CS61C Course Notes

Anmol Parande

Fall 2019 - Professors Dan Garcia and Miki Lustig

Contents

Disclaimer: These notes reflect 61C when I took the course (Fall 2019). They may not accurately reflect current course content, so use at your own risk. If you find any typos, errors, etc, please report them on the GitHub repository.

1 Binary Representation

Each bit of information can either be 1 or 0. As a result, N bits can represent at most 2^N values.

1.1 Base conversions

Binary to Hex conversion

- Left pad the number with 0's to make 4 bit groups
- Convert each group its appropriate hex representation

Hex to Binary Conversion

- Expand each digit to its binary representation
- Drop any leading zeros

1.2 Numeric Representations

When binary bit patterns are used to represent numbers, there are nuances to how the resulting representations are used.

Definition 1 *Unsigned integers are decimal integers directly converted into binary. They cannot be negative.*

Definition 2 *Signed integers are decimal integers whose representation in binary contains information denoting negative numbers.*

Binary math follows the same algorithms as decimal math. However, because computers only have a finite number of bits allocated to each number, if an operation's result exceeds the number of allocated bits, the leftmost bits are lost. This is called **overflow**

1.2.1 Sign and Magnitude

In the sign and magnitude representation, the leftmost bit is the sign bit. A 1 means the number is negative where as 0 means it is positive. The downside to this is when overflow occurs, adding one does not wrap the bits properly.

1.2.2 One's Complement

To fix the bit wrapping, to form the negative number, we can flip each bit. This solves the wrapping issue so even with overflow, when adding 1, the result will continue increasing.

1.2.3 Two's Complement

To convert from decimal to two's complement: If the number is positive, convert to binary as normal. If the number is Negative:

1. Convert the positive version of the number to binary
2. Invert each bit
3. Add one to the result

If given a two's complement number, you can find -1 times that number by following the same process (invert bits and add 1).

1.2.4 Bias Encoding

With bias encoding, the number is equal to its unsigned representation plus a bias turn. With a negative bias, we can center a positive range of $0 \to 2^N - 1$ on 0.

1.3 Floating Point Representation

To represent decimal numbers in binary, we use floating point representation. Any number in scientific notation has the following components

- Mantissa: The number in front of the point
- Significand: The digits after the point
- Radix: The base of the number
- Exponent: How many times the point should be shifted to recover the original number

$$\underbrace{1}_{\text{Mantissa}} \underbrace{.}_{\text{Binary Point}} \underbrace{01}_{\text{significand}} \cdot \underbrace{2^{\overbrace{-1}^{\text{exponent}}}}_{\text{radix}}$$

If we have a 32 bit system, then by the floating point convention:

$$\underbrace{0}_{\text{Sign Bit}} \quad \overbrace{00000000}^{\text{Exponent Bits}} \quad \overbrace{\underbrace{00000000000000000000000}_{\text{Significand}}}$$

In the floating point representation, the manissa is always one because we are using scientific notation, so we don't bother storing it.

- Sign Bit: Determines the sign of the floating point number
- Exponent: 8 bit biased number (-127 bias)
- Significand: 23 significand bits representing $2^{-1}, 2^{-2}$...

Given a number in floating point representation, we can convert it back to decimal by applying the following formula:

$$n = (-1)^s(1 + significand) \cdot 2^{exponent-127}$$

Notice a couple things about this representation.

- If the exponent is larger than 8 bits, then overflow will occur
- If a negative exponent is more than 8 bits, then underflow occurs
- There are 2 0's. Positive 0 and negative 0

Because of the way that floating point is built, certain sequences are designated to be specific numbers

Exponent	Significand	Object
0	0	0
0	nonzero	denorm
1-254	anything	$\pm$ #
255	0	$\pm\infty$
255	nonzero	NaN

A **denormed number** is a number where we don't have an implicit 1 as the mantissa. These numbers let us represent incredibly small numbers. They have an implicit exponent of -126.

2 C

2.1 C Features

- C is a compiled language $\implies$ executables are rebuilt for each new system
- Every variable holds garbage until initialization
- Function parameters are pass by value

2.2 Bitwise Operators

Bitwise operators are operators which change the bits of integer-like objects (ints, chars, etc)

- &: Bitwise AND. Useful for creating masks
- |: Bitwise OR. Useful for flipping bits on
- $\wedge$: Bitwise XOR. Useful for flipping bits off
- <<: Left shifts the bits of the first operand left by the number of bits specified by the second operand.
- >>: Right shifts the bits of the first operand right by the number of bits specified by the second operand.
- $\sim$: Inverts the bits.

2.3 Pointers

Definition 3 *The address of an object is its location in memory*

Definition 4 *A pointer is a variable whose value is the address of an object in memory*

2.3.1 Pointer Operators

- &: Get the address of a variable
- $*$: Get the value pointed to by a pointer

Because C is pass by value, pointers allow us to pass around objects without having them copied. They can also lead to cleaner, more compact code. However, always remember that declaring a pointer creates space for an object but **does not** put anything in that space (i.e it will be garbage).

Pointers can point to anything, even other points. For example, $int * *a$ is a pointer to an integer pointer. This is called a handle

2.4 Arrays

In C, arrays are represented as adjacent blocks of memory. The way that we interact with them is through pointers. Consider

```
int a[];
int *b;
```

Both of these variables represent arrays in C. They point to the first memory location of the array. C arrays can be indexed into using [] subscripts like other programming languages. They can also be index into using pointer arithmatic. For example, $*(a+2) \equiv a[2]$. By adding 2 to a, C knows to look two memory locations into the array (i.e the third element).

C is able to do this because it automatically computes the size of the objects in the array to know how much to advance the pointer by.

```
int a[] = {1, 2, 3, 4, 5}; //ints are 4 bytes;
printf("%u", &a); // 0x2000
printf("%u", &(a+2)); // 0x2008
char *c = "abcdef"; //chars are 1 byte
printf("%u", &c); // 0x3000
printf("%u", &(a+c)); // 0x3002
```

One important thing to remember is that declared arrays are only allocated while the scope is valid. That means once a function returns, their memory is free to be taken.

Another important consideration in C is that C arrays do not know their own

lengths and they do not check their bounds. This means whenever you are passing an array to a function, you should always be passing its size as well.

2.5 Structs

Structs are the basic datastructures in C. Like classes, they are composed of simpler data structures, but there is no inheritance.

2.5.1 Struct Operators

- $->$: dereference a struct and get a subfield

typedef can be a useful command with structs because it lets us name them cleanly.

3 Memory Management

3.1 Memory Basics

In memory, a **word** is 4 bytes. When objects are saved to memory, they are saved by words. How the words are ordered depends on the type of system. In **Little Endian** systems, the Least Significant Byte is placed at the lowest memory address. In other words, the memory address points to the least significant byte

The opposite is true in **Big Endian** systems. For example, lets say we have the number $0x12345678$ stored at the memory address $0x00$

	$0x00$	$0x04$	$0x08$	$0x0C$
Little Endian:	78	56	34	12
Big Endian:	12	34	56	78

There are four sections of memory: the stack, the heap, static, and code.

Definition 5 *The Stack is where local variables are stored. This includes parameters and return addresses. It is the "highest" level of memory and grows "downward" towards heap memory.*

Definition 6 *The Heap is where dynamic memory is stored. Data lives here until the programmer deallocates it. It sits below the stack and grows upwards to toward it.*

Definition 7 *Static storage is where global variables are stored. This storage does not change sizes and is mostly permanent.*

Definition 8 *Code storage is where the "code" is located. This includes preprocessing instructions and function calls.*

Definition 9 *Stackoverflow is when the stack grows so large that it intersects with heap memory. This is mostly unavoidable.*

Definition 10 *Heap pileup is when the heap grows so large that it starts to intersect with the stack. This is very avoidable because the programmer manages it.*

Definition 11 *All of the memory which a program uses is collectively referred to as the address space of the program.*

3.2 The Stack

The stack is named that way because every time a function call is made, a stack frame is created. A stack frame includes the address of the return instruction and the parameters to the function. As the function executes, local variables are added to the frame. When the function returns, the frame is popped off. In this way, frames are handled in Last-In-First-Out (LIFO) order.

Definition 12 *The stack pointer is a pointer which points to the current stack frame.*

Important: Deallocated memory is not cleared. It is merely overwritten later.

3.3 The Heap

The heap is a larger pool of memory than the stack and it is not in contiguous order. Back to back allocations to the heap may be very far apart.

Definition 13 *Heap fragmentation is when most of the free memory in the heap is in many small chunks*

Fragmentation is bad because if we want to allocate space for a large object, we may have enough cumulative space on the heap, but if none of the remaining contiguous spaces are open, then there is no way to create our object.

Implementation:

Every block in the heap has a header containing its size and a pointer to the next block. The free blocks of memory are stored as a circular linked list. When memory needs to be allocated to the heap, this linked list is searched. When memory is freed, adjacent empty blocks are coalesced into a single larger block. There are three different strategies which can be used to do this allocation/freeing.

Definition 14 *Best-fit allocation is where the entire linked list is searched to find the smallest block large enough to fit the requirements*

Definition 15 *First-fit allocation is where the first block that is large enough to fit the requirement is returned*

Definition 16 *Next-fit allocation is like first-fit allocation except the memory manager remembers where it left off in the free list and resumes searching from there.*

3.4 Heap Management

As a C programmer, it is up to us to manage the heap memory. This is done through the *malloc* function.

```
malloc(size_t size)
```

Malloc takes a size in bytes and returns a pointer to the allocated memory in the heap. If there is no space in the heap, then it returns $NULL$. Any space created with *malloc* must be freed using the $free$ function.
Things to watch out for:

- Dangling reference (A pointer used before malloc)
- Memory Leak (When $free$ isn't called or the pointer is lost but $free$ wasn't called)
- Do not free the same memory twice
- Do not use free on something not created with malloc
- malloc does not overwrite what was in the current memory location.

Because we need to tell *malloc* how much space we need allocated, we will often use the $sizeof$ function. This returns the size in bytes of whatever type or object you give it. This allows us to write code for different architectures (i.e 32bit vs 64bit)

4 RISC-V

4.1 Basics of Assembly Languages

In Assembly Languages like RISC-V, instructions are executed directly by the CPU. The basic job of the CPU is to taken a set of instructions and execute them in order. The **Instruction Set Architecture** (ISA) determines how this is done. In **Reduced Instruction Set Computing** (RISC) languages, the instruction set is limited because it makes the hardware simple. Complex operations are left to the software.

Registers are the operands of assembly language operations. A register is a memory location on the CPU itself. There are a limited number of them, but they are very fast. In RISC-V processors, there are 32 registers. Each register can store one **word**. This is 32 bits on a 32-bit system.

Registers are labeled x0-x31. x0 is a special register because it always holds

the value 0. Unlike variables, registers have no types. The operation is what determines what the content of the register is treated as.

Immediates are numerical constants. They can also be used as the operands of assembly intructions.

4.2 RISC-V Structure

The general format of a RISC-V instruction is

```
operation_name destination source1 source2
```

Labels are text in the program which denote certain locations in code. **Branches** change the flow of the program, usually by jumping to a label or an address in the code portion of memory.

Pseudo-instructions are instructions which are translated into different instructions by the assembler. They exist because they increase readability of the program.

In order to increase program legibility, labels are hardly referred to by their number (x15, x20, etc). Instead, they have symbolic names. Here are a few.

- a0 - a7: The argument registers
- s0 - s7: The saved registers
- ra: return address register
- sp: stack pointer register
- pc: Program Counter

4.2.1 Caller Callee Convention

Because functions can always overwrite registers, programmers set up conventions for calling and returning from functions. The **Caller** is the function which calls another function. The **Callee** is a function which is being called.

Functional Control Flow

1. Put paramters where the function can access them (a0-a7)
2. Transfer control to the function (jump)
3. Acquire the local storage resources for function (Increase stack)
4. Perform the desired task of the function
5. Put result in $a0$ where the calling function can access it

6. Release local variables and return data to used registers so the caller can access them (Decrease stack)

7. Return control to the calling function (Jump to ra)

Because every function must have this control flow, the caller-callee convention was set up as follows

1. sp, gp, tp, s0-s11 are preserved across a function call

2. t0-t7, a0-a7 are not preserved across a function call

If a register is not preserved across a function call, then the caller cannot expect its value to be the same after the callee returns. In order to preserve registers across a function call, we use the **stack**. The Stack Frame stores the variables which need to be saved in order to adhere to caller callee convention. Every RISC-V function has a **Prologue** where it saves the neccessary registers to the stack. An example might look like

```
addi sp, sp, -16
sw s0, 0(sp)
sw s1, 4(sp)
sw s2, 8(sp)
sw ra, 12(sp)
```

This function must use the s0-s3 saved registers. We first create the stack frame by decrementing the stack pointer. Then we save the saved registers to the newly allocated memory. This function must be calling another function, so it has to remember its return address. That is why we save it to stack. The **Epilogue** is the part of the function before it returns where everything on the stack is put back.

```
lw s0, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
lw ra, 12(sp)
jr ra
```

4.2.2 Directives

Directives are special demarcations in an assembly file which designated different pieces of data.

.text: Subsequent items are put into the text segment of memory (i.e the code)

.data: Subsequent items are put into the data segment of memory (i.e static variables)

.global sym: Declares a symbol to be global, meaning it can be referenced from other files

.string str Stores a null-terminated string in the data memory segment

.word Store n 32 bit quantities into contiguous memory words.

4.3 Instruction Formats

In a **Stored Program Computer**, instructions are represented as bit patterns. They are stored in memory just like data. As a result, everything has a memory address, including lines of code. The **Program Counter** is the memory address of the current instruction. Each instruction in RISC-V is 1 word, or 32 bits. It is divided into fields, each of which tells the processor something about the instruction.

R Type: Register-register arithmetic

I Type: Register-immediate arithmetic

S Type: Store instructions

B Type: Branch instructions

U Type: 20 bit upper immediate instructions

J Type: Jump Instructions

Every instruction type has a 7 bit **opcode** which tells the processor what instruction type it is processing. These are always the last 7 bits. Some instructions also have **funct3** and **funct7** fields which define the operation to perform in conjunction with the opcode.

R-Type

31:25	24:20	19:15	14:12	11:7	6:0
funct7	rs2	rs1	funct3	rd	opcode

I-Type

31:20	19:15	14:12	11:7	6:0
Imm[11:0]	rs1	funct3	rd	opcode

S-Type

31:25	24:20	19:15	14:12	11:7	6:0
Imm[11:5]	rs2	rs1	funct3	Imm[4:0]	opcode

B-Type

31	30:25	24:20	19:15	14:12	11:8	7	6:0
Imm[12]	Imm[10:5]	rs2	rs1	funct3	Imm[4:1]	Imm[11]	opcode

U-Type

31:12	11:7	6:0
Imm[31:12]	rd	opcode

J-Type

31	30:21	20	19:12	11:7	6:0
Imm[20]	Imm[10:1]	Imm[11]	Imm[19:12]	rd	opcode

4.3.1 Addressing

Notice that the J-Type and B-Type instructions require use a label in code. These labels are encoded in the instruction format as an offset from the program counter. This is known as **PC Relative Addressing**. Since each instruction in RISC-V is 1 word, we will never have an odd address. As a result, we don't need to store the last bit of the immediate in B-Type and J-Type instructions because it is automatically 0.

4.4 Compiling, Assembling, Linking, Loading

Compiling, Asssembling, Linking, and Loading (**CALL**) are the four steps of loading a program.

4.4.1 Compiler

The input to the compiler is a file written in a high level programming language. It outputs assembly language code which is built for the machine the code was compiled on. The output of the compile may still include pseudo-instructions in assembly.

4.4.2 Assembler

The Assembler is the program which converts assembly language code to machine language code. It reads and uses directives, replaces psuedo-instructions with their real equivalent, and produces machine language code (i.e bits) where it can.

The output of the assembler is an object file. The object file contains the following elements:

- Object file header: The Size and position of the different sections of the object files
- Text Segment: The code
- Data Segment: binary representation of the static data in the source
- Relocation Table: A special data structure which contains the lines of code needing to be fixed

- Symbol Table: A special data structure which lists the files global labels and static data labels
- Debugging information

The symbol table contains information which is public to other files in the program.

- Global function labels
- Data Labels

The Relocation Table contains information that needs to be relocated in later steps. It essentially tracks everything that the Assembler cannot directly convert to machine code immediately because it doesn't have enough information.

- List of labels this file doesn't know about
- List of absolute labels that are jumped took
- Any piece of data in the static section

When the assembler parses a file, instructions which don't have a label are converted into machine language. When a label is defined, it's position is stored in the relocation table. When a label is encountered in code, the assembler looks to see if it's position was defined in the relocation table. If it is found, the label is replaced with the immediate and converted to machine code. Otherwise, the line is marked for relocation.

In order to do its job, the assembler must take two passes through the code. This is because of the **forward reference problem**. If a label is used before it is defined in the same file, the first time the assembler encounters it, it won't know how to convert it to machine code. To resolve this, the assembler simply takes two passes so it finds all labels in the first pass and convert the lines it originally couldn't in the second pass.

4.5 Linker

The Linker is responsible for linking all of the object files together and producing the final executable. The linker operates in three steps:

1. Put the text segments together
2. Put the data segments together
3. Resolve any referencing issues

After the linking step, the entire program is finally in pure machine code because all references to labels must be resolved. The linker knows the length of each text and data segment. It can use these lengths to order the segments appropriately. It assumes that the first word will be stored at $0x10000000$ and can calculate the absolute address of each word from there. To resolve references, it uses the relocation table to change addresses to their appropriate values.

- PC-Relative Addresses: Never relocated
- Absolute Function Addresses: Always relocated
- External Function Addresses: Always relocated
- Static data: Always relocated

When the linker encounters a label, it does the following.

1. Search for the reference in the symbol tables
2. Search for the reference libraries if the reference is not in the symbol tables
3. Fill in the machine code once the absolute address is determined

This approach is known as **Static Linking** because the executable doesn't change. By contrast, with dynamic linking, libraries are loaded during runtime to make the program smaller. However, this adds runtime overhead because libraries must be searched for during runtime.

4.6 Loader

The Loader is responsible for taking an executable and running it.

1. Load text and data segments into memory
2. Copy command line arguments into stack
3. Initialize the registers
4. Set the PC and run

5 Synchronous Digital Systems

Synchronous Digital Systems are the systems which run in our computers. They are **Digital** because electric signals are interpreted as either a 1(asserted) or a 0 (unasserted). They are **Synchronous** because all of the operations are coordinated by a clock. The clock is a device in the computer which alternates between being 1 and 0 at a regular interval, called the **clock period**. There are two critical elements in SDS.

1. Combinational logic element: An element whose output is only a function of the inputs
2. State Element: An element which stores a value for an indeterminate amout of time

Combinational logic elements are used to perform operations on inputs while state elements are used to store information and control the flow of information between combinational logic blocks.

5.1 Registers

Registers are state elements frequently used in circuits. On the rising edge of the clock, the input d is sampled and transferred the output q. At all other times, the input d is ignored. There are three critical timing requirements which are specific to registers

1. Setup Time: How long the input must be stable before the rising edge of the clock for the register to properly read it
2. Hold Time: How long the input must be stable after the rising edge of the clock for the register to properly read it
3. Clock-to-Q Time: How long after the rising edge of the clock it takes for input to appear at the registers output