01:198:211 - Unified Notes

Pranav Tikkawar

February 16, 2025

Contents

1 Intro **3**
- 1.1 Main components . . . 3
- 1.2 CPU . . . 3
- 1.3 Moore's Law . . . 3
- 1.4 Summary . . . 3

2 Unit 1 **4**
- 2.1 Intro to C . . . 4
- 2.2 Comments . . . 4
- 2.3 Variable Declaration . . . 4
- 2.4 Basic Data Types, Operators, and Expressions . . . 4
- 2.5 Control Statements . . . 5
- 2.6 Functions . . . 6
- 2.7 Input and Output . . . 6
- 2.8 Memory . . . 7
- 2.9 Pointers . . . 7
- 2.10 Type Casting . . . 7
- 2.11 Arrays . . . 7
- 2.12 Pointer Arithmetic . . . 8
- 2.13 Structs . . . 8
- 2.14 Dynamic Allocation . . . 8
- 2.15 Type Def . . . 9
- 2.16 File I/O . . . 9

3 Unit 2a **10**
3.1 Multidimensional Array 10
3.2 Compilation 10
3.3 Number Theory 10
3.3.1 Base 10 10
3.3.2 Base n 10
3.3.3 Base 2 10
3.3.4 Base 16 10
3.3.5 Base 8 10
3.3.6 Base n Rationals 10
3.4 Data Sizes 11
3.5 Overflow 11
3.6 Negative numbers 11
3.6.1 Sign-Magnitutde 11
3.6.2 1s complement 11
3.6.3 2s complement 11
3.6.4 un/signed overflow 11
3.7 Bit Shifting 12
3.7.1 Left Shift 12
3.7.2 Right Shift 12
3.8 Floating point 12
3.8.1 IEEE 754 12
3.8.2 Special Values 12

4 Unit 2b **13**
4.1 Assembly 13
4.1.1 Assembly Characteristics 13
4.1.2 x86 13
4.1.3 Instruction formatting 13
4.1.4 Machine Representaion 13
4.1.5 Registers 13
4.1.6 instructions 14
4.1.7 Stack Operations 14
4.1.8 Arithmetic Operations 14

1 Intro

1.1 Main components

- CPU
- Memory
- Bus
- I/O devices:

  Human interface devices

  Storage

  Networking

  Graphics

Von Neumann Model:

1.2 CPU

Fetch → Decode → Execute → Fetch Running on Hardware:

- High level $x = x + y$
- Assembly mov $-0x8(\%esp), \%eax$, add $\%ebx, \%eax$
- Machine Language 0x8B, 0x44, 0x24, 0x08

1.3 Moore's Law

Observed that the number of transistors on a chip double every 18 months.
Processor speeds double every 18 months.
Memory capacity doubles every 2 years.
Disk capacity doubles every year.

1.4 Summary

Modern systems are more complex
Don't expect speed up for single thread programs
Understanding the systems is curicual: Trade-offs, CPU-memory gap, power wall, etc.

2 Unit 1

2.1 Intro to C

C is closer to the machine so easier to see mapping The anatomy of a C program:

- include files: #include <stdio.h>
- declaration of global variables: `char cMessage[] = "Hello\n";`
- comment: `/*comment here*/`
- One or more function; each program starts with execution a "main": `int main() {...}`
- Declearation of local variables: `int i;`
- code implementing functions: `printf("Hello World\n");`

2.2 Comments

Begin with /* and end with */
Can span multiple lines

2.3 Variable Declaration

Variables are used as names for data items.
Each variable has a type which tells the compiler how the data is to be interpreted
Global: Declare outside the scope of any function accessible from anywhere
Local: Declare inside scope of a function accessible only from inode of the function

2.4 Basic Data Types, Operators, and Expressions

Type	Description	Size	Example
char	individual characters	1 byte	'a', 'b', '\t'
int	integers	4 bytes	-14, 0, 3
float	floating point numbers	4 bytes	3.14, 0.0, -1.0
double	double precision floating point numbers	8 bytes	3.14, 0.0, -1.0

Table 1: Basic Data Types

Be careful with type conversions:

Symbol	Operation	Usage	Assoc
*	Multiplication	a*b	L
/	Division	a/b	L
%	Modulus	a%b	L
+	Addition	a+b	L
-	Subtraction	a-b	L

Table 2: Basic Arithmetic Operators

Symbol	Operation	Usage	Assoc
++	Postincrement	a++	L
–	Postdecrement	a–	L
++	Preincrement	++a	R
–	Predecrement	–a	R

Table 3: Special Operators

Symbol	Operation	Usage	Assoc
>	Greater than	a>b	L
>=	Greater than or equal to	a>=b	L
<	Less than	a<b	L
<=	Less than or equal to	a<=b	L
==	Equal to	a==b	L
!=	Not equal to	a!=b	L

Table 4: Relational Operators

Symbol	Operation	Usage	Assoc
!	Logical NOT	!a	R
&&	Logical AND	a && b	L
——	Logical OR	a —— b	L

Table 5: Logical Operators

Symbol	Operation	Usage	Assoc
~	complement	~a	R
&	bitwise AND	a & b	L
—	bitwise OR	a — b	L

Table 6: Bitwise Operators

2.5 Control Statements

Conditionals: `if, else, switch`
Loops: `for, while, do-while`
Specialized "go-to" statements: `break, continue`

if:
`if (condition) {... } else {...}` Evaluates expression untill find first with non-zero value. If all states are false, else is executed.
switch:
`switch (expression) { case constant1: ... break; case constant2: ... break; default: ... }` Finds first case that matches the expression and executes the code. Default always matches always matches.
while:
`while (condition) {...}` zero or more times, while expression is non-zero. Compute expression before iterator.
do-while:
`do {...} while (condition);` one or more times, compute expression after iterator.
for:
`for (initialization; condition; iterator) {...}` zero or more times, while condition is non-zero. computer initialization before iterator.
break:
`break;` exits the loop or switch statement.
continue:
`continue;` skips the rest of the loop and goes to the next iteration.

2.6 Functions

Similar to java:

- Name
- Return Type
- Parameters
- Body

Function call as part of an expression: Arguments evaluated before function call

2.7 Input and Output

printf:
`printf("% n", counter);` String contains characters to print and formatting directives for variables.
scanf:
`scanf("%d", &counter);` String contains formatting directives for variables and addresses of variables.

2.8 Memory

C's memory model matches the underlying virtual memory system: Array of adressable bytes.
Variables are simply names for contiguous sequences of bytes.
Compilers translates names to addresses: typical maps to small address

2.9 Pointers

A pointer is just an address.
Can have variables of type pointer
When declaring a pointer varibale need to the declare the type of the data item the pointer will point to: `int *p;`
Pointer Operators:
`*` - dereference operator: `int x = 5; int *p = &x; int y = *p;`
`&` - address of operator: `int x = 5; int *p = &x;`
Null Pointer:
`int *p = NULL;`

2.10 Type Casting

`int x = 5; double y = (double) x;`
C is not strongly typed:
Type casting allows programmers to dynamically change the type of a data item.

2.11 Arrays

Arrays are contingous sequences of data items.
All elements are of the same type.
Declearation of an array of integers: `int a[10];`
Accessing elements: `a[0], a[1], ...`
Array index always start at 0.
The C compiler and runtime system do not check array boundaries.
Array Storage:
Elements are stored in memory in contiguous locations.
First element (grid[0]) is at the lowest address.
Array and Pointers:
An array name is essentially a pointer to the first element in the array.
`int a[10]; int *p = a;`
First we allocat space for 10 char items. second lines allocats space for a pointer and assigns the address of the first element of the array to the pointer.

a	p	&a[0]
a+n	p+n	&a[n]
*a	*p	a[0]
*(a+n)	*(p+n)	a[n]

Table 7: Equivalences

2.12 Pointer Arithmetic

`int a[10]; int *p = a;`
`p++` increments the pointer by the size of the data type.
`p+n` increments the pointer by n times the size of the data type.
`double x[10]; double *y = x; *(y+3) = 13` This is the same as `x[3] = 13;`

2.13 Structs

Structs are user defined data types.
`struct { int x; int y; } point;`
`point.x = 5; point.y = 10;`
`struct point p; p.x = 5; p.y = 10;`
We can also use arrays of structs.
Pointer to a struct: `struct point *p = &point;`
We also have a special syntax for accessing struct member through a point
Passing Structs as Arguments:
Struct item is passed by value most of the time we want to pass a pointer to the struct.

2.14 Dynamic Allocation

Call stack: the class stack is an area of memory that is used to store information about the currently active functions.
It is useful for recursion
Heap: The heap is an area of memory that is used for dynamic memory allocation.
Malloc: allocates a contigous reigion of memroy of size numBytes if there is enough freespace and returns a pointer to the first byte of the reigion.
`int *p = (int *) malloc(10 * sizeof(int));`
Free: deallocates the memory reigion pointed to by ptr.
`free(p);`

2.15 Type Def

`typedef int Length;`
We can use typedef to define a new type. Mainly for clarity and readability.

2.16 File I/O

fopen `FILE *fopen(char* name, char* mode);`
First arugment is the name of the file, the second is the mode of the file: r- read, w- write, a- append.
fprintf acts the same as printf but writes to a file.
fscanf acts the same as scanf but reads from a file.

3 Unit 2a

3.1 Multidimensional Array

`int m[3][4]` This is an array of 3 arrays of 4 ints
m names the entire the array, m[0] is the pointer to the first row, m[0][0] is the pointer to the first element of the first row
The array index works like: address = address of the array + (row number * row length + column number)*element size **important**
They must be rectangular: where all the rows are the same length.

3.2 Compilation

C is compiled by source and library into the compiler and then it goes out to the executable.

3.3 Number Theory

3.3.1 Base 10

$\sum_{i=0}^{n} a_i 10^i$ where $a_i \in [0, 9]$ This basically is $a_n a_{n-1} a_{n-2} ... a_2 a_1$

3.3.2 Base n

We can do the same thing but replace 10 to any number n.

3.3.3 Base 2

Now we can do the same thing but with just 0s and 1s. ie 101010001

3.3.4 Base 16

Now we can consider 16 digits: 0-9 and A-F
More compact than binary and the conversion is super easy.

3.3.5 Base 8

Now our digits are 0-7 more compact to binary but its kinda mid.

3.3.6 Base n Rationals

The decimal point extends base 10 integers with $1/n, 1/n^2, 1/n^3$

Data Type	16-bit	32-bit	64-bit
char	1	1	1
short int	2	2	2
int	2	4	4
long int	4	4	8
void *	2	4	8
float	4	4	4
double	8	8	8

Table 8: Data Sizes and Ranges

3.4 Data Sizes

3.5 Overflow

For efficieny hardware works with fixed-width integers
Use n bits to represent 2^n
We can use overflow to understand what happens if we have an overflow of addition and multiplication

Wrapping We can say that when we add it is congruent mod 256 for 8 bit addtion.

3.6 Negative numbers

3.6.1 Sign-Magnitutde

The first bit is a negative sign. Required other asthmatic operators. Distinct positive and negative 0s

3.6.2 1s complement

Flip every bit for the negative. First bit still acts like a negative. addition is similar to unsigned addition. Add 1 if you overflow. distinct 0, and -0.

3.6.3 2s complement

Flip all bits and add 1. Addition is the same as unsigned addition. No distinct 0 and -0.

3.6.4 un/signed overflow

Unsigned overflow or carry out ocours when the correct answer is too large to fit in the given number of bits.
This happened when we carry into the sign bit.

3.7 Bit Shifting

3.7.1 Left Shift

Shifts all bits to the left. The leftmost bit is lost and the rightmost bit is filled with 0. This is like multiplying by 2.

3.7.2 Right Shift

Shifts all bits to the right. The rightmost bit is lost and the leftmost bit is filled with 0. This is like dividing by 2.

3.8 Floating point

3.8.1 IEEE 754

32-bit and 64-bit floating point numbers.

Sign	Exponent	Mantissa	Value	Decimal
0	00000000	00000000000000000000000	0	0.0
0	01111111	00000000000000000000000	1	1.0
0	10000000	00000000000000000000000	2	2.0
0	10000000	10000000000000000000000	3	3.0
0	10000000	01000000000000000000000	2.5	2.5

Table 9: Floating Point Numbers

3.8.2 Special Values

We represent 0 as all 0s.
We represent subnormal as the all 0s in the exponent. where the first elemetn of the mantissa is 0.
for infinity we have all 1s in the exponent and all 0s in the mantissa.
NaN is all 1s in the exponent and any non-zero value in the mantissa.

4 Unit 2b

4.1 Assembly

IA32 (X86 ISA)
There are many assembly languages because they are processor specific. **RISC vs CISC**
Risc is simple instruction set and CISC is complex instruction set.
pg 22 has a table to understand the steps of putting the CPU and memory tg

4.1.1 Assembly Characteristics

Minimal data types: Integer, Floating Point, and no aggregate types
No type checking
3 Types of primitive operations: Arithmetic on regster, transfer data between memory and register, control flow

4.1.2 x86

The Characteristics of x86 are:
Variable length instructions
Can adress memory directly in most instructions
uses little-endian byte ordering

4.1.3 Instruction formatting

`opcode operand 1, operand 2`
`movl %eax, %ebx`
Opcode is a short mnemonic for instruction purpose **Operands** are the source and destination of the operation. They can be immediate register or memory.

4.1.4 Machine Representaion

Each assembly instruction is translate to a sequence of 1-15 bytes.
First the binary representor of the opcode.
Second, instructions specific the addressing mode
Some instructions can be single byte because operand an addressing mode are implicitly specified by the instruction.

4.1.5 Registers

General purpose registers are 32 bits
Originally categorized as data register and pointer/index registers.

Data : EAX, EBX, ECX, EDX.
Pointer/Index: EBP, ESP, EIP, ESI, ESI
Segment Registers: CS, DS, SS, ES,
ESP is the stack pointer.
EBP is the base register

4.1.6 instructions

movl - move data from source to destination
Example: `movl %eax, %ebx`
Moves the contents of %eax to %ebx
`ebx = eax`
In immediate Adressing. operand is immediate. Operand value is found immediatly following the isntruction. $ in front of immediate operand.
% denotes register operand.
Example: `movl $eax, %ebx` copy contnet of %eax to %ebx
Example: `movl $0x1, %eax` copy 0x1 to %eax
Example: `movl %eax, 0x1` copy %eax to memory location 0x1
Example: `movl (%ebp, %esi), %eax` copy value at address = ebp + esi to %eax
Example: `movl 8(%ebp, %esi), %eax` copy value at address = ebp + esi + 8 to %eax
Example: `movl 0x80 (%ebx, %esi, 4), %eax` copy value at address = ebx + esi*4 + 0x80 to %eax
This is super import for arrays
pg 41 is a good image!!!

4.1.7 Stack Operations

pushl - push data onto the stack
popl - pop data off the stack
`pushl %eax` - push %eax onto the stack. esp = esp -4; Memory[esp] = eax
`popl %eax` - pop the top of the stack into %eax. eax = Memory[esp]; esp = esp + 4
leal Compute address using addressing mode without accessing memory
`leal src , dest`
It is used for computing adresses without memory refrence.
EG. it is `p = &a[i]`
Example: `leal 7(%edx, %edx, 4), %eax` eax = 4edx+edx+7

4.1.8 Arithmetic Operations

addl - add data from source to destination
Example: `addl %eax, %ebx`
Adds the contents of %eax to %ebx
`ebx = ebx + eax`

subl - subtract data from source to destination
Example: `subl %eax, %ebx`
Subtracts the contents of %eax from %ebx
`ebx = ebx - eax`

imull - signed multiply
Example: `imull %eax, %ebx`
Multiplies the contents of %eax by %ebx
`ebx = ebx * eax`

sall - arithmetic shift left
Example: `sall $2, %eax`
Shifts the contents of %eax left by 2 bits
`eax = eax << 2`

sarl - arithmetic shift right
Example: `sarl $2, %eax`
Shifts the contents of %eax right by 2 bits
`eax = eax >> 2`

xorl - bitwise exclusive OR
Example: `xorl %eax, %ebx`
Performs a bitwise exclusive OR on %eax and %ebx
`ebx = ebx ^eax`

andl - bitwise AND
Example: `andl %eax, %ebx`
Performs a bitwise AND on %eax and %ebx
`ebx = ebx & eax`

orl - bitwise OR
Example: `orl %eax, %ebx`
Performs a bitwise OR on %eax and %ebx
`ebx = ebx | eax`

incl - increment
Example: `incl %eax`
Increments the contents of %eax by 1
`eax = eax + 1`

decl - decrement
Example: `decl %eax`
Decrements the contents of %eax by 1
`eax = eax - 1`

negl - negate
Example: `negl %eax`
Negates the contents of %eax
`eax = -eax`

notl - bitwise NOT
Example: `notl %eax`
Performs a bitwise NOT on %eax
`eax = ~eax`