C Programming Language (1)

Outline

- Compiler drivers
- Assembly code and object code
- Suggested reading
  - 1.2

The Hello Program

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```


The Hello Program

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

The Hello Program

- Source program
  - Created by editor and saved as a text file (Consists exclusively of ASCII characters)
  - Binary file
    - Not text file
- Begins life as a high-level C program
  - Can be read and understand by human beings
- The individual C statements must be translated by *compiler drivers*
  - So that the hello program can run on a computer system

The Hello Program

- The C programs are translated into
  - A sequence of low-level *machine-language* instructions
- These instructions are then packaged in a form
  - called an *object program*
- *Object program* are stored as a binary disk file
  - Also referred to as *executable object files*

The Context of a Compiler (gcc)

hello.c — Source program (text)

↓ **Preprocessor (cpp)**

hello.i — Modified source program (text)

↓ **Compiler (cc1)**

hello.s — Assembly program (text)

↓ **Assembler (as)**

hello.o — Relocatable object program (binary)

↓ **Linker (ld)**

hello — Executable object program (binary)

Preprocessor

Macro Substitution

a.c
```
#include "b.h"

main()
{
    return i+j ;
}
```

b.h
```
#define i  100
int  j ;
```

Obtain with command

`gcc -E a.c`

Source file `a.i`

```
# 1 "a.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "a.c"
# 1 "b.h" 1

int j;
# 2 "a.c" 2

main()
{
 return 100 +j ;
}
```


Source Code and Assembly Code

```
//C code
long mult2(long, long);
void multstore(long x, long y, long *dest)
{
  long t = mult2(x, y);
  *dest = t;
}
```

Obtain with command

`gcc -Og -S mstore.c`

```
Assembly file mstore.s
multstore:
      pushq  %rbx
      movq   %rdx,%rbx
      call   mult2
      movq   %rax, (%rbx)
      popq   %rbx
      ret
```


Relocatable Object Code

```
53 48 89 d3 e8 00
00 00 00 48 89 03
5b c3
```

Obtain with command

`gcc -Og -c mstore.c`

Relocatable object file `mstore.o`

Executable Object Code

```
#include <stdio.h>
void mulstore(long, long, long *)
int main() {
     long d;
     multstore(2, 3, &d);
     printf( "2 * 3 --> %d\n", d);
     return 0;
}

long mult2(long a, long b) {
     long s = a * b ;
     return s;
}
```

```
53 48 89 d3 e8 42 00
00 00 48 89 03 5b c3
```

Obtain with command

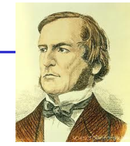
`gcc -Og -o prog main.c mstore.c`

Manipulating Information

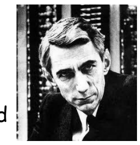
Outline

- Bit-level operations
- Suggested reading
  - 2.1.6~2.1.9

Boolean Algebra



- Developed by George Boole(1815-1864)
  - Algebraic representation of logic
    - Encode "True" as 1
    - Encode "False" as 0
- Claude Shannon(1916–2001)founded the information theory
  - made the connection between Boolean algebra and digital logic
- Plays a central role in the design and analysis of digital systems



Boolean Algebra

And

A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

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


General Boolean Algebras

- Representation of Sets
  - Width w bit vector represents subsets of {0, …, w-1}
  - $a_j = 1$ if $j \in A$
    - 01101001 { 0, 3, 5, 6 }
    - 01010101 { 0, 2, 4, 6 }
  - & Intersection 01000001 { 0, 6 }
  - | Union 01111101 { 0, 2, 3, 4, 5, 6 }
  - ^ Symmetric difference 00111100 { 2, 3, 4, 5 }
  - ~Complement 10101010 { 1, 3, 5, 7 }

Bit-Level Operations in C

- Operations &, |, ~, ^ Available in C
  - Apply to any "integral" data type
    - long, int, short, char
  - View arguments as bit vectors
  - Arguments applied bit-wise

Bit-Level Operations in C

`~0x41`	`0xBE`
`~0100 0001`	`1011 1110`
`~0x00`	`0xFF`
`~0000 0000`	`1111 1111`
`0x69 & 0x55`	`0X41`
`0110 1001 & 0101 0101`	`0100 0001`
`0x69 | 0x55`	`0x7D`
`0110 1001 | 0101 0101`	`0111 1101`

Cool Stuff with Xor

- Bitwise Xor is form of addition
- With extra property that every value is its own additive inverse
  - A ^ A = 0

Cool Stuff with Xor

```
int  inplace_swap(int *x, int *y)
{
   *x = *x ^ *y;    /* #1 */
   *y = *x ^ *y;    /* #2 */
   *x = *x ^ *y;    /* #3 */
}
```

Step	*x	*y
Begin	A	B
1	A^B	B
2	A^B	(A^B)^B = A^(B^B) = A^0 = A
3	(A^B)^A = (B^A)^A = B^(A^A) = B^0 = B	A
End	B	A

Cool Stuff with Xor

```
void reverse_array(int a[], int cnt) {
     int first, last;
     for (first = 0, last = cnt-1;
          first <= last;
          first++,last--)
        inplace_swap(&a[first], &a[last]);
}
```


Mask Operations

- Bit pattern
  - 0xFF
    - Having 1s for the least significant eight bits
    - Indicates the lower-order byte of a word
- Mask Operation
  - X = 0x89ABCDEF
  - X & 0xFF =?
- Bit Pattern ~0
  - Why not 0xFFFFFFFF?

Mask Operations

- Write C expressions that work for any word size $w \geq 8$
- For $x = 0x87654321$, with $w = 32$
- The least significant byte of x, with all other bits set to 0
  - [0x00000021]   x & 0xFF

Mask Operations

- All but the least significant byte of complemented, with the least significant byte left unchanged
  - [0x789ABC21] x ^ ~0xFF
- The least significant byte set to all 1s, and all other bytes of x left unchanged.
  - [0x876543FF]. x | ~0xFF

Logical Operations in C

- Logical Operators
  - &&, ||, !
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination (short cut)

Logical Operations in C

- Examples (char data type)
  - !0x41 --> 0x00
  - !0x00 --> 0x01
  - !!0x41 --> 0x01
  - 0x69 && 0x55 --> 0x01
  - 0x69 || 0x55 --> 0x01

Short Cut in Logical Operations

- a && 5/a
  - If a is zero, the evaluation of 5/a is stopped
  - avoid division by zero
- p && *p
  - Never cause the dereferencing of a null pointer
- Using only bit-level and logical operations
  - Implement x == y
  - it returns 1 when x and y are equal, and 0 otherwise
  - !(x^y)

Shift Operations in C

- 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
<< 3	00010*000*

Argument x	10100010
<< 3	00010*000*

Shift Operations in C

- 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 right
    - Useful with two's complement integer representation (especially for the negative number)

Argument x	01100010
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Shift Operations in C

- What happens ?
  - int lval = 0xFEDCBA98 << 32;
  - int aval = 0xFEDCBA98 >> 36;
  - unsigned uval = 0xFEDCBA98u >> 40;
- It may be
  - lval 0xFEDCBA98 (0)
  - aval 0xFFEDCBA9 (4)
  - uval 0x00FEDCBA (8)
- Be careful about
  - 1<<2 + 3<<4 means 1<<(2 + 3)<<4

bitCount

- Returns number of 1's a in word
- Examples: bitCount(5) = 2, bitCount(7) = 3
- Legal ops: ! ~ & ^ | + << >>
- Max ops: 40

Sum 8 groups of 4 bits each

```
int bitCount(int x) {
   int m1 = 0x11 | (0x11 << 8);
   int mask = m1 | (m1 << 16);
   int s = x & mask;
   s += x>>1 & mask;
   s += x>>2 & mask;
   s += x>>3 & mask;
```


Combine the sums

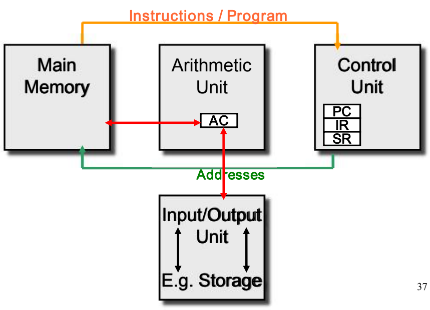
```
  /* Now combine high and low order sums  */
   s = s + (s >> 16);

   /* Low order 16 bits now consists of 4 sums.
      Split into two groups and sum  */
   mask = 0xF | (0xF << 8);
   s = (s & mask) + ((s >> 4) & mask);
   return (s + (s>>8)) & 0x3F;
}
```


Information Storage

Outline

- Pointers and word size
- Suggested reading
  - The first paragraph of 2.1
  - 2.1.2
  - 1.7.3

Computer Hardware - Von Neumann Architecture



Storage

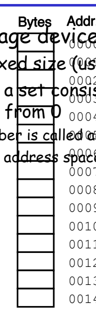
- The system component that remembers data values for use in computation
- A wide-ranging technology
  - RAM chip
  - Flash memory
  - Magnetic disk
  - CD
- Abstract model
  - READ and WRITE operations

READ/WRITE operations

- Tow important concepts
  - Name and value
- WRITE(*name, value*)   value ← READ(*name*)
- WRITE operation specifies
  - a value to be remembered
  - a name by which one can recall that value in the future
- READ operation specifies
  - the name of some previous remembered value
  - the memory device returns that value

Memory

- One kind of storage device
  - Value has only fixed size (usually byte)
  - Name belongs to a set consisting of consecutive integers started from 0
    - The integer number is called address
    - The set is called address space



Word Size

- A virtual address is encoded by
  - a word with fixed size
- Word size indicates the size of such kind of the word
  - Which defines the maximum size of the virtual address space
  - the most important system parameter determined by the word size

Word Size

- For machine with n-bit word size
  - Virtual address can range from 0 to 2^{n}-1
- Most current machines are 64 bits (8 bytes)
  - Potentially address ≈ 1.8 X 10^{19} bytes
- Most current machines also support 32 bits (4 bytes)
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- Unfortunately
  - it also used to indicate the normal size of integer

Data Size

- Machines support multiple data formats
  - Always integral number of bytes

Data Size

C Declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint_32	4	4
int64_t	uint_64	8	8
char *		4	8
float		4	4
double		8	8

intN_t and uintN_t

- Another class of integer types
  - specifying N-bit signed and unsigned integers
  - Introduced by the ISO C99 standard
  - In the file stdint.h.
- Typical values
  - int8_t, int16_t, int32_t, int64_t
  - unit8_t, uint16_t, uint32_t, uint64_t
  - N are implementation dependent

Data Size Related Bugs

- Difficulty to make programs portable across different machines and compilers
  - The program is sensitive to the exact sizes of the different data types
  - The C standard sets lower bounds on the numeric ranges of the different data types
  - but there are no upper bounds

Data Size Related Bugs

- 32-bit machines have been the standard from 1990s to 2010s
- Many programs have been written
  - assuming the allocations listed as "32-bit" in the table
- With the increasing of 64-bit machines
  - many hidden word size dependencies show up as
    - bugs in migrating these programs to new machines

Example

- At the time 32-bit dominated, many programmers assumed that
  - a program object declared as type int can be used to store a pointer
- This works fine for most 32-bit machines
- But leads to problems on an 64-bit machine

Address issues

- IBM S/360: 24-bit address
- PDP-11: 16-bit address
- Intel 8086: 16-bit address
- X86 (80386): 32-bit address
- X86 32/64: 32/64-bit address

64-bit data models

Processors											
4-bit	8-bit	12-bit	16-bit	18-bit	24-bit	31-bit	32-bit	36-bit	48-bit	64-bit	128-bit
Applications											
			16-bit				32-bit			64-bit	
Data Sizes											
nibble octet byte word dword qword											

64-bit data models

Data model	short	int	long	long long	pointers	Sample operating systems
LLP64	16	32	32	64	64	Microsoft Win64 (X64/IA64)
LP64	16	32	64	64	64	Most Unix and Unix-like systems (Solaris, Linux, etc.)
ILP64	16	64	64	64	64	HAL(Fujitsu subsidiary)
SILP64	64	64	64	64	64	?