Computer Organization

October 7, 2015

Contents

1 *<2015-09-03 Thu>*

1.1 C stuff

- C

- C is like a portable assembly
- Very low level
- Does not trash collect automatically

Java	C
OO	Fucntion-oriented
Strongly-typed	can be overrriden
polymorphsim	very limited
classes fro name space	single name space, file oriented
macros are external, rarely used	macros common (preprocessor)
layered I/O model	byte-stream I/O
automatic memory management	function calls
no pointers	pointers (memory adresses) common
*by-reference, by-value	by-value parameters
exceptions, exception handling	if (f() <0) {error} OS signals
concurrency (threads)	library functions
length of array	on your own
string as a type	just bytes (char []) , with 0 end
dozens of common libraries	OS-defined

*by-reference: functions are passed the reference to variable *by-value: functions are passed the value the variable represents

Objects are data and operations on that data.

Interpreted vs Compiled programs

Interepreted	Compiled
less effcient	more effcient
Many layers of abstraction	Closser to the mettle
Portable (targeing the VM)	portable but not as much
Easier to write	Not as easy to write

Executing a C program

- gcc: C compiler
- running programs: gcc [options] [files]
  - **-Wall** prints all warnings. USE ALWAYS.

Macros

- #DEFINE [name] = [value]

- like a variable but more effcient
- compiler literally does a search an replace on text with [name]

C Does not have any bools

- 0 = false
- 1 or non-zero = trie

1.1.1 Variables

-variables have addresses and values

- Memory can be thought of as a large array
- Each location has an address
- a variable is a mapping from a name to an address
  - addresses of a variable can be accessed using the address symbol "&'
    - eg: &x
- can assign addresses using the operator *
  - eg: int *xp (p is a convention for pointer)
    - xp stores address of x

-Can change the value of x via assignment of xp

2 *<2015-09-08 Tue>*

2.1 Bit

- All data is bits
- bit = anything that takes on the value of 1 or 0
  - represented by a 1 or 0
  - Instantiated in hardware via a voltage range (i.e. .9-1.1v = 1, 0-.2V = 0

2.2 Bytes and Words

- **Byte:** A collection of 8-bits
  - 2^8 possible bytes
- **Word:** Default space allocated to things like pointers
  - in modern computers word size is 64 bit (8 bytes)
- **Hexedecimal (hex)**: A base 16 representation
  - A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
- Rightmost number in a binary string is the **Least significant bit**
- Leftmost is **Most Significant bit**
- An int is4 bytes

2.3 Decimal –> Binary

Ex: 42

- 32 is the largest powr of 2 <= 42
  - So theres a 1 in the 32s place
- subtract 32 from 42 and do the same thing w remainder

Ex: 75

- 1 in 64s place: 1000000
  - 75 - 64 = 11
- 1 in 8 place: 1001000
  - 11- 8 = 3
- 1001011 = 75

2.4 Binary –> Hex

- Group Binary into sets of 4
- convert each set of 4bts to a hex bit (4bits = a nibble!)

2.5 Hex –> Decimal

- Hex number denoted by "0x"
  - eg 0x8B2F6
- Convert each digit into 4bits

2.6 Some C definitions

2.6.1 Object

- A distinct region of storage
- Associated with a name

2.6.2 Aliases

Multiple names fro the same object

- Different pointers to the same object are called aliases of each other

2.6.3 Definition

Allocates storage and makes a name for it.

- Ex:
  - int foo;
  - char bar;
- NOTE: The above are defined not initialized.

2.6.4 Decleration

Alerts the compiler that there exusts an object of some name/type, but does nto allocate the space for it.

- Ex.

```
extern int errno;
int func(void)
```

Used when you know your gonna use a func or var from another file but have yet to link them up.

2.7 Object sizes (C)

the function: sizeof(int) will tell you the size of an for ex.

- sizeof does not return an int but a value of tpye $\mathbf{size_t}$, which represents the number of bytes in an object

2.8 Derived types

These are types that you build from the fundemental types (or other derived types)

2.8.1 Arrays

- Defined using []
- Array element are laid out in contiguous memory (in order)
- Elements are accessed by index
- First element is accessed by 0

2.8.2 Structs

Sort of similar to a pyhton object. Lets you associate objects together. ex

```
struct point {
int x;
int y;
int x;
}

\* definition of a point */
struct point p;
```

1. typedefs lets you create a new type.
    - Basically the same as struct except lets you ommit "struct" in decleration

```
typedef struct {
int x;
int y;
```

```
} point;


point p;
```

2.8.3 Unions

Used when you want different representations of the same data.

```
union data {
int intval;
struct {
:w
```

2.8.4 Accessing parts of union/structs

We use dot or arrow operator to access diff parts of union/struct to access the name field in a student struct:

```
student jason, *jasonp;

jason.name /*returns name*/
jasonp->name /*also returns name*/
```

3 *<2015-09-10 Thu>*

3.1 Pointer Arithmetic

- Given pointer, P, to something of type T, P + i is identical to &P[i]
  - P = an address, and i some = i.val * i.size.

Take home: Arrays are closely tied to pointers

3.2 Boolean Algebra

And:

&	0	1
0	0	0
1	0	1

Or

		0	1
0	0	1	
1	1	1	

Xor

^	0	1
0	0	1
1	1	0

These are bit-wise operations, so they are done on the bit level
0110 &1011

0010

3.3 Byte Ordering

How are bytes within a multi-byte wrd ordered in memorY?

3.3.1 Big Endian

Least significant byte has the highest address

3.3.2 Little Endian

x86 ARM, most significant byte has the lowest address

4 *<2015-09-15 Tue>*

4.1 Representing integers

- given n bits, we can represent 2^n values

4.1.1 Overflow

when we have a result that doesnt fit in the n bits we have chosen

- eg. using 4 bits: 0xF + 0x1
  - 0xF = 1111
  - 0x1 = 0001
    - = 10000

4.2 Representing Negative Integers

Three common Encodings

4.2.1 Sign and magnitude

Don't use this because addition and subtraction are v diff from unsigned addition and subtraction

4.2.2 Ones compliment

if integer k is represented by bits $b_1 \ldots b\_$, then -k is represented by $11\ldots11$ - $b_1..b_1$

- This is equivalent to just flipping the bits in k
  - eg. 011 = 3 –> 100 = -3
- the biggest bit is always the negated AND one is added to it ($-2^{(n-1)} + 1$) where n = num of bits
  - 101 –> ($-2^2 + 1$) + 1 = -2
  - to representations of 0:
    - 000 = 0
    - 111 = 0

4.2.3 Two's Compliment

Very similar to ones compliment

- Difference: biggest bit is $-2^{(n-1)}$, not $-2^{(n-1)} + 1$ where n = num of bits
  - $1011 = -2^3 + 2^0 + 2^1 = -5$
    - Biggest value = 0111 = 7, smallest value = 1000 = -8
- How to convert positive int to negative:
  - do ones compliment + 1 (flip bits and add 1)
  - e.g. 0110 –> 1001 + 0001 = 1010 = -6
- Only one 0
- -1 always = $111111111\ldots11$

1. Same implementation of arithmetic operations as unsigned numbers

4.3 Floating point representation

4.3.1 Fractional binary numbers

- 5 3/4 = 101.11 –> = 5 + 1/2 + 1/4
- 2 7/8 = 10.111 = 2 + 1/2 + 1/4 + 1/8

1. Limitations
   - Can only exactly represent numbers of the form x/(2^k)
     - Other rational numbers have repeating bit representations

4.3.2 encodeing

Broken up into threee sections

s (sign 1, or 0) exp (unsigned int with a bias) Frac

5 *<2015-09-17 Thu>*

5.1 Intel x86 Processor

x86-64, the standard architecture

5.2 Architecture

Also known as the instruction set architecture (ISA). The part of the processor design that one needs to understand machine code

5.2.1 Examples

- Intel: x86, IA32, Itanium, x86-64
- ARM: Used in almost all mobile phones

5.3 Microarchitecture

Implementation of the architecture.

- e.g. chache size and core frequency.

5.4 Code forms

5.4.1 Machine Code

Byte level programs that the processor executes

5.4.2 Assembly code

A text representation of machine code.

5.5 Structure

5.5.1 Registers

- Certain registers have certain conventions associated with them
  - %rax for ex stores return values calculated by functions.
- registers that start with "e" is the lower 32-bits ofa given register.
- registers that start with "r" are a full 64 bits.

memory on the CPU that is v small, but v fast

5.5.2 Memory

- byte-adressable array
- where the programs are stored.
- Sends data to CPU based on address stored in the `program counter`

5.5.3 Program counter

A special type of register that points to next instruction to be executed in `memory` (stores the address)

5.5.4 Condition codes

- Stores status information about most recent arithmetic or logical operation

5.6 Assembly characteristics

- integer data of 1 (char), 2 (short), 4(int), or 8(long)
  - pointers are untyped, just addresses.

5.6.1 Operations

- Perform arithmetic function on register or memory data
- Transfers data between memory and register
- Transfer controls
  - if statements, for ex

6 *<2015-09-22 Tue>*

6.1 Assembly Basics

6.1.1 Moving data

`movq source, dest`

- moves a copy of source to a destination register
- source and dest are examples of `operands`

NOTE: Cannot move a value from one memory location to another
Examples

- `movq $0x4, %rax` –> `temp = 0x;`
- `movq $-147 , %rax)` –> `*p = -147;`
- `movq %rax, %rdx` –> `temp2 =temp1;`
- `movq %rax, (%rdx) *p = temp;`

1. Operand types
   (a) Immediate constant integer data
   - denoted by prefixed "$"
     - $0x400, $-533
   - Encoded by up to 4 bytes

   (b) Register
   - 16 in all
   - eg %rax
   - always prefixed byy "%"

(c) Memory

- denoted by "()"
- e.g. (%rax)
- treats whats inside parens register as an address and gets value at that address

6.1.2 Instruction suffixes

Most assembly instructions take instructions

- b (byte: 1 byte)
- e (word: 2 bytes)
- l (long word: 4 bytes)
- q (quad word: 8 bytes)

ex: `movb $-17, %al`
In general only the specific register or bytes are modified

- NOTE: The exception being "l" which will 0 all the uper bits

6.1.3 Normal memory addressing modes

All these things calculate MEMORY ADDRESSES to be accessed

1. Normal (R) Mem[Re[R]]

   - R: register
   - Reg[R]: value at R (address)
   - Mem[Reg[R]]: value in memory at a address Reg[R]

   Equivalent to dereferencing a pointer in C!

2. Displacement D(R) Mem[Reg[R] + D]

   - R: specifies start of memory address
   - D: Add D to value at R

3. Indexed (R_b, R_i) Mem[Reg[$R_{b]}$ + Reg[$Reg_{i]]}$ Calculates the memory address to be accessed by adding values stored at R_b and R_i

   - R_b often spcifies a base memory address

- R_i acts as the index
  `movq (%rcs, %rdx), %rax`

  Good for accessingchar arrays

4. Scaled Index (R_b, R_i, s) Mem[Reg[$R_{b]}$ + Reg[$R_{i]}$* s]
   - s: the scaling factors
   - Must be 1, 2, 4 , 8
   - Allows to iterate through arrays containing vals >= 1 byte
5. Most general form D(R_b, R_i, s) Mem[Reg[$R_{b]}$ + Reg[$R_{i]}$* s + D]
6. Adressing Practice
   - 0x8(%rdx) –> 0x8 + 0xf000 –> 0xf008
   - (%rdx, %rcx) –> 0xf000 + 0x0100 –> oxf100
   - (%rdx, %rcx, 4) —> 0xf000 + 4*0x0100 –> 0xf400

6.2 Arithmetic Instructions

6.2.1 leaq Src, dest

computes the address at src and stores it at dst

7 *<2015-09-24 Thu>*

7.1 Arithmetic operations

7.1.1 ==instruct src,dest

DEST SHOULD ALWAYS COME FIRST

7.2 Condition Codes

7.2.1 Single bit registers

1. CF (Carry flag) set when there's unsigned overflow
2. ZF (zero flag if t == 0
3. SF (sign flag) t < 0
4. OF (overflow flag) Same as CF but for signed numbers

7.2.2 Explicitly set by compare instruction: `cmpq src1, src2`

test b-a without changing dest

7.2.3 Test instruction: test1 sc1, src2

equivalent to a&b but does not change the destination

7.2.4 SetX instruction

- set low-order byte of destination to 0 or 1 based on combination of conditioned codes.
- Does not alter remianing bytes

7.2.5 movzbl

zeroes the upperlevel bytes (excluding the lowest order bytes)

7.2.6 Register mnemonic

Diane Silk Dress Cost $89

- rDi
- rSi
- rDx
- rCi
- r8
- r9

7.3 Conditional Branching

7.3.1 Jumping

Jump to different part of the code depending on condition codes

7.4 Loops

7.5 Switch statement

Allows you define multiple cases (sort of like conds)

7.5.1 Jump table

A bunch of addresses that match up with the case values of the switch function.

- addresses lead to body of each case.

7.5.2 Indirect jump

`jmp *0xfff901`

- star designates that you should jump to the address at address 0xff901

8 *<2015-09-29 Tue>*

COMPUTER DIES. MISSED SOME GOOD SHIT

9 *<2015-10-01 Thu>*

9.1 Function calls

When `callq` calls a function, it pushes its address on the stack so that when the embedded function returns, itll return to the next instruction after callq.

9.2 Function inputs

When you have more than 6 input arguments, the rest are pushed on the stack IN REVERSE ORDER.

- i.e. if you have 9 arguments, the 9th will be pushed first, than 8th, etc.

9.3 Register saving conventions

Registers assigned specific values by the caller are pushed to the stack so that the callee can modify the contents of that register with impunity.

- What if a certain value is not being modified? Will it still be pushed to the stack ? (inefficient)

9.3.1 Caller saved registers

caller-saved registers are used to hold temporary quantities that need not be preserved across calls.

- For that reason, it is the caller's responsibility to push these registers onto the stack if it wants to restore this value after a procedure call.

9.3.2 Callee saved registers

Holds long-lived values that should be preserved across calls.

- rbx, r12-r14, rbp

9.4 Arrays

`T A[L]`

- Array of data type T and length L
- contiguously allocated region of L* sizeof(T) byes of memory

-"A" by itself is an address

- E.g. A –> 0x0ff0345

C does not care if you index thats not in the array!

- e.g. `int val[5];`
  - `val[10]`

9.4.1 Pointer arithmetic

`int val[5];`

- &val –> x
  - val+1 –> x+4 because its an int array
-

9.4.2 Multidimensional arrays

Syntax `int A[R] [C];`

1. Row-major order Rows are laid out in memory sequentially
    - eg |Row 1|Row 2|Row 3| etc.|
    - `A[1] [2];` == row 2, column 3
2. Nested Array Access

9.4.3 Multilevel array

An array of pointers to other arrays

1. Multilevel array access Accessed the same as a nested array
    - first index location of the correct pointer
    - then you access the correct element at the pointer location

9.5 Structure Representation

- represented as a block of memory
- fields ordered according to decleration

9.6 Structures and alignments

Inefficient to load or store datum that spans quad word boundaries

- Therefore alignment rules are enforced for efficiency

9.6.1 Aligned data

A primitive data type of K bytes must have an address that is a multiple of K

- In order to accomplish this, sometimes the compiler adds byte "padding" between fields of a structure.
- Entire struct needs to be aligned properly as well

9.6.2 You can save space (minimize padding) by putt largest data types first in struct.

10 *<2015-10-06 Tue>*

10.1 Buffer Overflow

When exceeding the memory size allocated for an array

- #1 technical cause of security vulnerabilities

Overflowing the buffer will overwrite return address to a function. This means that you overtly change the return address to e your malicious code and reek havoc.

10.1.1 Most common forms

Unchecked lengths of string inputs

10.1.2 Avoiding Buffer Overflows

- Use library routines that limit string lengths

1. System level protection
   (a) Randomized stack offsets
   (b) Nonexecutable code segments Can designate regions of the stack as nonexecutable
   (c) Stack canaries
      - Places a special value between stack and return address.
      - Checks to see if the value has changed after function calls.
      - If it does change, throws an error.