Indian Institute of Technology, Guwahati

CS 348 Implementation of Programming Languages Lab

Quiz 2

Duration: 55 minutes, Date: 24-02-2023

Answer all questions

1. What is the problem of forward reference in Assembler? How is it handled in one pass assembler? Illustrate with an example.

2. Mention some of the Machine-Dependent and Machine-Independent Assembler Features. Why is a particular feature called Machine-Dependent or Machine Independent? Explain.

3. Mention the different addressing modes with examples. Which addressing modes are suitable for program relocation at run time and why?

4. Generate the Symbol Table, Literal Table, and Pool Table, and then generate the Intermediate Code for the following generic Assembly Language Program. The necessary tables are given below.

```
START 100
MOVER AREG, A
LOOP: PRINT B
ADD BREG, ='9'
SUB BREG, D
COMP CREG, ='23'
LTORG

A DS 3
LABEL: EQU LOOP
ORIGIN 500
L1: MULT CREG, ='7'
SUB BREG, ='93'
LTORG

B DC 10
MOVEM CREG, ='7'
PRINT ='7'
D DC 8
END
```

(MOT)	
OP-Code	**Mnemonic**
01	MOVER
02	MOVEM
03	ADD
04	SUB
05	MULT
06	DIV
07	BC
08	COMP
09	PRINT
10	READ

(POT)	
OP-Code	**Mnemonic**
01	START
02	END
03	EQU
04	ORIGIN
05	LTORG

(DL)	
OP-Code	**Mnemonic**
01	DS
02	DC

REGISTERS	
Reg No	**Name**
01	AREG
02	BREG
03	CREG
04	DREG

5. Explain subroutine linkage with the help of an example.

6. What is the difference between a relocating linker and a relocating loader?

7. Explain how an absolute loader is easier to implement as compared to other types of loaders. Draw comparisons with at least one other type of loader.

8. What do CS, SS, DS, ES, FS and GS stand for? What are their functions? Which of these are still in use.

9. Differentiate between static and dynamic linking. State at least two advantages and disadvantages of each.

10. What are the steps involved in compiling a c code with all intermediate steps. Draw a flow chart explaining the same.

11. Convert the following regular expressions into their corresponding NFA(Non-deterministic Finite Automata) followed by DFA(Deterministic Finite Automata).The Regular Expression is: $(xyz)^*(yzx)^*$

12. Define Token. Write down the syntax of the token and explain each field of the syntax with some illustration.

Best wishes

QUIZ 2

1. The forward reference problem is a common issue encountered in assemblers when a symbol is referenced before it is defined. In other words, the assembler encounters an instruction that references a label or symbol that has not yet been defined. This can occur when a label is defined later in the program or in a different section of the code.
   The forward reference problem can create challenges for the assembler because it needs to generate the correct object code, which requires knowing the address or location of the symbol being referenced. Assemblers have different techniques to handle forward references, such as using a two-pass assembler, back patching, or dummy labels.
   In One Pass forward referencing source program is translated instruction by instruction. Assembler leave address space for label when it is referenced and when assembler found the declaration of label, it uses *back patching*. Back patching is a technique used in assemblers to handle forward references. It involves creating a table of incomplete instructions. This table is used to keep track of all the instructions that have a forward reference to a symbol that has not yet been defined.
   Each entry in the table contains the address of the instruction with the forward reference and a pointer to a linked list of references to that symbol. When the definition of the symbol is encountered, the assembler updates the address of the symbol in the symbol table and then updates all the entries in the linked list with the correct address of the symbol.
   In other words, the table of incomplete instructions is used to keep track of all the instructions that have a forward reference to a symbol. When the symbol is defined, the assembler updates the address of the symbol in the symbol table and then back patches all the instructions in the linked list with the correct address. This allows the assembler to generate correct object code even when there are forward references to symbols that are not yet defined.

- **Forward reference: reference to a label that is defined later in the program.**

Loc	Label	Operator	Operand	
1000	FIRST	STL	RETADR	
1003	CLOOP	JSUB	RDREC	
...	...	...	...	...
1012		J	CLOOP	
...	...	...	...	...
1033	RETADR	RESW	1	

2. Machine Dependent:

   A. Instruction formats and addressing modes
   B. Program Relocation

   Machine Independent:

   A. Literals
   B. Symbol-Defining Statements
   C. Expressions
   D. Program Blocks
   E. Control Sections and Program Linking

   A feature is classified as machine-dependent or machine-independent based on whether it is specific to a particular hardware architecture (machine).

   Machine-dependent features are instructions, directives, or other programming constructs that are specific to a particular hardware platform. These features are closely tied to the machine's hardware and operating system and are often used to perform low-level operations that are necessary to control the computer's hardware

directly. Examples of machine-dependent features include machine-specific assembly instructions, such as those used for accessing registers or manipulating bits in memory, as well as system calls or interrupt handlers that interact directly with the operating system.
In contrast, machine-independent features are those that are not specific to a particular hardware architecture or operating system. These features are designed to be portable across different machines and are usually implemented in a way that is independent of the underlying hardware. Examples of machine-independent features include assembly directives, which provide high-level control over the assembly process itself, as well as macros, which are reusable code blocks that can be used to simplify and streamline the assembly process.

3.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri)	EA = [Ri]
	(LOC)	EA = [LOC]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri,Rj)	EA = [Ril + [Rj]
Base with index and offset	X(Ri,Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] +X
Autoincrement	(Ri)+	EA = [Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA = [Ri]

EA = effective address Value = a signed number

Fig. 1.7 Addressing modes

Several addressing modes are suitable for program relocation at run time, as they allow for the program to be loaded and executed at different memory locations. Here are some of the addressing modes that are commonly used for program relocation at run time:

1. Base-relative addressing: In this addressing mode, the address of a memory location is calculated relative to a base address. The base address can be changed at run time to relocate the program. This addressing mode is particularly useful when the program needs to be loaded into different memory locations and the code is position-independent.

2. PC-relative addressing: In this addressing mode, the address of a memory location is calculated relative to the current program counter (PC) value. This allows for relative addressing within the code segment and can be used for relocation at run time. This addressing mode is commonly used in processors with a Harvard architecture.

3. Indirect addressing: In this addressing mode, the address of a memory location is stored in a register or memory location, and the value at that address is accessed. This allows for dynamic memory allocation and relocation at run time. This addressing mode is particularly useful in languages that use dynamic memory allocation, such as C and C++.

4. Indexed addressing: In this addressing mode, the address of a memory location is calculated by adding an index or offset to a base address. This can be used for relocation by changing the base address or index at run time. This addressing mode is particularly useful when accessing arrays or structures.

These addressing modes are suitable for program relocation at run time because they allow the program to be loaded and executed at different memory locations, which is useful in embedded systems and other applications where memory is limited or the program needs to be loaded into different memory locations.

4.

Intermediate Code

(AD, 01) - (C,100)
100) (IS, 01) 01 (S, 01)
101) (IS, 09) - (S, 03)
102) (IS, 03) 02 (L, 1)
103) (IS, 04) 02 (S, 04)
104) (IS, 08) 03 (L, 02)
105) (AD, 05) - 009
106) (AD, 05) - 023
107) (DL,01) - 03
500) (IS, 05) 03 (L, 03)
501) (IS, 04) 02 (L,04)
502) (AD, 05) - 007
503) (AD, 05) - 093
504) (DL,02) - 010
505) (IS, 02) 03 (L, 05)
506) (IS, 09) - (L,05)
507) (DL, 02) - 008
508) (AD, 02) - 007

Symbol Table

Sym_no	Symbol	Address
1	A	107
2	LOOP	101
3	B	504
4	D	507
5	LABEL	101
6	L1	504

Literal Table

Lit_no	Literal	Address
1	='9'	105
2	='23'	106
3	='7'	502
4	='93'	503
5	='7'	508

5. The subroutine linkage method is a way in which computers call and return the Subroutine. The simplest way of Subroutine linkage is saving the return address in a specific location, such as a register. The control can leave the process and return to the original process using the location saved in the register.

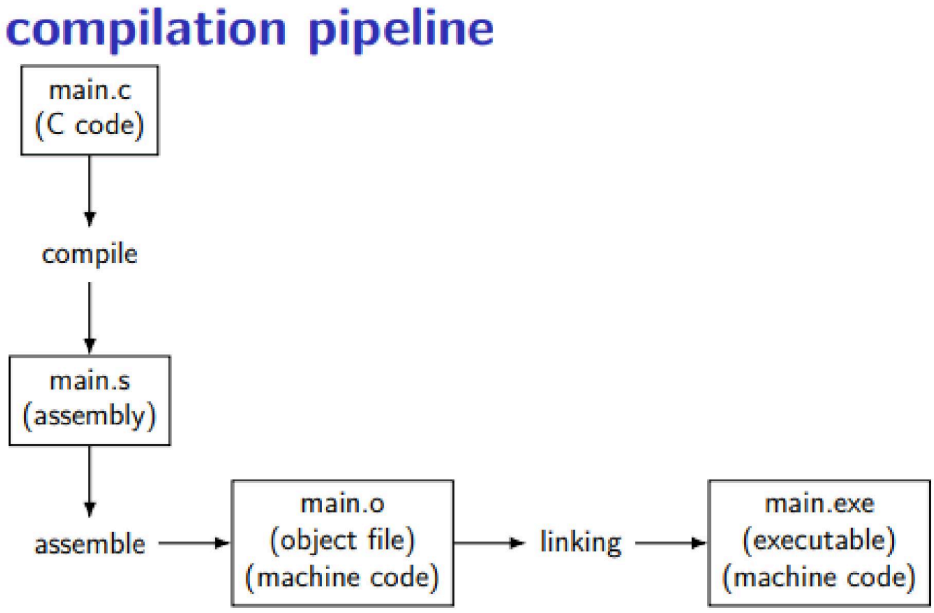
6. A relocating linker performs relocation in conjunction with symbol resolution. It searches files and libraries to replace symbolic references or names of libraries with usable addresses of the libraries or symbolic references before the program executes.
   A relocating loader is responsible for relocating the records. Relocating loaders look for available and sufficient memory spaces and the program is loaded wherever there is room for it.

7. Absolute Loaders are easier to implement as the relocation address is handled by the programmer or the assembler themselves. The loader simply places records into the memory at the specified location.
   Relocating loader is responsible for relocating the records. Relocating loaders look for available and sufficient memory spaces and the program is loaded wherever there is room for it.

8. CS: Code Segment (jump, call, etc.). SS: Stack segment (push, pop, etc.), DS: Data Segment (mov, add, etc.), ES: Additional Data Segment (string instructions), FS and GS: extra segments. FS and GS are still in use.

9. Static linking is performed to create an executable file. This executable file contains all the necessary libraries into itself. All modules are already present in the single file.
   Dynamic linking links the dependencies when the executable is run. Every dynamically linked program contains a small, statically linked function that is called when the program starts. This static function only maps the link library into memory and runs the code that the function contains. The link library determines what are all the dynamic libraries which the program requires along with the names of the variables and functions needed from those libraries by reading the information contained in sections of the library. After which it maps the libraries into the middle of virtual memory and resolves the references to the symbols contained in those libraries.
   Advantages of dynamic linking: smaller executables, easier upgrades, less memory usage.
   Disadvantages of dynamic linking: library upgrades breaking programs, possibly slower.
   Advantages of static linking: Faster, Program is more compatible with other OS versions.
   Disadvantages of static linking: upgrades require re-creation of executable, executables are large in size.

10. Since loading is now a function of the OS, I have not included it in the flow chart below. The loader will load the executable into the main memory for execution to start.

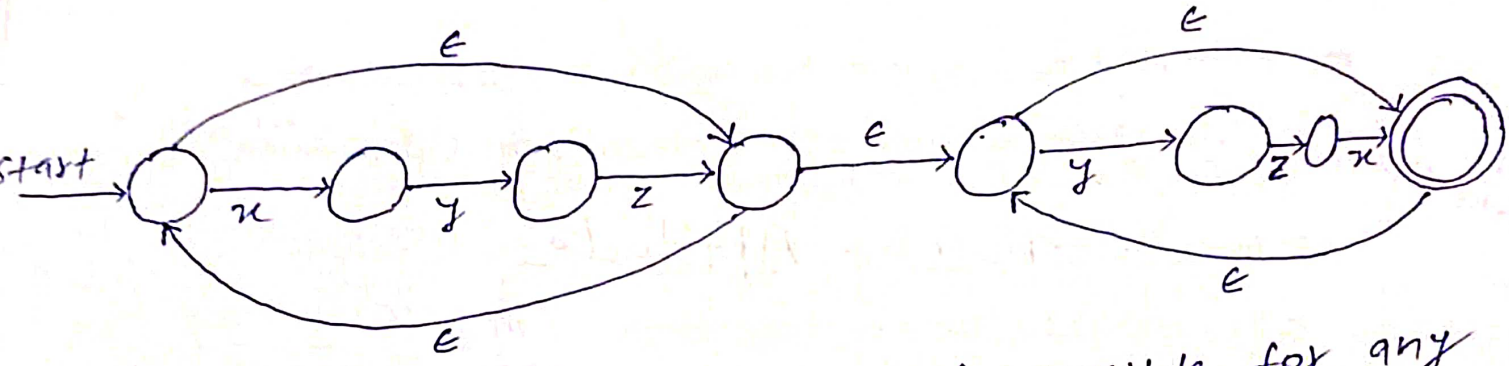


Q.11—

Ans:—

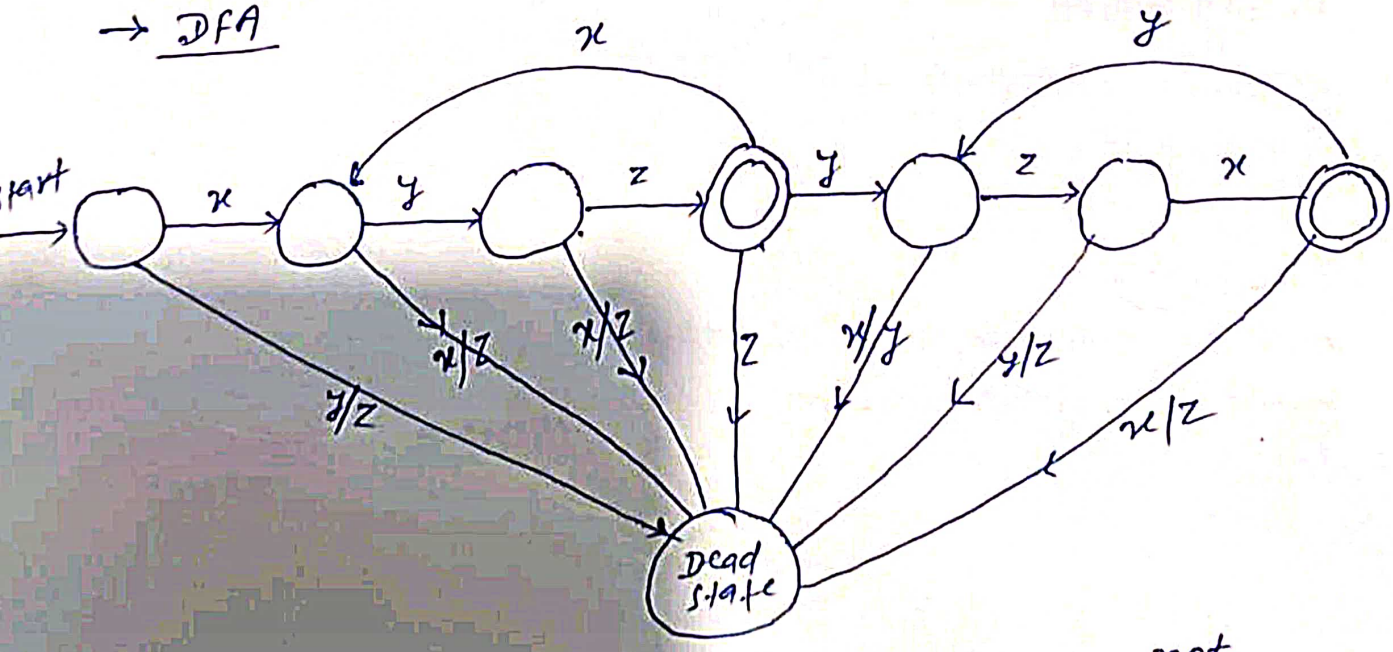
Regular expression : $(xyz)^* (yzx)^*$

→ One of the NFA for above Regular expression is :—



Note:— There are plethora of NFA's possible for any Regular expression, so please verify accordingly.

→ DFA



Note: ① Every DFA is also a NFA but not vice versa.

② for DFA, Dead state (explanation) is necessary.

Q.12.

Ans:-

Token:- A sequence of characters that are treated as single unit as it cannot be further broken down.

✓ A lexeme is an actual character sequence forming a specific instance of a token, such as id, num etc

Syntax of token

<token-name, attribute-value>

token-name → Abstract symbol, which is used during syntax analysis

eg:- id is the abstract symbol for identifier.

attribute-value → points to the entry in the symbol table.

Illustration expression

position = initial + rate * 60

✓ Token symbol representation of token 'position' is as,

<id, 1>

1	position	
2	initial	

Symbol table