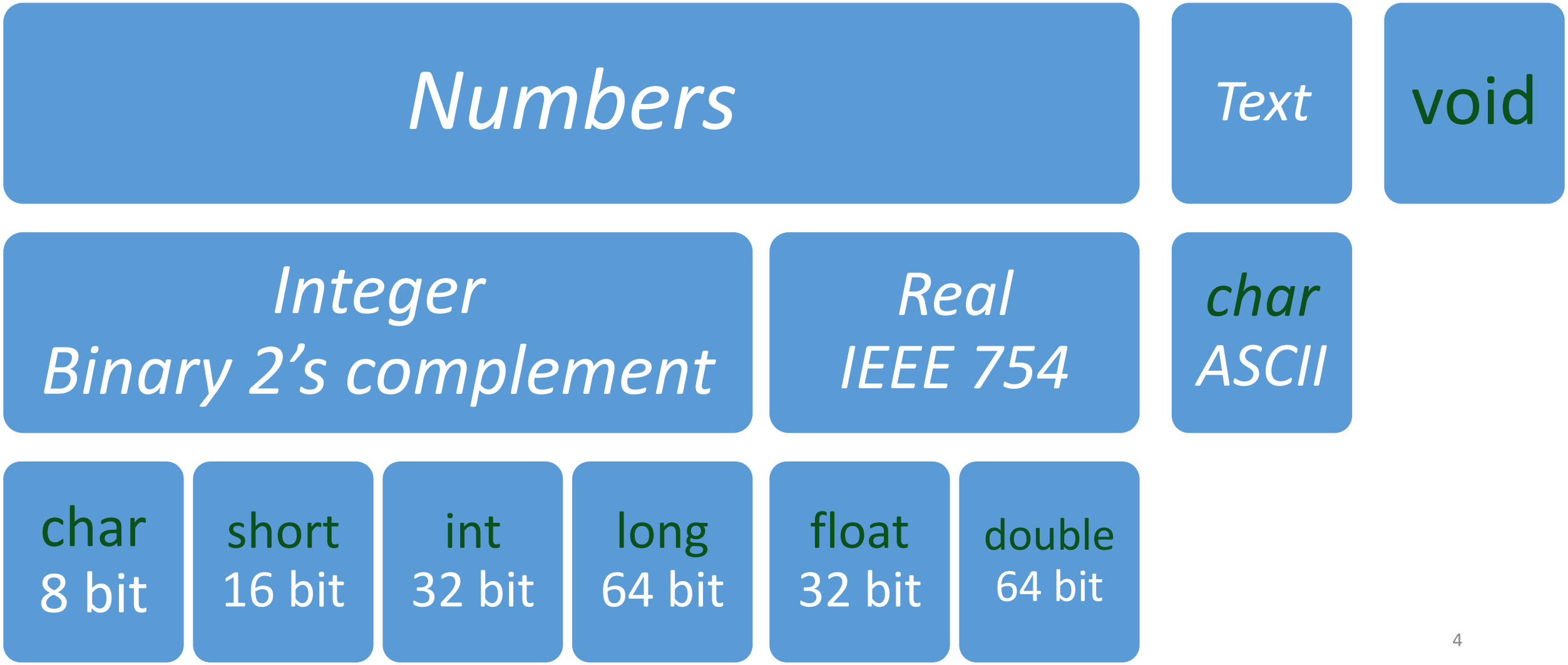
Variable Concept



Data Types

- Tag each piece of data / location in memory with a “type”
- Type tells compiler how many bits to use
- Type tells compiler how to interpret those bits
- Enables automatic conversion from one type to another
- Enables compiler to check to make sure data is used correctly

C Built-in Types



Two's Complement Binary

Chap 2.2

- Number represented by a vector of *n* binary digits (1's or 0's)
- If leftmost bit is a zero, number is positive, simple binary

$$value = \sum_{i=0}^{n-1} b_i \times 2^i$$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	1	0	1	0

- If leftmost bit is a one, number is negative

$$value = \left(\sum_{i=0}^{n-1} b_i \times 2^i \right) - 2^n$$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	1	0	1	0

C Idiom: (Using Leaks) 2^x

- In binary representation, 2^x is just a single 1 in the xth bit
  - e.g. 2^3 = 0b0000 1000 = 8

- Use the "shift" operator, <<, to calculate 2^x

```
int x=3;
int y=1<<x;
printf("y is %d\n",y); // prints y is 8
```

C idiom: (using Leaks) Strings of *x* 1 bits

- Use the fact that $2^x - 1$ is a string of *x* 1 bits
  - e.g. 2^3 = 8 = 0b000 1000
    -1 = 0b000 0001

  - $2^3 - 1$ = 7 = 0b000 0111

- Combine 2^x idiom with -1

```
int x=3;
int y=(1<<x)-1; // y is 0b0000 … 0000 0111
```

Two's Complement Leaky Shortcut

- To multiply by -1, flip the bits and add 1
  - "Flip this bits" is the same as 0b111… 1111 – u_value
    - 1-0=1, flip a zero to a one
    - 1-1=0, flip a one to a zero
    - Never carry
  - 0b111…. 1111 is 2^n-1, so "flip the bits" is (2^n-1)-u_value
    - $u\_value = \sum_{i=0}^{n-1} b_i \times 2^i$
  - flip the bits and add 1 is:

    (2^n-1)-u_value+1=2^n-u_value
    =-(u_value-2^n)
    = -value

C idiom: Bitwise AND to select bits

- Create a “mask” that has 1 bits for every interesting bit, and 0 for every bit you want to ignore
- Bitwise AND the mask with the original value
- Result has the original bit in the interesting positions
  - 0&1=0, 1&1=1
- Result has 0 in the ignorable positions
  - 0&0 = 0, 1&0=0
- Evaluate “truth” of result – “True” means at least one interesting bit was on (1), “False” means all interesting bits were off (0).

Example: Print Binary…

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
        int n=atoi(argv[1]); int i; printf("%d = 0b",n);
        for(i=31;i>=0;i--) {
                printf("%c",(n&1<<i)?'1':'0');
                if (0==i%4) printf(" ");
        }
        printf("\n"); return 0;
}
```

Using Hexadecimal Numbers

- Shorthand for writing bits
- Each hexadecimal digit represents 4 bits
- Much easier to read/write/remember

Dec	Hex	Bin
0	0x0	0b0000
1	0x1	0b0001
2	0x2	0b0010
3	0x3	0b0011
4	0x4	0b0100
5	0x5	0b0101
6	0x6	0b0110
7	0x7	0b0111
8	0x8	0b1000
9	0x9	0b1001
10	0xA	0b1010
11	0xB	0b1011
12	0xC	0b1100
13	0xD	0b1101
14	0xE	0b1110
15	0xF	0b1111

Example: Print hex (without %x)

```
int main(int argc, char **argv) {
        long n=atol(argv[1]);
        int i; printf("%ld = 0x",n);
        unsigned long mask=(1<<4)-1; // Four ones in the rightmost byte
        char * xd="0123456789ABCDEF";
        for(i=2*sizeof(n)-1;i>=0;i--) {
                int v=(n&(mask<<(i*4)))>>(i*4);
                printf("%c",xd[v]);
                if (0==i%4) printf(" ");
        }
        printf("\n"); return 0;
}
```

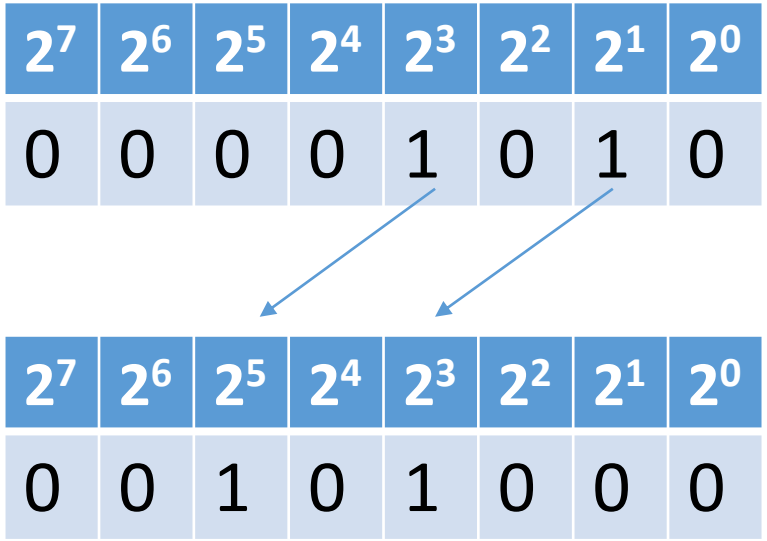
Example: alternate print hex (without %x)

```
int main(int argc, char **argv) {
    union { long ln; unsigned char lc[8]; } n;
    char * xd="0123456789ABCDEF";
    n.ln=atol(argv[1]); printf("%ld = 0x",n.ln);
    for(int i=0;i<sizeof(n);i++) {
        if (i>0 && 0==i%2) printf(" ");
        printf("%c%c",xd[(n.lc[i]&0xF0)>>4],xd[n.lc[i]&0x0F]);
    }
    printf("\n");
    return 0;
}
```

C Leaky Idiom – Multiply by 2^x

- Instead of multiplying a number by a power of two, shift it to the left

```
int x=2;
int y=atoi(argv[1]);
int ym = y<<x; // * ym = y * 2^x
```

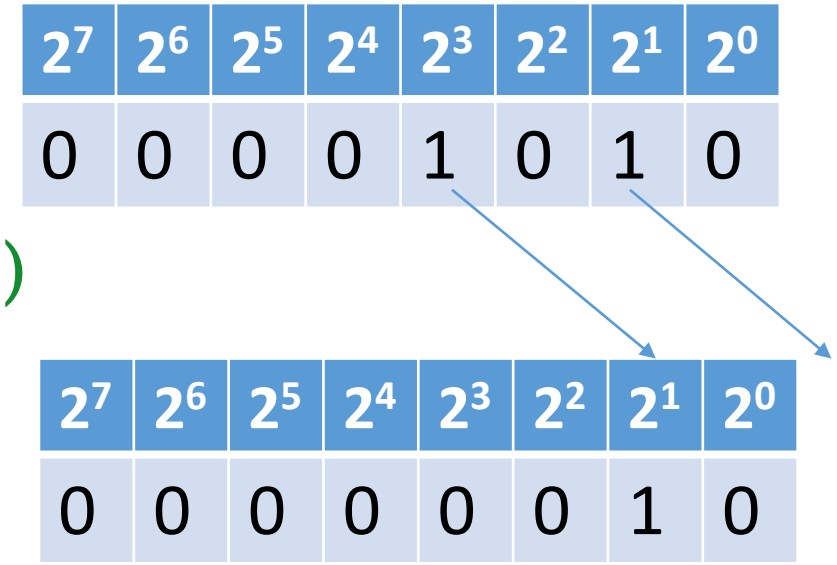


- Shifts are faster than multiplication

C Leaky Idiom – Divide by 2^x

- Instead of dividing by 2^x, shift right by x

```
int x=2;
int y=atoi(argv[1]);
int yd = y>>x; // * yd = y / 2^x (almost)
```



- Warning – negative numbers round differently!
  - Multiplication – rounds towards zero
  - Shifting – rounds down (towards –infinity)

Unsigned Integers

Chap 2.2.2

- C has an “unsigned” keyword that can be used to modify type
  - e.g. unsigned short x;
- Enables representation in raw binary – not two’s complement
- Extra bit enables two-times larger maximum number
  - Not much of a benefit
  - Impossible to represent negative numbers
- Unsigned often leads to counter-intuitive results!
  - More explanation coming later

Q: Why four flavors of Integers?

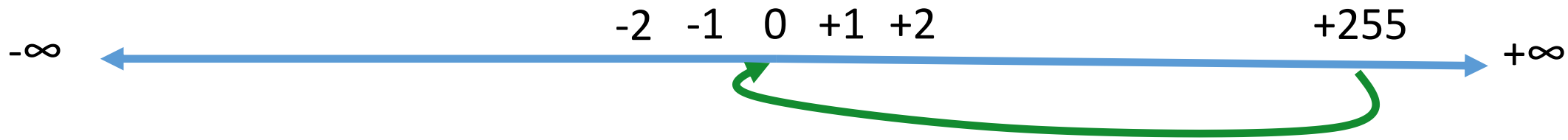
- A: Space vs. Capacity trade-off

Type	# Bits[1]	Min	Max	U Max
char	8	$-2^7=-128$	$2^7-1=127$	$2^8-1=255$
short	16	$-2^{15}=-32,768$	$2^{15}-1=32,767$	$2^{16}-1=65,535$
int	32	$\sim-2.15 \times 10^9$	$\sim2.15 \times 10^9$	$\sim4.3 \times 10^9$
long	64	$\sim-9 \times 10^{18}$	$\sim9 \times 10^{18}$	$\sim18.5 \times 10^{18}$

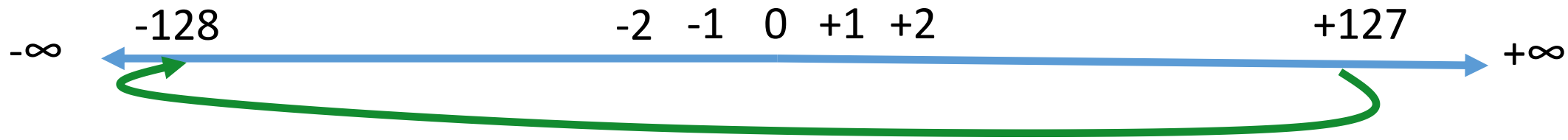
[1]Number of bits may be different on different machines!

Leaky Abstraction: Infinite Integers

- In mathematics, every integer, *n* has a successor, *n+1*
- In C, integers are finite…
  - No indication of overflow!
- For unsigned char, 255=0b1111 1111 + 1 = 0b0000 0000 = 0



- For char, 127 = 0b0111 1111 +1 = 0b1000 0000 = -128



Constants

- Decimal numbers are signed integers: `int x=93;`
- Add "U" suffix to make them unsigned: `unsigned char uy=183U;`
  - Almost never used – only needed if signed value doesn't fit
- Prefix with 0 to interpret as an octal number: `int xa=013; // == 11`
- Prefix with x, 0x, X, or 0X for hexadecimal: `int xb=0x12; //== 18`
- Add "L" suffix to make them long : `long yl = 0X00001111000FFFFL;`
  - Almost never used – only needed if constant value doesn't fit int

Casting : C Data Format Conversion

- When you cast a movie, you decide what role each actor will play
- In programming, we use the term “cast” to indicate what role (type) we expect the bits to play
- In other words, how should your program interpret the specific bits in memory
- Unless explicitly cast, determine the interpretation based on the declared type of a variable

Mixing Integer Types

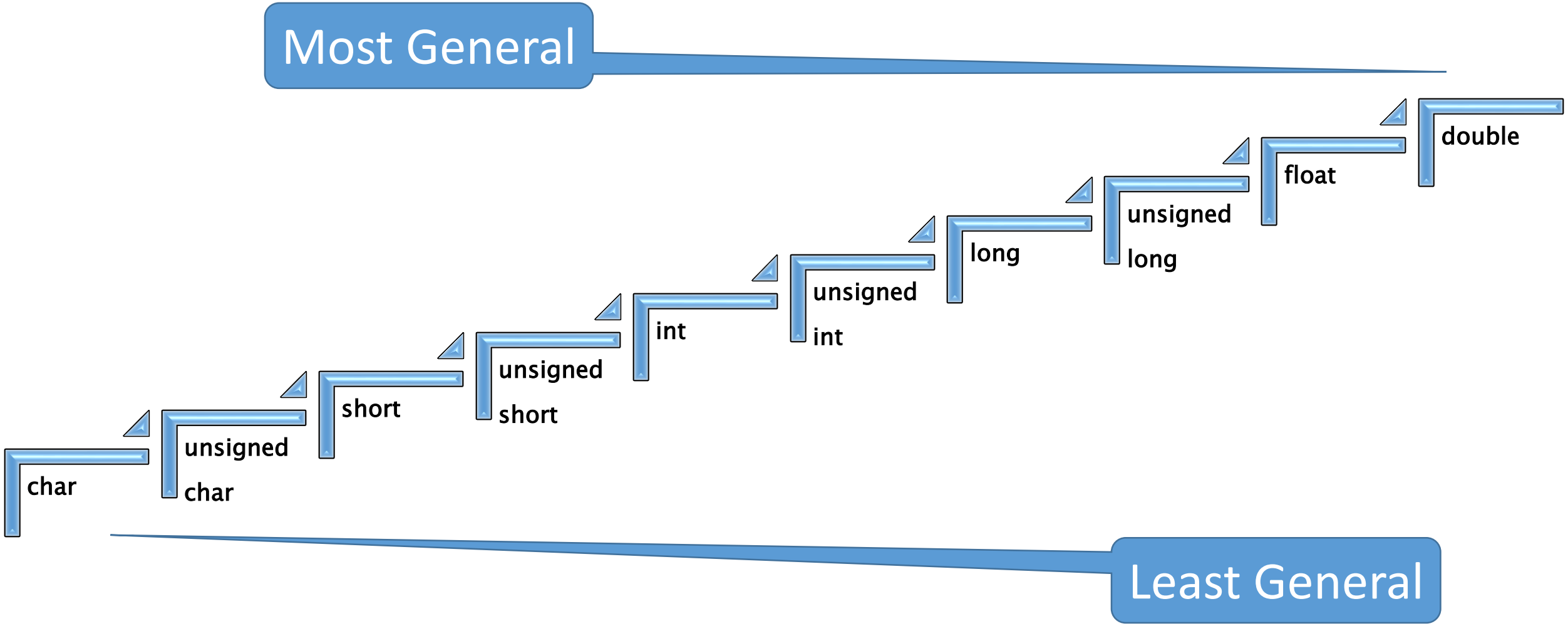
- Mixed Type Expressions
  ```
  int x; long y; y=y*x;
  ```
- Assignment Statements
  ```
  int x; long y; x=y*3.0;
  ```
- Argument Evaluation
  ```
  int myfn(long x); int y=myfn(3);
  ```
- Explicit Casting
  ```
  int x=7; long y = ((long)x)/3;
  ```

- C compiler could issue an error message for mixed types
- C compiler could require explicit casting for all conversions
- C could automatically convert values, based on pre-defined rules

C Automatic type conversion rules

- In an expression (or part of an expression), C converts all components in that expression to the most “general” type, and then evaluates the expression using that general type
- In an assignment (or argument evaluation), C converts the value of the expression to the type of the receiver
- C converts expressions with a valid explicit cast

Generality of Numeric Types



Converting Signed vs Unsigned

Chap 2.2.5

- Bits stay exactly the same, but the bits are INTERPRETTED differently

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	1	0	1	0

```
char x = -22;
unsigned char y=x;
printf("y=%d\n",y);
// prints y=234
```

```
unsigned char w = 234;
char z=w;
printf("z=%d\n",z);
// prints z=-22
```

Changing Integer Size

Chap 2.2.6

- Truncate or Pad on left with sign bit

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	1	0	1	0

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0

```
char x = -22;
short y=x;
```

```
short w = -22;
char z=w;
```

Changing Integer Size (unsigned)

- Truncate or Pad on left with sign bit (=0)

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	1	0	1	0

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	0	0	0	1	1	1	0	1	0	1	0

```
unsigned char x = 234;
unsigned short y=x;
```

```
unsigned short w = 234;
unsigned char z=w;
```

Leaky Abstractions: Conversion Errors

- When C converts a negative signed number to a positive number

```
char x = -1; unsigned char y = x; printf("y = %d\n",y);
```

- When C converts a wide number to a smaller width, but the number doesn't fit

```
short x=260; char y = x; printf("y = %d\n",y);
```

Unsigned Pitfall

```
unsigned int width=+8;
signed int leftX = -13;
if ((leftX < 0) && (leftX + width) > 0)  {
        printf("Rectangle crosses y axis\n");
}
```

Abstraction

Bits are stored in memory from most significant at left

to least significant at right

(int 100,000 = 0x0001 86A0)

2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0
0				0				0				1				8				6				A				0			
Byte m				Byte m+1				Byte m+2				Byte m+3																			

Leaky Abstraction – "Endian"-ness

Chap 2.1.3

Some machines store as expected… (Big–endian)

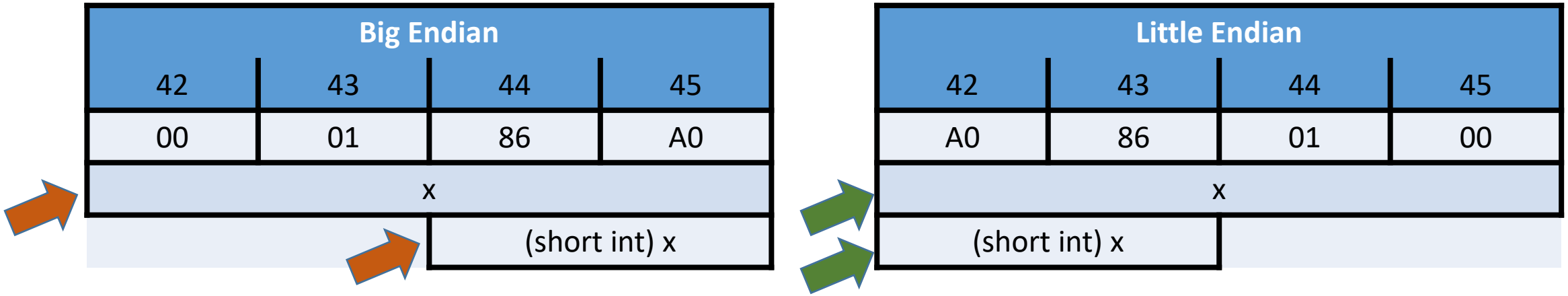
S	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_{9}	b_{8}	b_{7}	b_{6}	b_{5}	b_{4}	b_{3}	b_{2}	b_{1}	b_{0}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0
0				0				0				1				8				6				A				0			
Byte 42								Byte 43								Byte 44								Byte 45							

Some machines store least significant ***byte*** first! (Little-endian)

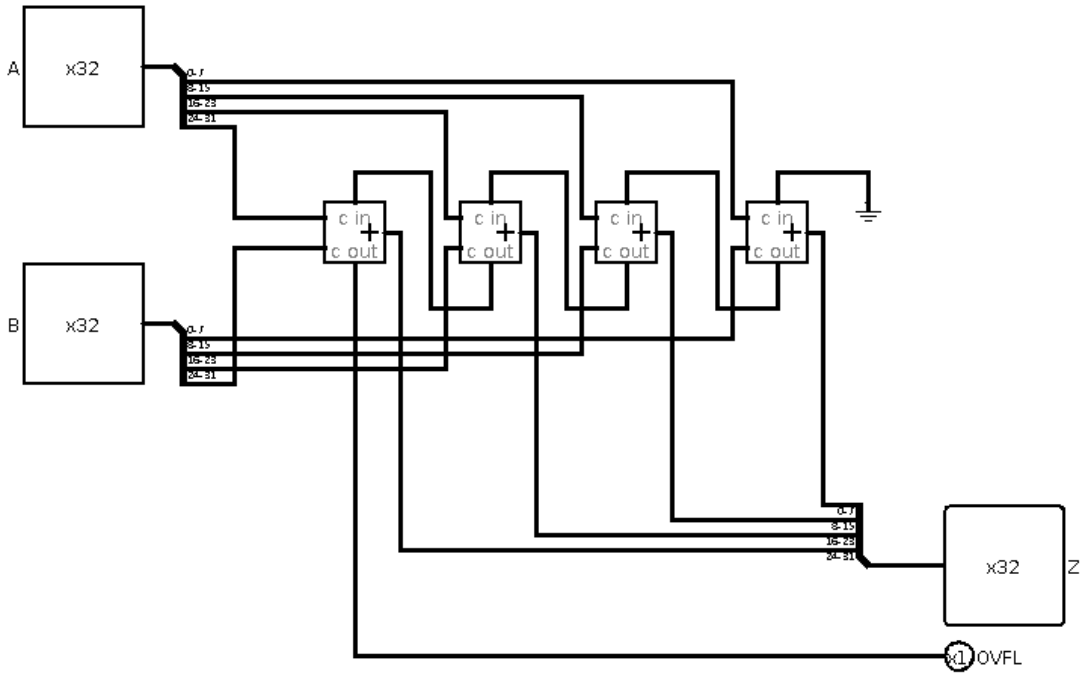
2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	S	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}
b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_{9}	b_{8}	b_{7}	b_{6}	b_{5}	b_{4}	b_{3}	b_{2}	b_{1}	b_{0}
1	0	1	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
A				0				8				6				0				1				0				0			
Byte 42								Byte 43								Byte 44								Byte 45							

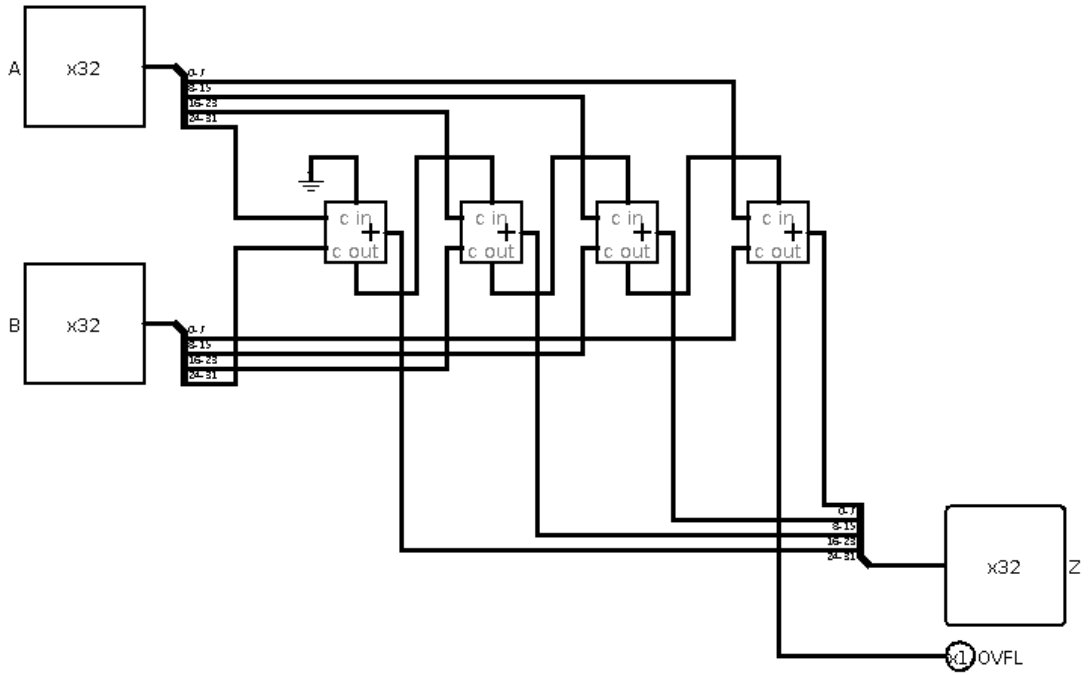
Why Little Endian?

- Casting/Conversion:
  - int x; /* 32 bits starting at byte 42 */
  - y = (short int) x; /* Put the least significant 16 bits from x into y */



Big Endian vs. Little Endian Adder





When does Endian-ness Leak?

- Only on multi-byte data types (short, int, long, float, double)
- Only when we look at the underlying bits or transfer binary data
- Big-endian machine: First byte is the most significant byte
  - Everything works until we get binary data from a little-endian machine
- Little-endian machine: First byte is the least significant byte
  - When hardware interprets the bits/bytes as a number, bytes are switched
  - We don’t even know if a machine is big-endian or little-endian!
  - Until: we get binary data from a big-endian machine OR
  - Until we look at the bit representation of the data, not treated as a number

Managing Endian-Ness

- Network standard is big-endian
- stdlib functions
  - machine representation → network (big-endian) representation
    - htons (short) , htonl (long)
  - Network representation (big-endian) → machine representation
    - ntohs (short), ntohl (long)
  - No-ops when hardware is big-endian
- endian.h functions
  - htobe16, htobe32, htobe64, htole16, htole32, htole64
  - be16toh, be32toh, be64toh, le16toh, le32toh, le64toh

See xmp_endian/network.c

Endian-ness in this Course

- LDAP machines are little-endian
- When you put binary data in a file, you must enter numbers in little-endian format!
- When you print memory from gdb,
  - if you specify byte output formats, data will appear little endian.
  - (When you specify multi-byte numeric output formats, the infrastructure switches the bytes for you.)
- If you make a union of byte-data and multi-byte formats, the byte data will show endian-ness
- Otherwise, everything looks like big-endian!

Resources

- The C Programming Language, (K&R) Sections 2.2 and 2.7
- Wikipedia Type Conversion: https://en.wikipedia.org/wiki/Type_conversion
- C Tutorial – Cast operator: http://www.crasseux.com/books/ctutorial/The-cast-operator.html#The%20cast%20operator
- C-FaQ Cast Operator Section: http://c-faq.com/~scs/cclass/int/sx4bb.html