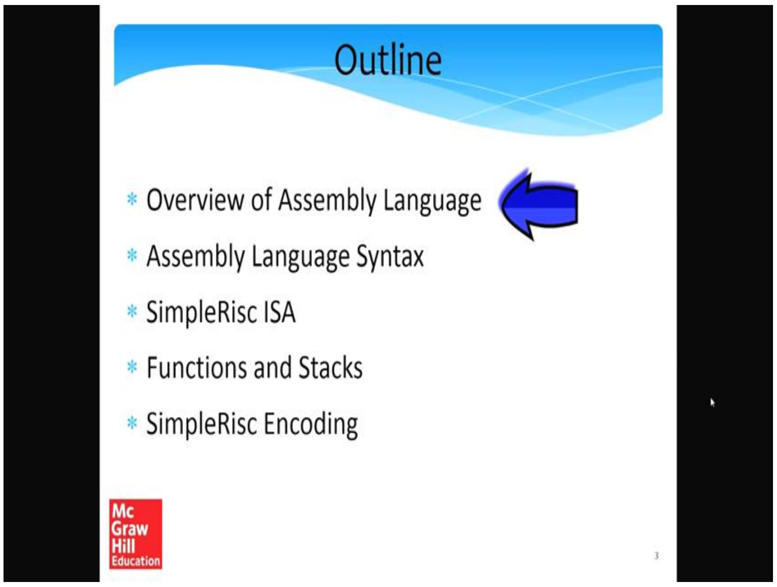
Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 05
Assembly Language Part-1

Welcome to chapter 3. In this chapter we will cover assembly language, which is the vital link between the hardware and software. Just to retreat, this is chapter 3 of the book computer organizational architecture, and the slides are meant to be used along with the book, otherwise the learning will remain incomplete.

(Refer Slide Time: 01:25)



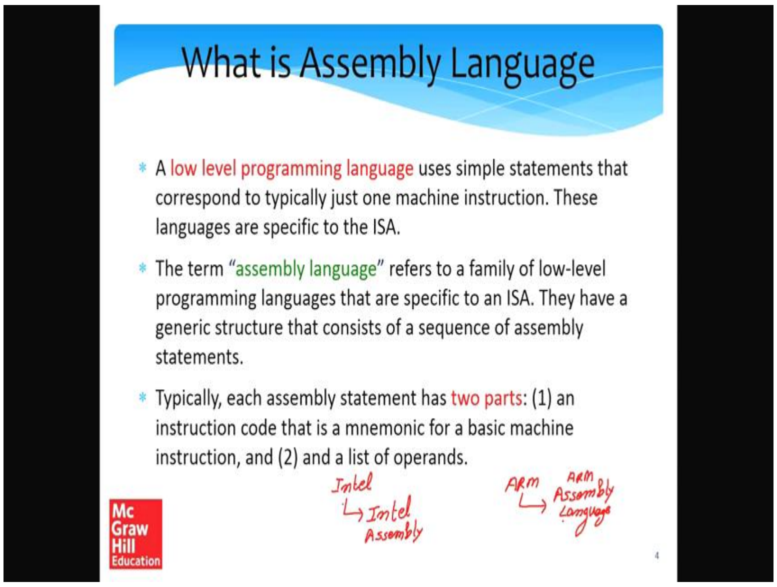
Outline

- Overview of Assembly Language
- Assembly Language Syntax
- SimpleRisc ISA
- Functions and Stacks
- SimpleRisc Encoding

Mc Graw Hill Education
3

So, let us first discuss in the overview of Assembly Language. Subsequently we will discuss the syntax of assembly language. We will design a simple assembly language of our own called the simplerisc assembly language, along with it is instructions in the instruction set architecture. Subsequently we will move on to functions and stacks, an advanced concept in assembly language, and we will discuss the encoding.

(Refer Slide Time: 01:57)



What is Assembly Language

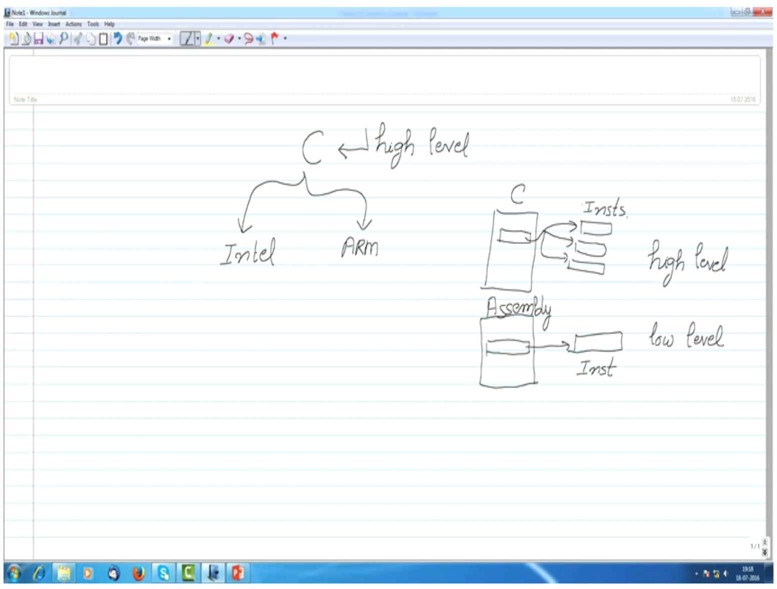
- A low level programming language uses simple statements that correspond to typically just one machine instruction. These languages are specific to the ISA.
- The term "assembly language" refers to a family of low-level programming languages that are specific to an ISA. They have a generic structure that consists of a sequence of assembly statements.
- Typically, each assembly statement has two parts: (1) an instruction code that is a mnemonic for a basic machine instruction, and (2) and a list of operands.

Intel $\hookrightarrow$ Intel Assembly

ARM $\hookrightarrow$ ARM Assembly Language

So, let us fist define what is an assembly language. So, in assembly language, is a low level programming language. So we will discuss what is high level, and what is low level. So, a low level programming language uses very simple statements, where typically one statement corresponds to a single machine instruction. So, these languages are specific to the instruction set architecture and. So, for example; C or java can run on any kind of a processor. However, as an assembly language is specific to only a particular process or a particular family of processors. For example, it is possible to run c.

(Refer Slide Time: 02:53)

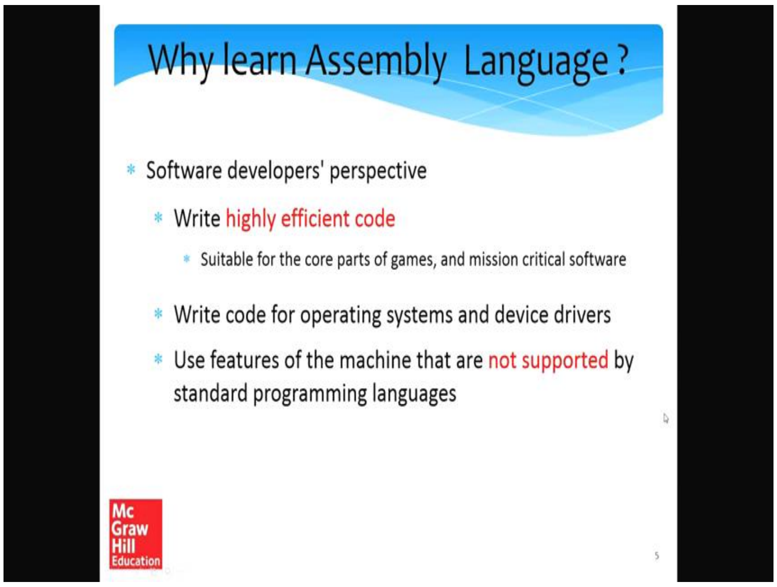


Let us move to the different media. For example, it is possible to run a C program, on both an Intel machine and ARM machine. However Intel has it is separate assembly language syntax, and ARM as it is separate assembly language syntax. So, so we will discuss what it is. So, C is a high level language, because concepts are at a very high and abstract level, and also this is a C program; a single statement in the C program can actually be many machine instructions, not 1, but actually multiple machine instructions, sequence of 0's and 1's, multiple instructions.

Whereas, in an assembly language program, a single statement typically the most often corresponds, to a single machine instruction. So, this is the vital in the most important difference between a high level language and a low level language. So, this C would be an example of a high level language, and an assembly would be an example of a low level language.

So, actually an assembly language is not a single language, it rather refers to a family of low level programming languages. For each instruction set architecture has it is own assembly language. For example, an Intel machine; like the one that is running on the desktop right now, has it is own assembly language. You can call it Intel assembly. Similarly an ARM machine, an ARM processor, which has a separate I S A, separate instruction set architecture, has it is own assembly language right. You can call it ARM assembly language so maybe. So, as it is own separate ARM assembly language. So, different processors, different family of processors have their own assembly languages, and each assembly statement corresponds to typically a single machine instruction. So, we can think of it as a language that is very close to the hardware. So, typical assembly statement has two parts, it has an instruction code, which represents a basic machine instruction, something like add or subtract or multiply, and it has a list of operands.

(Refer Slide Time: 06:13)

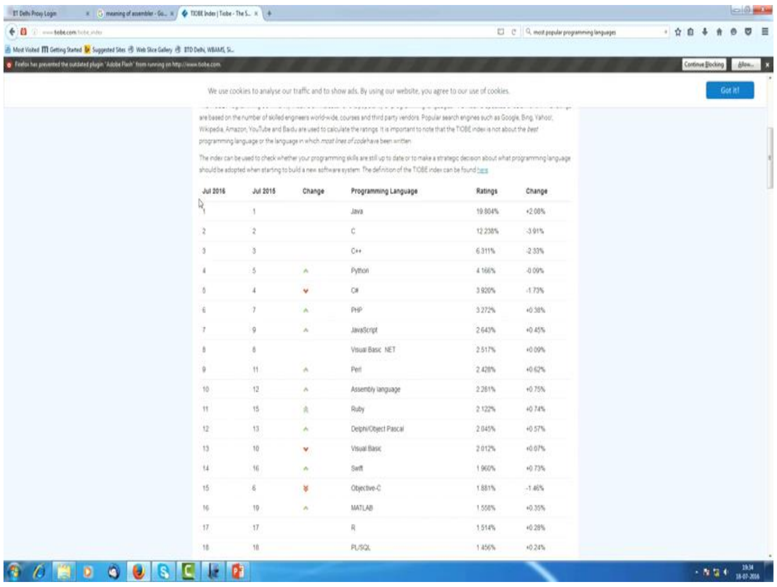


Why learn Assembly Language ?

- Software developers' perspective
  - Write highly efficient code
    - Suitable for the core parts of games, and mission critical software
  - Write code for operating systems and device drivers
  - Use features of the machine that are not supported by standard programming languages

5

(Refer Slide Time: 06:20)



We use cookies to analyse our traffic and to show ads. By using our website, you agree to our use of cookies.

are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the *best* programming language or the language in which *most lines of code* have been written.

The index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The definition of the TIOBE index can be found here.

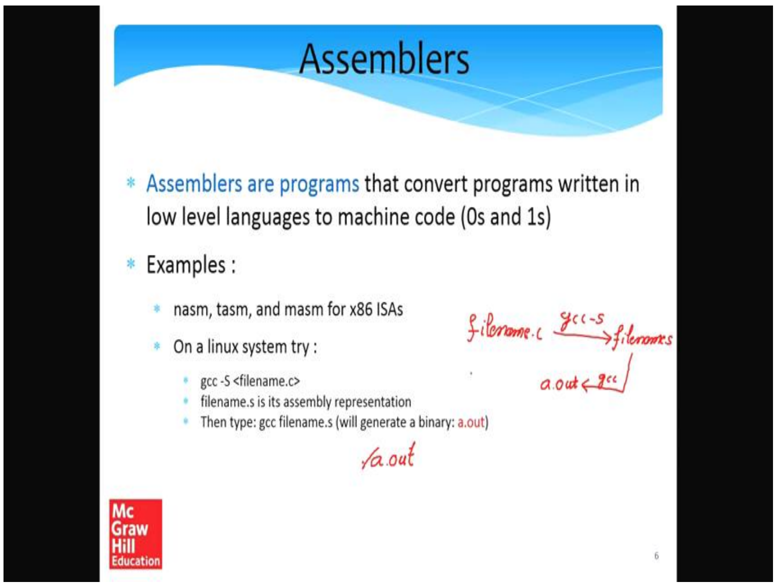
Jul 2016	Jul 2015	Change	Programming Language	Ratings	Change
1	1		Java	19.804%	+2.08%
2	2		C	12.238%	-3.91%
3	3		C++	6.311%	-2.33%
4	5	^	Python	4.166%	-0.09%
5	4	v	C#	3.920%	-1.73%
6	7	^	PHP	3.272%	+0.38%
7	9	^	JavaScript	2.643%	+0.45%
8	8		Visual Basic .NET	2.517%	+0.09%
9	11	^	Perl	2.428%	+0.62%
10	12	^	Assembly language	2.281%	+0.75%
11	15	^^	Ruby	2.122%	+0.74%
12	13	^	Delphi/Object Pascal	2.045%	+0.57%
13	10	v	Visual Basic	2.012%	+0.07%
14	16	^	Swift	1.960%	+0.73%
15	6	vv	Objective-C	1.881%	-1.46%
16	19	^	MATLAB	1.556%	+0.35%
17	17		R	1.514%	+0.28%
18	18		PL/SQL	1.456%	+0.24%

Let us now take a look at the popularity of a assembly. So, I have a browser window open, which shows the popularity of the most common programming languages as of July 2016. So, the first three are usual suspect's java C and C++; that let us scroll down to the 10th entry. The 10th entry is actually assembly language.

Again assembly language is not a single language, it is a family of languages, where each processor family has it is own assembly language, but we see that from July 2015 till July 2016, assembly languages together are the tenth most popular language among all programming languages. So, this means that it is a very important language for students to learn, and they should be familiar with at least one flavor of assembly languages; such that they can program hardware much better, and also understand how compilers and assemblers really work.

So, that is why it makes a case where at least learning these languages. One reason for the popularity of assembly languages is that, we have many small devices today. For example, there are small controllers in almost everything that we use. So, now, there is a smart lights, that send if a user is in the room, turn the light on, otherwise they turn it off. So, this is actually an advantage of, you know very small hardware, and these small pieces of hardware are not programmed using high level languages, rather they are programmed using assembly languages, for the reason for its efficiency.

(Refer Slide Time: 08:05)



Assemblers

* Assemblers are programs that convert programs written in low level languages to machine code (0s and 1s)
* Examples :
  * nasm, tasm, and masm for x86 ISAs
  * On a linux system try :
    * gcc -S <filename.c>
    * filename.s is its assembly representation
    * Then type: gcc filename.s (will generate a binary: a.out)

$$\begin{tikzcd} \text{filename.c} \arrow[r, "\text{gcc -S}"] & \text{filename.s} \arrow[d, "\text{gcc}"] \\ & \text{a.out} \end{tikzcd}$$

./a.out

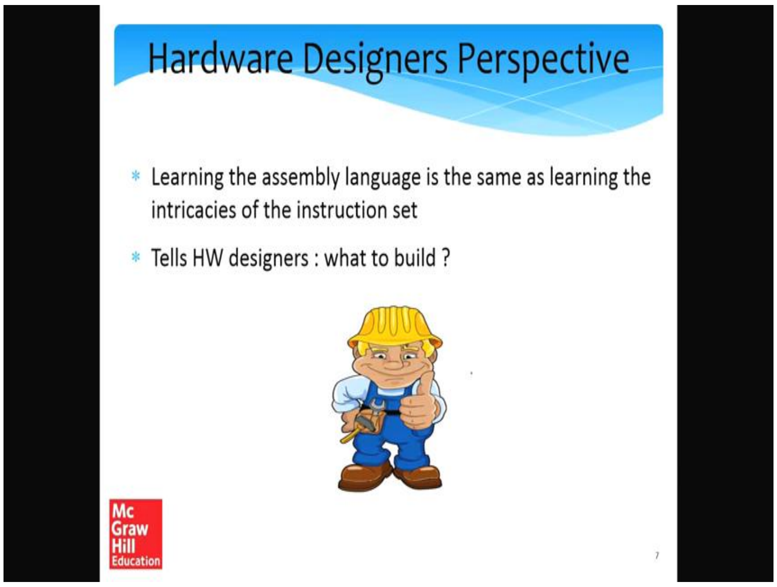
6

So, let us introduce a term now called assemblers. So, assemblers are programs, that convert assembly language programs, written in low level languages to machine code, it is sequence is 0’s and 1’s, essentially convert an assembly language program to a binary. So, examples of these would be for x86: nasm, tasm, and, masm are three very popular assemblers for x86 ISAs. There are similar popular assemblers for ARM as well. The readers are requested to take a look at the website of the book, to get an idea of assemblers for different assembly languages.

So, on a Linux system it is possible to generate an assembly file, it is very easy. So, we need to run this command gcc minus S, and the name of the C file or the name of the C++ file. So, say the name of the C file is file name dot C; it is assembly file name will be file name dot s; that is this assembly representation.

So, what gcc minus s tells is that it converts the C file, it complies it to produce an assembly file, and once an assembly file is generated, you can open it in a text editor, and see the details of assembly code that has been generated. Then to generate the binary, we can run the same gcc command once again, and run the command gcc file name dot s, which will generate binary called a dot out, and then running the binary is very easy, what we need to do is that. So, running the binary is very easy. So, in the Linux it is just as simple as dot slash a dot out. So, in this case what we see is that if we have a C program with a name file name dot c, a compiler converts it to. So, the compiler command is gcc minus S; that converts the C file into an assembly file; that is file name dot s. And the file name dot s can be further converted to a binary which the processer can understand; the sequence of 0's and 1's why I am using the command gcc. So, this holds in the Linux system, it will also hold on a system with mac os x with code blocks installed. On widows there are different commands, I will not get into that.

(Refer Slide Time: 11:03)

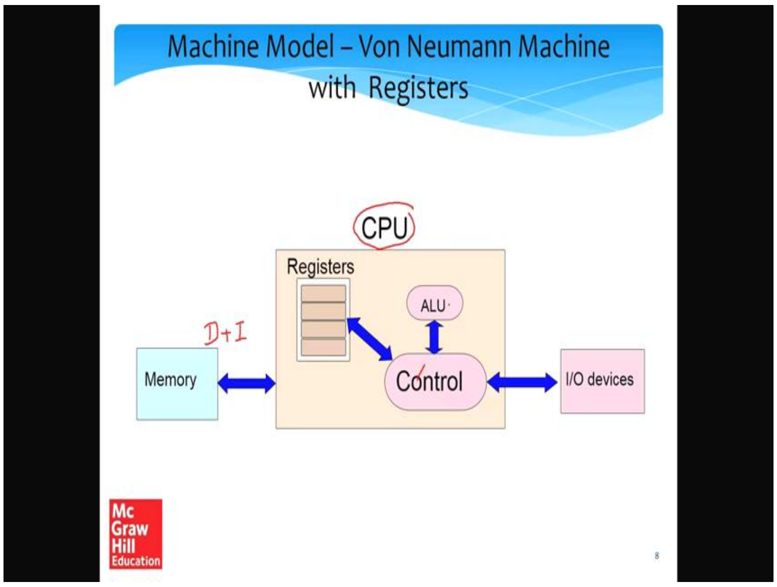


Hardware Designers Perspective

* Learning the assembly language is the same as learning the intricacies of the instruction set
* Tells HW designers : what to build ?

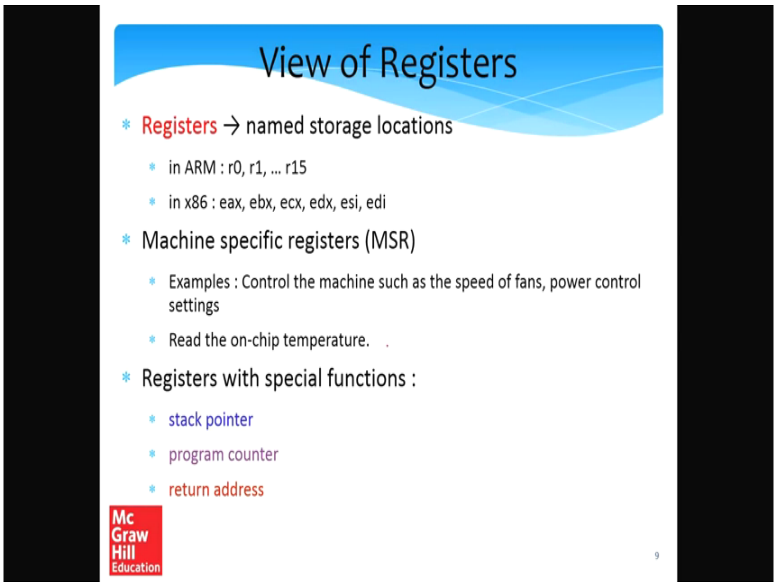
So, let us now take a look at the importance of assembly language in a hardware designer perspective. So, learning the assembly is a same as learning the intricacies of the instruction set, because every assembly language statement corresponds to a single instruction. So, once a hardware designer knows the format of assembly language, he pretty much knows what is to be built. So, what is the interface between software and hardware which makes it easy to design hardware as well?

(Refer Slide Time: 11:37)



So, now let us start with the integrities of assembly language, but let us go back to chapter one, where we had discussed the structure of Von Neumann machine. So, Von Neumann machine had the CPU at the center. So, the CPU was at the center, it was connected to memory the memory contains data as well as data, as well as instructions and if the CPU is also connected to I O devices. Inside the CPU we have a control unit in a arithmetic logic unit, and a set of registers.

(Refer Slide Time: 12:12)



View of Registers

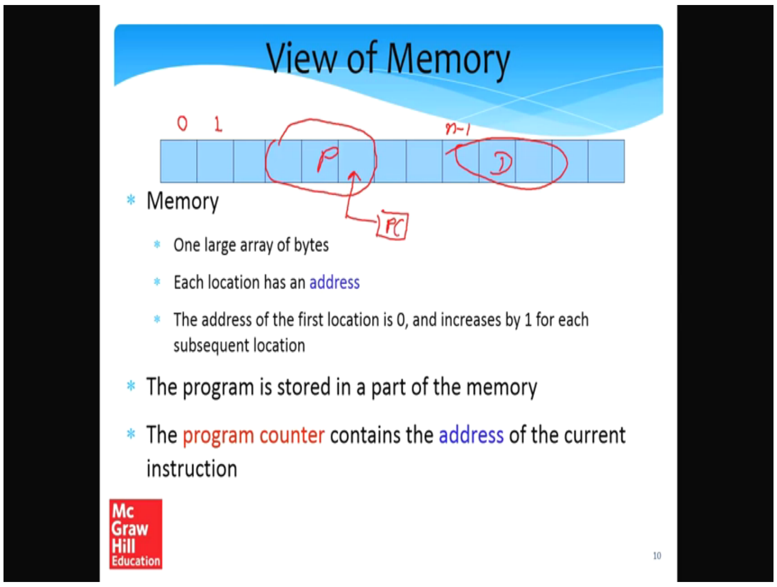
- Registers → named storage locations
  - in ARM : r0, r1, … r15
  - in x86 : eax, ebx, ecx, edx, esi, edi
- Machine specific registers (MSR)
  - Examples : Control the machine such as the speed of fans, power control settings
  - Read the on-chip temperature.
- Registers with special functions :
  - stack pointer
  - program counter
  - return address

9

So, the registers are name storage locations in ARM assembly they are called r0 r1 till r15. In x86 assembly, the Intel family Intel and A M D family or processors use x86 assembly to registers are typically referred to as eax, ebx, ecx, edx, esi and edi. So, these are general purpose registers. Additionally there are machines specific registers, which allow you very fine grained, and you know internal access to changing things such as the speed of fans, the power, and temperature controls settings.

So, these are called machine specific registers which are you know, specific to the processor not necessarily specific to the processor family. Then there are reregisters with some special functions. So, one of them is a stack pointer that we shall discuss later. Then we have the program counter we all know what the program counter stands for, the chapter one. It essentially points to the address of the currently executing instruction, and then there is also a written address register that we will come to later.

(Refer Slide Time: 13:24)


View of Memory

- Memory
  - One large array of bytes
  - Each location has an address
  - The address of the first location is 0, and increases by 1 for each subsequent location
- The program is stored in a part of the memory
- The program counter contains the address of the current instruction

Mc Graw Hill Education

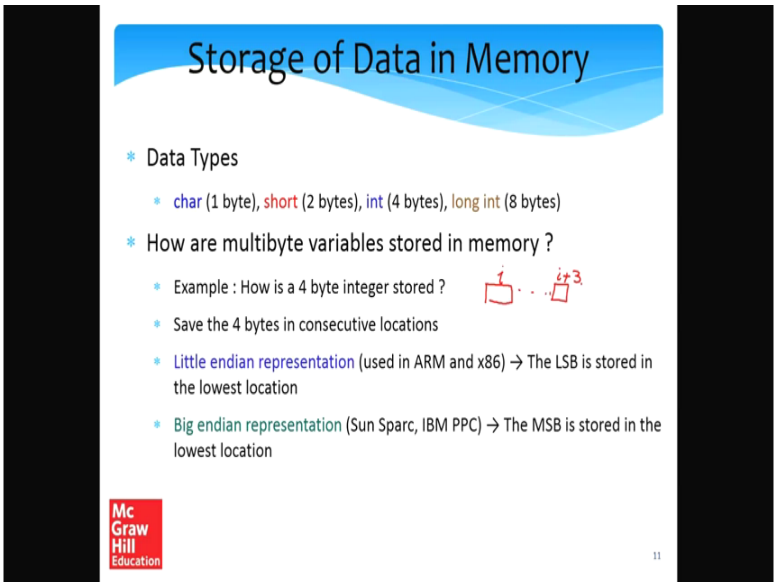
10

So, now let us take a look at memory. So, memory all assembly language programs are all programs so that are matter look at memory as one large array of bytes, one huge very large array of bytes. Each byte or each location has an address; so the address of the first memory location, the first byte. So, we start counting from 0 in computer science. So, the address of the first byte is 0, the address of the second byte is 1.

Similarly the address of the nth byte is n minus 1. So, the program is stored in a certain part of memory. So, we can assume that the program is stored over here, and the data is stored in some other part of memory. So, we can assume that it is stored somewhere over here. Say every binary assumes that the entire memory belongs to itself and. So, we are up till now not discussing what happens if you are running multiple programs at the same time. We will come to that slightly later. At the moment we are only talking about how an assembly program or binary program, would actually view memory. And the program counter contains the address, which is essentially you know the address of the location, of the current instruction.

So, the current instruction starts from here. So, this address is contained inside the program counter.

(Refer Slide Time: 15:00)



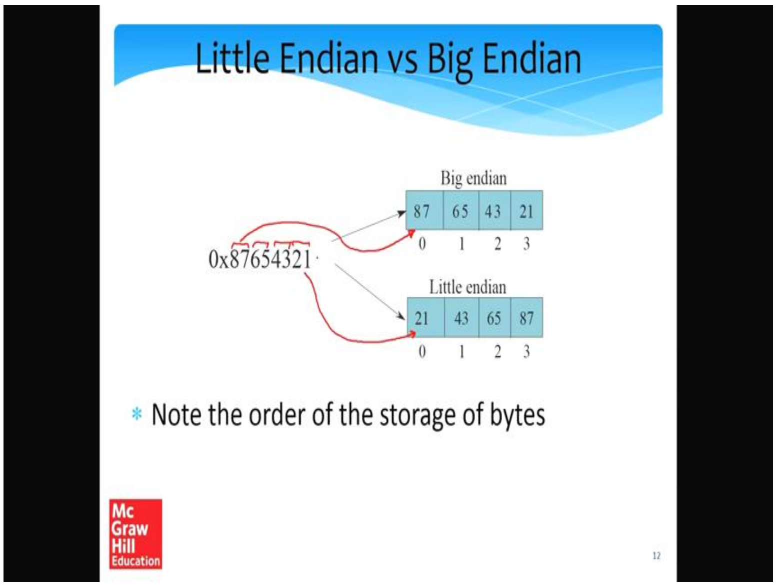
Storage of Data in Memory

- Data Types
  - char (1 byte), short (2 bytes), int (4 bytes), long int (8 bytes)
- How are multibyte variables stored in memory ?
  - Example : How is a 4 byte integer stored ?
  - Save the 4 bytes in consecutive locations
  - Little endian representation (used in ARM and x86) → The LSB is stored in the lowest location
  - Big endian representation (Sun Sparc, IBM PPC) → The MSB is stored in the lowest location

So, now let us take a look at how data is stored in memory, before we actually discuss more about assembly languages. So, there are four mainly in a there is a different kind of data, but the typical data types that we use in C r of this type. We have a char or a character which is 1 byte. We have a short integer which is 2 bytes, a regular integer which is typically 4 bytes, and a long integer typically 8 bytes. So, both C as well as java follows this convention. So, the question that we need to answer is that a memory how exactly is data stored.

So, how is a 4 byte integer stored? So, what we do is that we save the 4 bytes in consecutive locations in memory starting, select the first location is i, and the last location is I plus 3. So, there are two ways in which we can store data. So, one is called a little Indian representation. This is used in the ARM and x86 family of processors, here the least significant byte is stored in the lowest location. So, we shall you know see a visual depiction of this on the next slide. Then we have the big Indian representation used by Sun Sparc and IBM Power PC, here the most significant byte is stored in the lowest location.

(Refer Slide Time: 16:34)



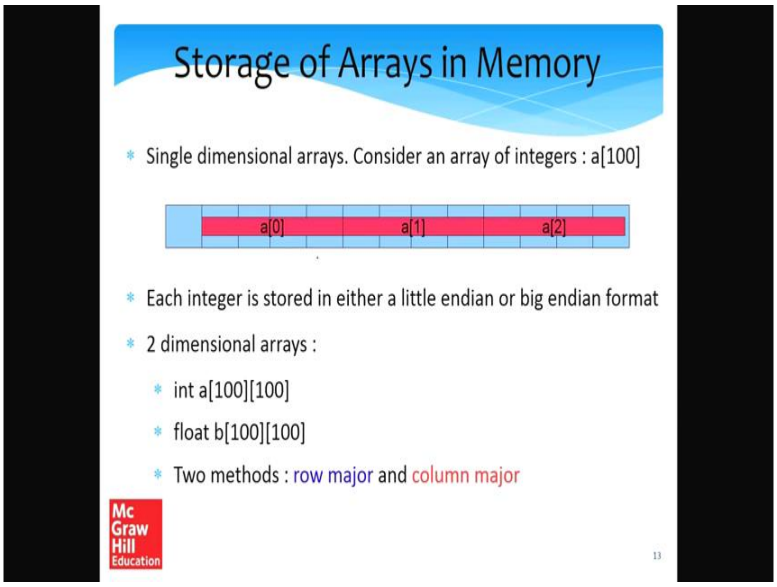
Little Endian vs Big Endian

- Note the order of the storage of bytes

So, let us take a look at examples of this. So, let us consider an integer of this form that actually contains 4 bytes is in hex notations. So, 1 digit is 4 bits. So, as a result 2 digits is 1 byte. So, this is the first byte, second byte, third and fourth. Say in the big Indian notation the way that we actually store, it is that the most significant byte the m s p is actually stored in the lowest or the smallest memory location. So, first we store 8 7, then we store 6 5 then 4 3 and then 2 1, in a little Indian location. We store the least significant byte first. So, we store 2 1 over here, then 4 3, then 6 5 and then 8 7.

So, what is different is, the order of the storage of bytes, and it is there is a matter of convention one in a, we cannot argue that little endian is better than big endian or vice versa it is just a matter of convention of how data is stored. So, ARM and x86 designers have gone for little Endean, whereas IBM and Son designers are gone for the big endian scheme. So, different conventions are there in the hardware, as well as the assembly language right at, the assembler and compiler writers, need to be aware of the particular mechanism a scheme; that is being used.

(Refer Slide Time: 18:09)



Storage of Arrays in Memory

- Single dimensional arrays. Consider an array of integers : a[100]

a[0]
a[1]
a[2]

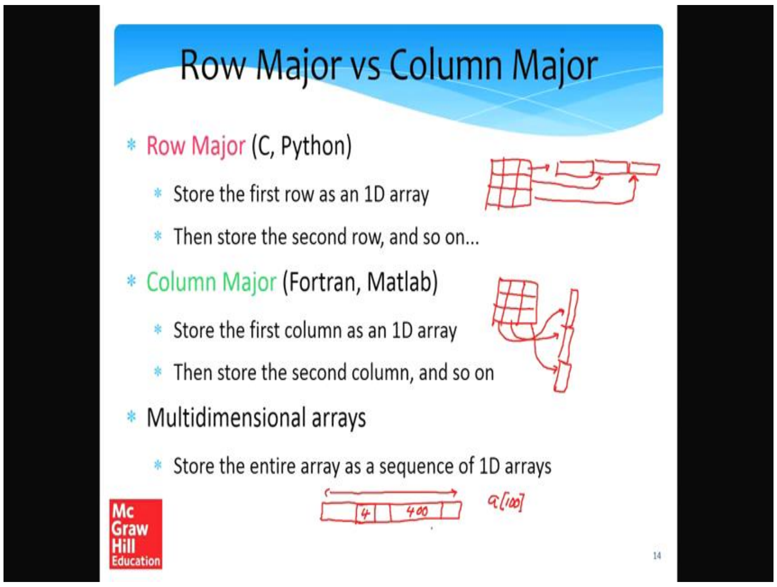
- Each integer is stored in either a little endian or big endian format
- 2 dimensional arrays :
  - int a[100][100]
  - float b[100][100]
  - Two methods : row major and column major

Mc Graw Hill Education
13

So, now, that we know of how individual integers are stored. We should also take a look at how arrays are stored. So, in array of integers is basically a set of n integers. So, let us consider an array of integers a 100. So, where it is stored is actually very simple. We store the first, we store the first element of the array a 0, and then in a consecutive location we store a 1 and a 2 and a 3 and so on. Say 1 integer is 4 bytes the entire array will take 400 bytes, and we save the integers in consecutive memory locations right. And each integer is stored in either a little endian or big endian format, depending upon the convention followed by the processer.

Now, there are an interesting question arises, that how do we store a two dimensional array. You know an array that has both rows and columns. So, how do we store it? So, there are two methods row major and column major.

(Refer Slide Time: 19:22)



Row Major vs Column Major

- Row Major (C, Python)
  - Store the first row as an 1D array
  - Then store the second row, and so on...
- Column Major (Fortran, Matlab)
  - Store the first column as an 1D array
  - Then store the second column, and so on
- Multidimensional arrays
  - Store the entire array as a sequence of 1D arrays

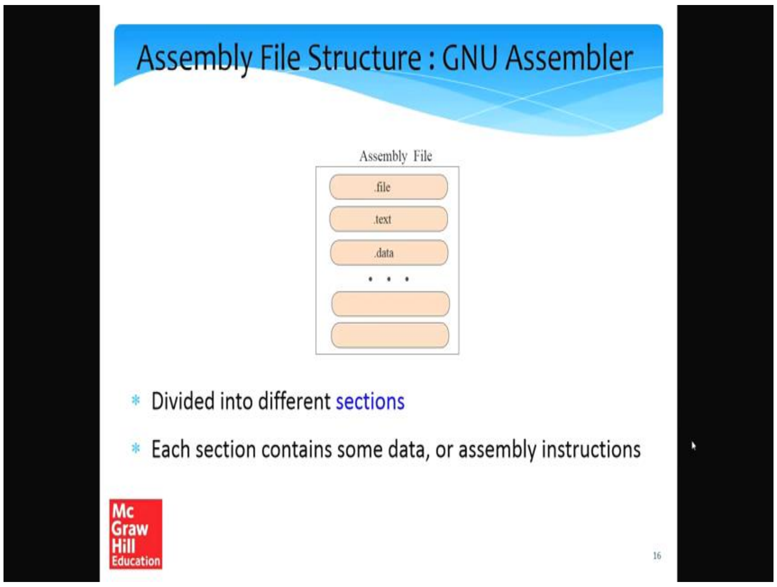
4
400
a[100]
14

So, row major scheme is used in high level languages; such as C and python. So, here what we do is, that we store the first row as a 1 D array, and then we store the second row and so on. So, let me just show you with an example. So, let us assume that this is the 2 d array with rows and columns. So, a 2 D array is a matrix. So, what we do in a row major scheme is that we first store the first row. Then we store the integers belonging to the second row. Then we store the data belonging to the third row. The other scheme called column major, which is followed by programming languages such as FORTRAN and MATLAB. They actually store the first column, then the second column and so on.

So, there are tradeoffs between row major and column major, but we are not in a position to appreciate this right now. We will discuss more of it in chapter 10. So, what exactly is column major is. In column major the first column is stored, then the second column is stored, and then the third column is stored, in memory. So, both of these approaches are, sort of opposites of each other. In one case we traverse the ray row wise; in another case we traverse the array column wise.

So, there are tradeoffs in this. So, this depends upon the way that we write programs. So, we will discuss more about this in chapter 10, but what is important to know. So, the main take home point here is, that all programs including assembly languages and processors, fundamentally assume that, the memory is one large array of bytes. To save an integers, which is 4 byte integer, we take 4 consecutive locations in memory and save the integer, in either big endian or little endian format. To save an array of integers we can save it in any form that we want; row major and column major. So, let us assume we want to save the array a 100. If 1 integer is 4 bytes we need to take up a space in memory which is 400 bytes, and save the 100 integers, and each integer can be saved in either the big endian or the little endian format.

(Refer Slide Time: 22:16)

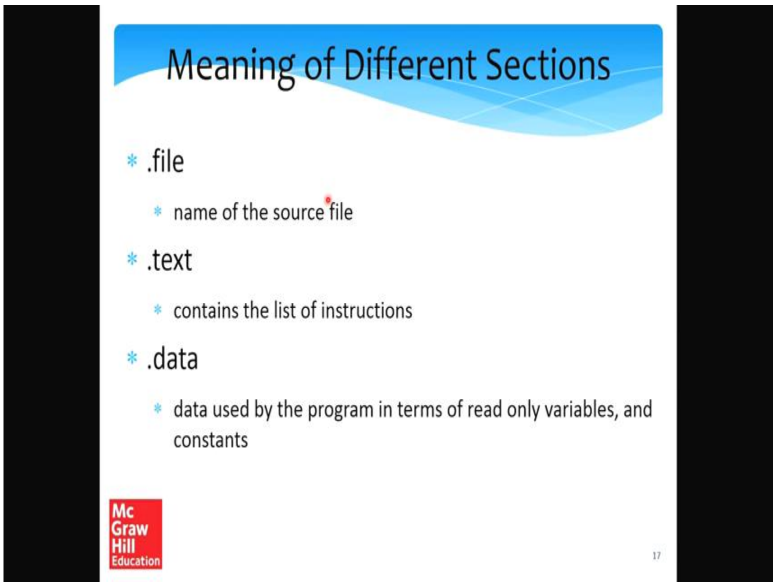

Assembly File Structure : GNU Assembler

- Divided into different sections
- Each section contains some data, or assembly instructions

Now, let us take a look at the syntax of assembly language. So, syntax basically refers to the way that assembly language is written. So, typical assembly file looks as shown over here. So, there are many kinds of assembly languages for many kinds of architectures, many kinds of ISAs. So, clearly all of them have different format, sometimes very different formats. We will discuss the assembly structure of the GNU assembler, which is very common in Linux and macros. Even other assembly files in diverse operating systems; such as windows, have a roughly similar structure, even though there are differences, but we only want to discuss the generic parts in this lecture.

So, if I consider a certain assembly file, the first part of the assembly file contains Meta information; like the name of the original program that you know, that this assembly file corresponds to. The dot text section corresponds to all the instructions, and the dot data section corresponds to all the data that is accessed by the program all the constants, particularly that are accessed by the program. So, the important point to note over here is that an assembly file is divided into different sections, and each such section is demarcated with identifiers. In the GNU assembly case these identifiers are dot file dot text and dot data.

(Refer Slide Time: 23:54)



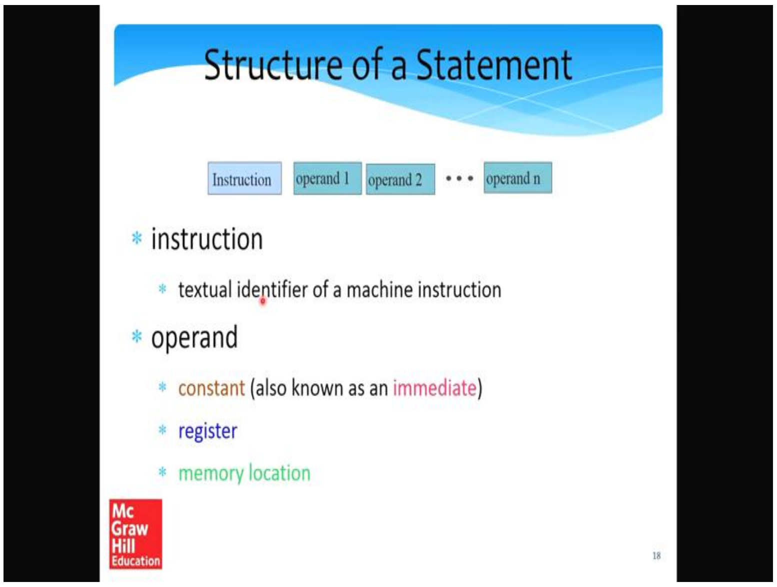
Meaning of Different Sections

- .file
  - name of the source file
- .text
  - contains the list of instructions
- .data
  - data used by the program in terms of read only variables, and constants

17

As mentioned in the previous slide dot file refers to the name of the source file, dot text section, the text section contains the list of instructions, and the data section contains the values of read only variables and constants.

(Refer Slide Time: 24:15)



Structure of a Statement

Instruction | operand 1 | operand 2 • • • operand n

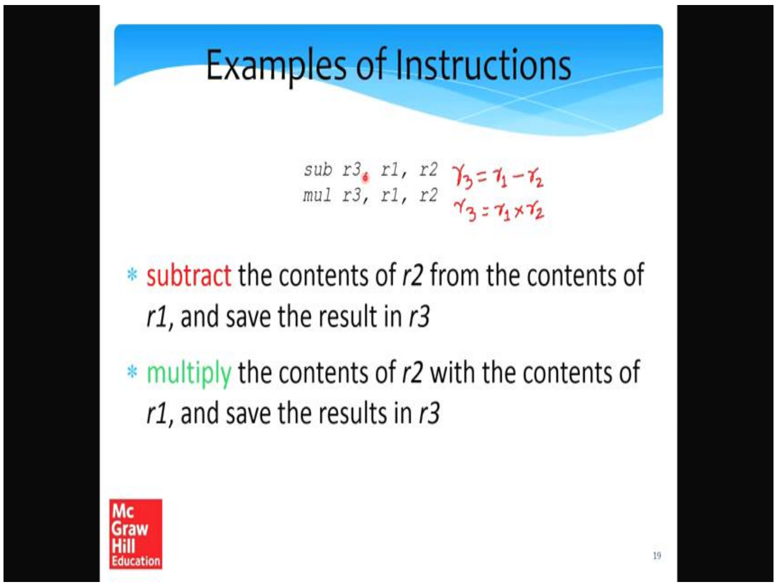
- instruction
  - textual identifier of a machine instruction
- operand
  - constant (also known as an immediate)
  - register
  - memory location

18

So, the structure of an assembly statement is typically like this, that we have an instruction, and then we have a list of operands, which are the arguments of the instruction. Each instruction as described before corresponds to typically one machine instruction, and the operands can be of different types; for example, they can be a constant. So, a constant is also known as an immediate a constant, examples of a constant are 3 4 5 in a numeric constants, character constants, string constants. So, in assembly language parlance, they are typically referred to as immediate. The operands can also be registers, and the operands can also be memory locations.

So, we will look at more of how to specify different kinds of operands.

(Refer Slide Time: 25:12)



Examples of Instructions

```
sub r3, r1, r2
mul r3, r1, r2
```

$r_3 = r_1 - r_2$

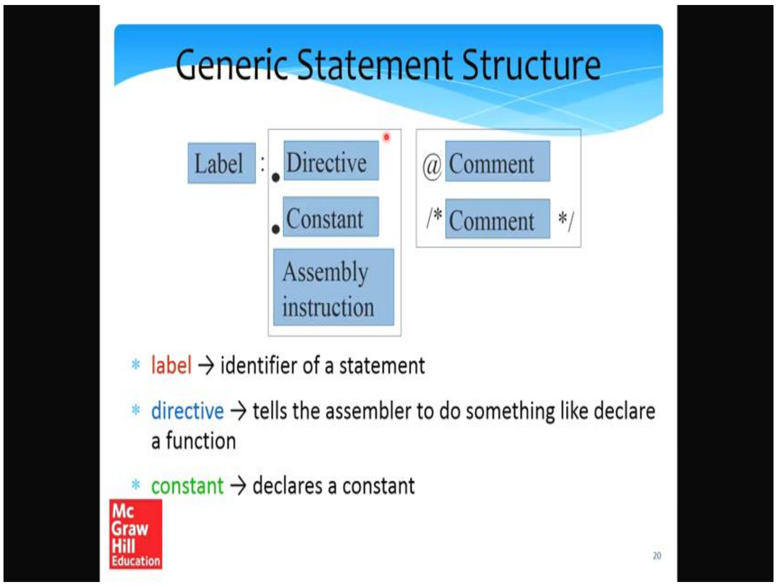
$r_3 = r_1 \times r_2$

* subtract the contents of *r2* from the contents of *r1*, and save the result in *r3*
* multiply the contents of *r2* with the contents of *r1*, and save the results in *r3*

19

So, let us take a look at examples of instructions. So, the first one is a sub instruction a subtract instruction. What is achieves is, that it is subtracts the contents of r2 from r1 and says the result in r3. So, if you want to write we can write this in this form. So, this is similar to. Similarly the next assembly instruction is similar to r3 is equal to r1 multiply by r2. Say in this sense, this is like a high level programming language; l bit the syntax is slightly different. So, the destination always comes first. So, the first operand is typically the destination, and the next two operands are the source operands. So, source operands in this case as specified using registers with, that need not be the case. So, we will take a look at that in the next few slides.

(Refer Slide Time: 26:42)



Generic Statement Structure

Label :
Directive
Constant
Assembly instruction
@ Comment
/* Comment */

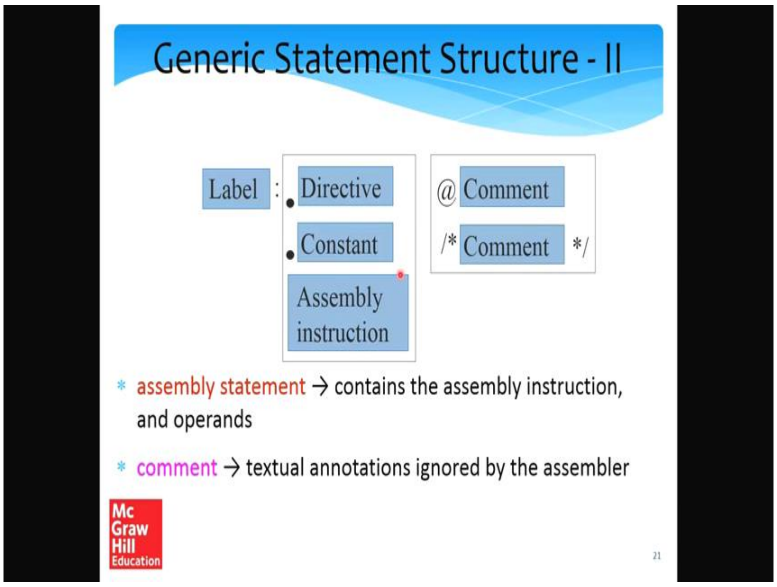
- label → identifier of a statement
- directive → tells the assembler to do something like declare a function
- constant → declares a constant

So, generic, but this is one kind of the previous these two assembly statements are only one kind of assembly statements, that can be many other kinds as well. Let us take a look at some. So, we can have a generic assembly instruction, followed by comments. So, there are two ways of commenting, that we will look at in this chapter. The first comment starts an, say anything after an at is a comment, or we can have generic C style comments with a slash star, and then a trailing star slash. So, the comments can always come at the end of a statement or you can have an entire line with the comment. The assembly statement itself can be an assembly instruction, or we can define a constant, or we can have an assembler directive, which is essentially an instruction to the assembler to do something special.

So, we will discuss that later, and the entire assembly statement inclusive of the comment can be preceded by a label. So, label can be thought of as. See it is similar to a label that is used in a go to statement in c. It uniquely points to the assembly statement that is after it. So, we can have a label and then a colon, and then specified assembly instruction. Later on if you want to come back to this label, we just need to have something to the effect go to label, and the control will jump back to this particular label.

So, label is an optional argument, it is not required all the time, but whenever you want to jump to this particular statement from anywhere else in the program, we can use the label to indicate this particular line. So, a label is essentially the identifier of a statement, and as I told you before a directive, tells the assembler to do something, in a maybe like say declare a function or declare, the fact that a new section is starting in the file. So, there are many kinds of directors, we will discuss that gradually.

(Refer Slide Time: 29:02)



Generic Statement Structure - II

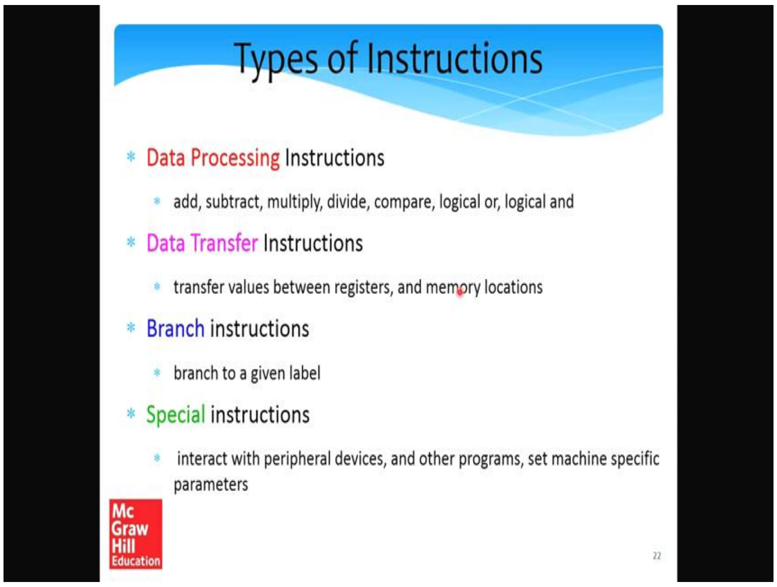
Label : .Directive .Constant Assembly instruction @ Comment /* Comment */

- assembly statement → contains the assembly instruction, and operands
- comment → textual annotations ignored by the assembler

21

So, in the generic statement structure, we have already looked at the assembly instruction part, and the comment part. So, we can have many kinds of instructions.

(Refer Slide Time: 29:10)



Types of Instructions

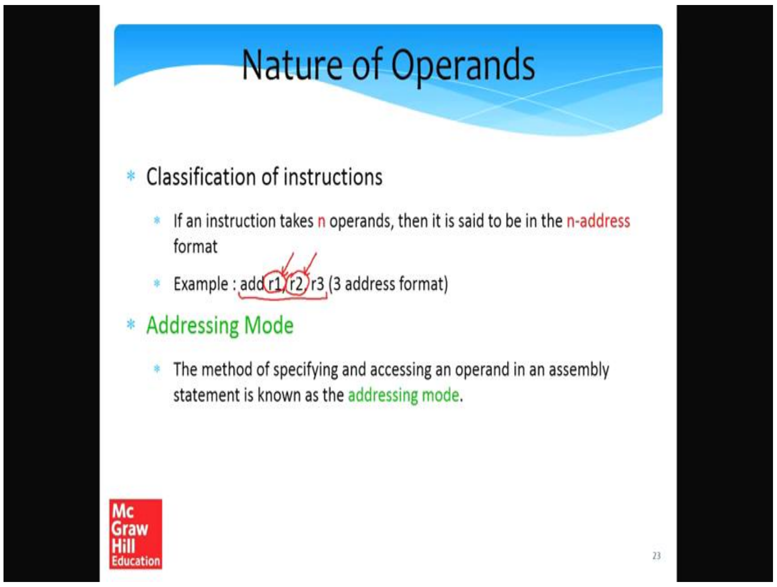
- Data Processing Instructions
  - add, subtract, multiply, divide, compare, logical or, logical and
- Data Transfer Instructions
  - transfer values between registers, and memory locations
- Branch instructions
  - branch to a given label
- Special instructions
  - interact with peripheral devices, and other programs, set machine specific parameters

22

So, primarily four kinds: we can have data processing instructions. So, these instructions are typically arithmetic and logical instructions. They can be used to add, subtract, multiply, divide; numbers perform arithmetic operations on them. Also they can be used to compare two numbers and store the result somewhere else. We will we will discuss where that somewhere else is. Also do Boolean operations between the value stored in registers are; namely a logical or a logical and a logical or not. So these are all Boolean operations that can be done, by this family of instructions called data processing instructions. Similarly, we can have data transfer instructions. These instructions transfer values between registers and memory locations.

So, these are also very useful to bring in data from memory to registers, and also to move data between registers, as well as move data from resistance to memory locations back. Then we of course, need branch instructions to implement for loops, while loops, if statements. So, branch instruction is typically like this that we branch to a given label. For the label is what we looked at in the previous slide, it uniquely identifies an instruction and assembly statement, along with data processing, data transfer and branch instructions, we can have special instructions, to interact with peripheral devises, and other programs offset machines, specific parameters, or essentially access special parts of the computer system, that are otherwise not accessible to high level program. So, these are special instructions that are not discussed in nature of operands.

(Refer Slide Time: 31:15)


Nature of Operands

- Classification of instructions
  - If an instruction takes n operands, then it is said to be in the n-address format
  - Example : add r1, r2, r3 (3 address format)
- Addressing Mode
  - The method of specifying and accessing an operand in an assembly statement is known as the addressing mode.

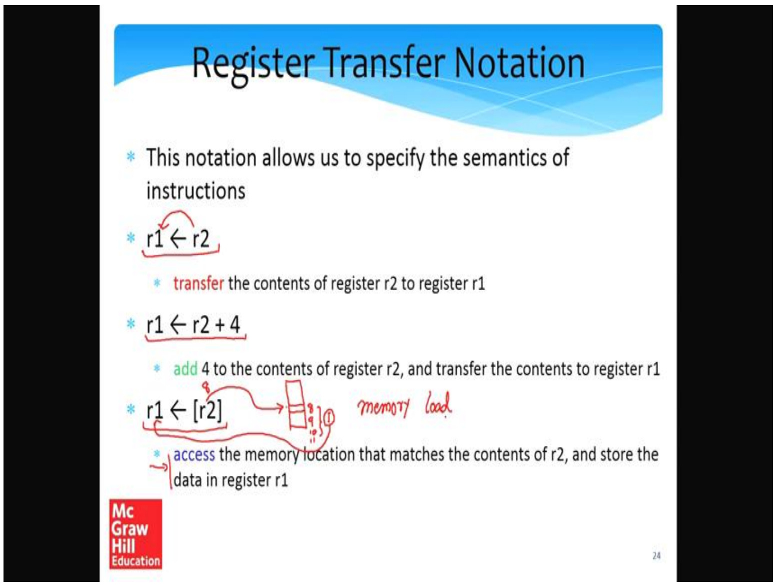
Mc Graw Hill Education

23

So, let us begin by classifying instructions and defining some basic terms. So, if an instruction takes n operands as a input, then we say that it is in the n address format. For example, this instruction over here, the add instruction, takes three operands as input. So, we say that it is in the three address format. So, similarly we can have instructions, in the 1 address, 2 address. We can even have 4 address format instructions. So, an address format basically refers to the number of operands that an instruction takes.

Let us then look at the addressing mode. So, the method of specifying in accessing an operand in an assembly statement is known as the addressing mode. So, we will see there are many kinds of addressing modes. For example, in this assembly statement over here, each operand is a register based operand. So, the addressing mode is a register based addressing mode.

(Refer Slide Time: 32:25)



Register Transfer Notation

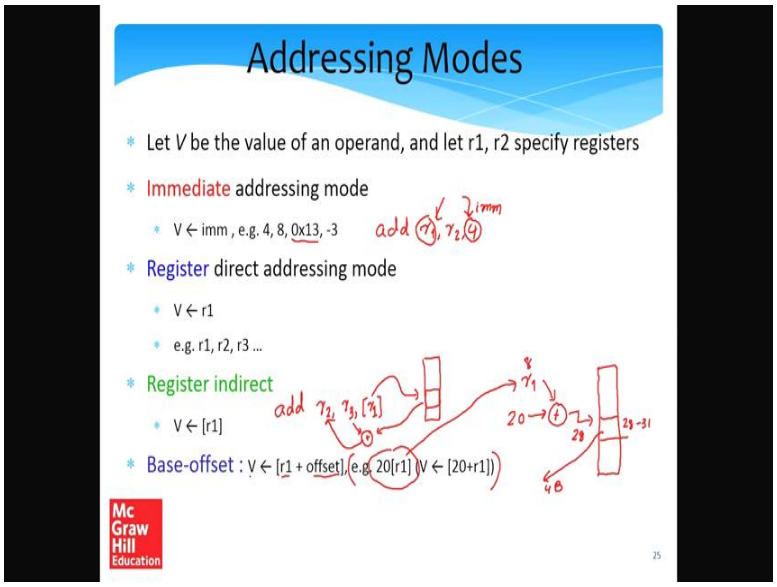
- This notation allows us to specify the semantics of instructions
- r1 ← r2
  - transfer the contents of register r2 to register r1
- r1 ← r2 + 4
  - add 4 to the contents of register r2, and transfer the contents to register r1
- r1 ← [r2]
  - access the memory location that matches the contents of r2, and store the data in register r1

So, now let us introduce the register transfer notation, we will actually required it, to specify the semantics, the way that our assembly instructions work. So, the in the register transfer notation, let us take a look at the most basic notation over here r1, the contents of r2 being assign to r1. So, this is basically. So, this arrow indicates that the direction of data transfer is from r2 to r1.

So, this tells the transfer the contents of register r2 to register r1. So, we can have a slightly more complicated representation as well. So, this representation says, that add 4 to the contents of register r2. Essentially at an immediate to the contents of register r2, and then transfer the contents to register r1. The last expression is slightly different. So, what this tells, is that we first access the contents of registrar r2 and then. So, assume that the contents of register r2 are let us say 8. We access the integers, the 4 byte integer, to starting address it is 8, and then we subsequently read, subsequently we read 4 bytes, starting at location 8.

So, essentially we read the bytes at location 8 9 10 and 11. So, these 4 bytes are make one integer. This integer is transferred to register r1. So, in this case we are actually accessing a location in memory, the address of the location is specified and registered r2. Say if we read the definition; that is mentioned over here it tells, we access the memory location that matches the contents of r2. So, in this case in the example the contents of r2 are 8. So, we start from memory location 8, and read 4 bytes, starting at 8 9 10 and 11. So, 4 bytes make 1 integer, and then the integer is stored. So, in this case the integer is the data; that is coming from memory, this is store in register r1. So, this is a memory load operation. So, this is this operation is called a memory load, because we are loading data from memory.

(Refer Slide Time: 35:16)



Addressing Modes

- Let *V* be the value of an operand, and let r1, r2 specify registers
- Immediate addressing mode
  - V ← imm , e.g. 4, 8, 0x13, -3
- Register direct addressing mode
  - V ← r1
  - e.g. r1, r2, r3 …
- Register indirect
  - V ← [r1]
- Base-offset : V ← [r1 + offset], (e.g. 20[r1] (V ← [20+r1]))

add r1, r2, 4
imm
add r2, r3, [r1]
8
r1
20
+
28
28-31
4B
25

So, let us take a look at some of the very basic addressing modes. So, let v be the value of an operand, and let r1 and r2 specify a registers. So, the convention is always; that r1, in any variable starting with r always specify the register. So, let us take a look at the immediate addressing mode. In immediate addressing mode, the constant itself is the value. For example, 4 8 or a hexadecimal constant over here which is 0 x 1 3, it is 19 and decimal or even a negative number, it can be anything, it can be any kind of a constant.

So, when a constant is directly used as is; for example, if I have an instruction of this form add r1 r2 and 4, this particular operand, is using the immediate addressing mode, because the constant is specified as is. Similarly we can use the register direct addressing mode where the value of stored in the register is, what is being used. So, in this case r1 is using the register direct addressing mode.

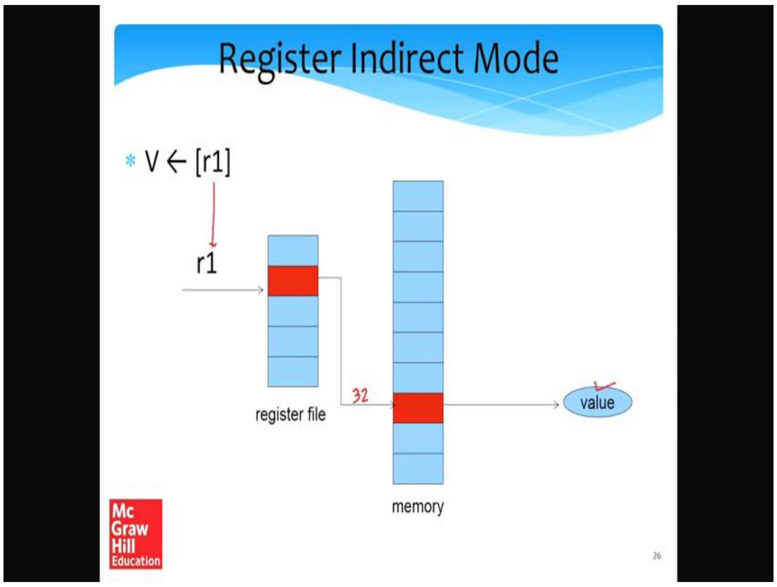
Similarly we can use the register indirect addressing mode, something that we had seen in the previous slide, in this particular point. For example we can. In some assembly languages we can write a function of this type, add r through r2 r3 and r1. So, in this case r1 actually contains a memory address. This will be used to access memory. We will read 4 bytes from there the size of an integer, and this will be added to the contents of r3, and finally, we saved an r2. So, this is called the register indirect addressing mode, because what the register contains is actually a memory address.

The address is sent to the memory; we get contents from the memory, and then use it in an instruction. Similarly we can use; we can slightly extend this, to create the base offset addressing mode. So, in this case, the actual value; that is being used by the program is actually r1. So, in this case let us consider an example, and then explain the theory part of it.

So, let us con consider this operand over here, which is using the base offset addressing mode. So, what is basically, such is that we take the contents of r1. So, let us, let this be the contents of r1, and then we add it with the constant here 20. So, you will assume that the contents of r1 were 8. So, you add 8 and 20 and we get 28. We use 28 to access the 28 locations or memory, and then we read 4 bytes from the 28 to the 31st location, and these 4 bytes are then used by the program. So, this is a base offset addressing mode, because in this case r1 contains the base address, and the offset is 20. So, we add both of them, we get a final address is called effective address, which is used to access memory, and then we get the contents of an integer from memory. We read an integer from memory and this is used by the program.

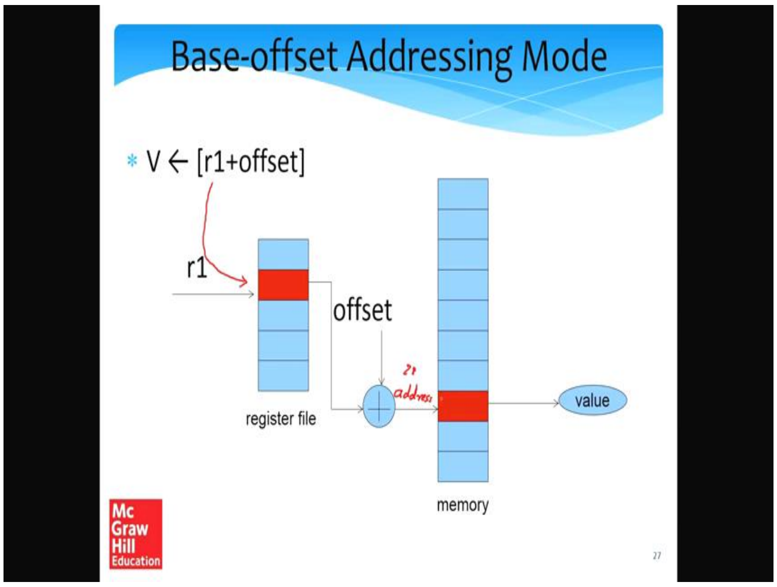
So, the same thing is shown over here, that we take the value of r1, we added to the offset. The squared brackets are indicated to the address, is sent to memory, and we read the contents from memory, and this is the final value that we use.

(Refer Slide Time: 39:40)



So, let us show that the register indirect mode once again using a set of better pictures. So, as mentioned over here, we take the value of r1, and whatever is the value of r1, we first read it from the register file, and then we use the value to access memory. So, let us assume that. For example, r1 contained the 32. So, we access a 30 second location in memory and read 4 bytes. So, for simplicity we are not shown four entries, and we finally read the value out.

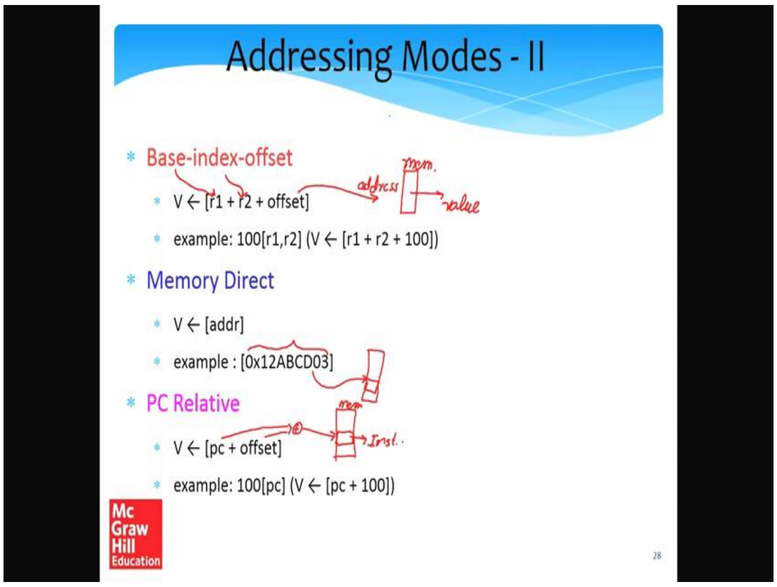
(Refer Slide Time: 40:18)



Now, let us discuss the base offset addressing mode. So, this mode will be very useful for implementing arrays. So, the main idea over here is to actually use a register call the base register. In this case, in this example it is r1. So, r1 is used to access the register file. From the register file we read the contents of r1, and we added to the offset. So, we get an address as the output, and this is called the effective address, but let me just call it as the address for the time being.

The address is used to access memory, and finally, we read 4 bytes from the memory. So, the 4 bytes are part of a single cell. This is drawn to enhance readability. So, the 4 bytes are the value of this particular operand. So, what exactly is a base offset addressing mode again. We have a registered and an offset. We read the contents of the register, added to the offset, this gives us an address, and we use this address to access memory. So, for example, let see the address is 28. So, we access the 28 location in memory, and read 4 bytes, and these 4 bytes are the final value of the instruction.

(Refer Slide Time: 41:43)



Addressing Modes - II

- Base-index-offset
  - V ← [r1 + r2 + offset]
  - example: 100[r1,r2] (V ← [r1 + r2 + 100])
- Memory Direct
  - V ← [addr]
  - example : [0x12ABCD03]
- PC Relative
  - V ← [pc + offset]
  - example: 100[pc] (V ← [pc + 100])

address
mem.
value
mem
Inst.
Mc Graw Hill Education
28

So, let us discuss some more addressing modes which are not very frequently used, but nevertheless they are common enough, that we should you know discuss them, and especially when we take a look at the x86 instruction set, you would find these addressing modes to be extremely useful. So, one of them is the base index offset addressing mode. So, in this case we instead of one register you have two registers. So, we have a base register, which in this case is r1, we have an index register which is r2 in this example, and we have an offset which is an immediate. So, we add the contents of the two registers, and the offset we get an address. This address is used to access memory, and we read the value out.

So, this is the value of the operand. So, one example would be 100 r1 r2. Here r1 is the base register, r2 is the index, and 100 is the offset. So, note that this sign of braces, the square braces, are used to indicate that the contains within the braces refer to an address, and once we add the braces, we are supposed to access memory with the address, and treat the locations in memory, as containing a value, as containing the value of the operand. So, we can again have a memory direct addressing mode, which is not there in most ISAs, but it is there in x86.

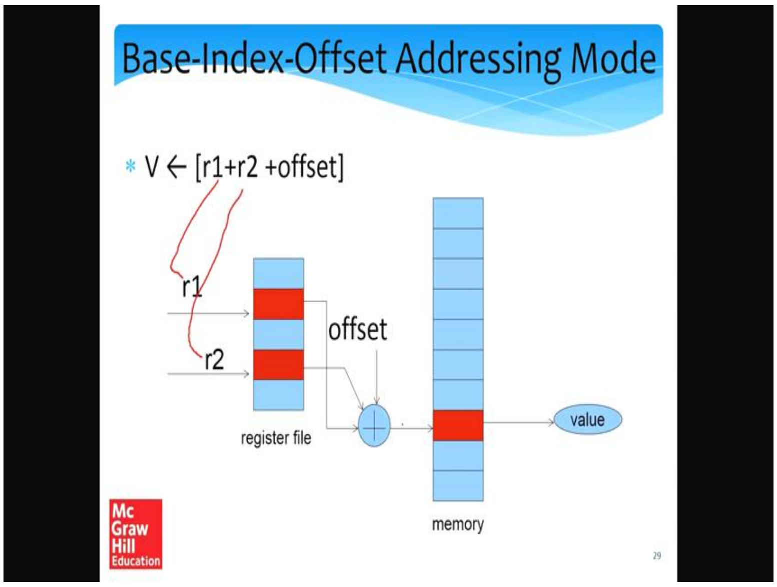
In this case if we know the address of the memory location you can directly specify it. We do not need to add it in a register; instead we can directly specify it. So, in this case if this is the address of the memory, we directly specify it, and enclose it and curly. I am sorry in square brackets sorry in braces, and this indicates that we access memory with this address, and the subsequent 4 bytes contain the data of the operand.

Lastly let us take a look at a certain variant of a register indirect addressing mode, which is actually very useful, it is called the PC relative addressing mode, where instead of using a general purpose data register, we use the program counter, we treat the program counter as a register. So, in this case, we take the contents of the program counter, add an offset to it to get an address, and we use the sum of both, to actually access instruction memory. So, let this be the memory in specific the instruction memory. So, then the next 4 bytes are actually the contents of the operand.

So, next 4 bytes are typically an instruction, in a 32 bit system, which we assume to be the default in at least in this book. Now this is. So, these 4 bytes from the PC will typically point to an instruction. So, we will find this addressing mode very useful. This addressing mode will be found to be useful, especially while implementing the concept of, a statement for loops and while loops in assembly, that we will be able to change the PC the program counter, essentially we will be able to change it is value, and jump to different points within the program, which is exactly what a next statement or a for loop and while loop do.

So, let us keep it, let us keep this discussion at this point, and when we discuss when we go into the netigrities or assembly language, we will come back and revisit this part.

(Refer Slide Time: 46:05)



Base-Index-Offset Addressing Mode

* V ← [r1+r2 +offset]

So, let us take a look at the base index offset addressing mode once again with a better diagram. In this case we have two registers we have r1 and we have r2. So, both the registers are used to access the register file. We add their contents plus the offset, then we access memory and we use the value.