Vivekanand Education Society's Institute of Technology, Chembur, Mumbai,
Department Of Computer Engineering,
Year: 2018-19 (Even Sem)
Test No.- 2

Class : D12	**Division: A / B / C**
Semester: VI	**Subject: SPCC**
Date:8/4/19	**Time: 1 hr**

Course Outcome	CO1	CO2	CO3	CO4	CO5	CO6
Percentage %	-	30%	15%	10%	-	45%

Q.1		**(Attempt any five of the following)**	**Marks (20)**	**CO**	**BL**
	a)	List different loading schemes. Explain any one in brief.	2M	CO4	1
	b)	With reference to macro processor explain MDT MNT.	2M	CO3	2
	c)	Write short note on parameterized macro.	2M	CO3	1
	d)	Explain Quadraples Triples and Indirect triples.	2M	CO6	2
	e)	Draw DAG and represent the following example (a/b)+(a/b)*(c*d).	2M	CO6	3
	f)	Explain various functions of loader.	2M	CO4	2
Q.2	a)	Discuss different issues in design of code generation.	5M	CO6	2
		OR			
	b)	Explain different types of code optimization techniques in compiler design.	5M	CO6	2
Q.3	a)	Draw flow chart for pass 1 of two pass assembler.	5M	CO2	3
		OR			
	b)	Design different data structures used in 2 pass Assembler also show the output of both passes PG3 START USING *,BASE SR 1,1 L 1,FOUR A 1,FIVE A 1,=F'3' ST 1,TEMP FOUR DC, F'4' FIVE DC ,F '5' BASE EQU 8 TEMP DS '1' F END	5M	CO2	3

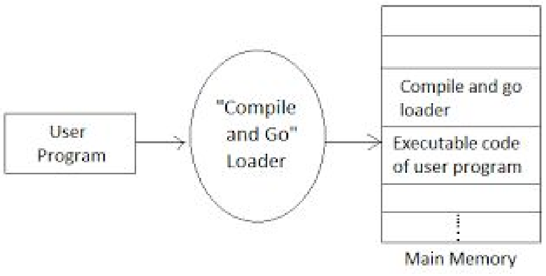
~ All the best!!! ~

Solution for SPCC Test 2 / Academic Year 2018-19 / Department of Computer Engineering

Q 1 a. Loader Schemes: Based on the various functionalities of loader, there are various types of loaders:

1. “compile and go” loader / Assemble and go
2. General Loader Scheme
3. Absolute Loader
4. Direct Linking Loader
5. Bootstrap Loader

“compile and go” loader: in this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter. The typical example is WATFOR-77, it’s a FORTRAN compiler which uses such “load and go” scheme. This loading scheme is also called as “**assemble and go”.**



Advantages:

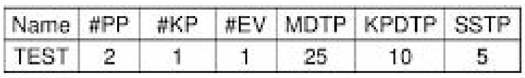
a. This scheme is simple to implement. Because assembler is placed at one part of the memory and loader simply loads assembled machine instructions into the memory.

Disadvantages:

a. In this scheme some portion of memory is occupied by assembler which is simply a wastage of memory. As this scheme is combination of assembler and loader activities, this combination program occupies large block of memory.
b. There is no production of .obj file, the source code is directly converted to executable form. Hence even though there is no modification in the source program it needs to be assembled and executed each time, which then becomes a time consuming activity.
c. It cannot handle multiple source programs or multiple programs written in different languages. This is because assembler can translate one source language to other target language.
d. For a programmer it is very difficult to make an orderly modulator program and also it becomes difficult to maintain such program, and the “compile and go” loader cannot handle such programs.
e. The execution time will be more in this scheme as every time program is assembled and then executed

Q1 b. MDT &MNT

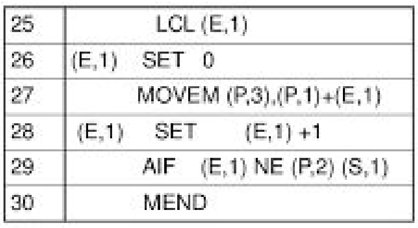
a. Macro Name Table (MNT) contains Macro name, Number of positional parameter(#PP), Number of keyword parameter(#KP), Number of expansion time variables(#EV), MDT pointer (MDTP). KPDTAB pointer (KPDTP). SSTAB pointer (SSTP)
b. Macro Definition Table (MDT) stores : Label, Opcode, Operands



Name	#PP	#KP	#EV	MDTP	KPDTP	SSTP
TEST	2	1	1	25	10	5

MNT

MDT



25	LCL (E,1)
26	(E,1) SET 0
27	MOVEM (P,3),(P,1)+(E,1)
28	(E,1) SET (E,1) +1
29	AIF (E,1) NE (P,2) (S,1)
30	MEND

Solution for SPCC Test 2 / Academic Year 2018-19 / Department of Computer Engineering

Q1 c. **Parameterized macros** **:** can be used to create templates for frequently used parts of code. Frequently, they serve as simple functions. #define helps to define a macro and the #undefine helps to remove a macro definition. Syntax of a parameterized macro is as follows:

```
    #define        identifier(parameterlist)        replacement-list
```

There should be no spaces between the identifier and the parameter list. There should be no spaces in the parameter list.

Example:

```
#define     getchar()     getc(stdin)

#define     IS_EVEN(m)    ((m)%2 == 0)

#define     max(a,b)      (((a)>(b))?(a):(b))

#define     toupper(x)    (('a'<=(x)&&(x) z="" x="" -="" a="" :="" br="">

#define     print(y)      printf("%d\n",y) print(c/d) /* becomes printf("%d\n",c/d); */
```

Advantages of a parameterized macro:

a. The compiled code will execute more quickly because the macro is substituted in line. No function calling overhead is wanted.
b. The parameters of functions are typed but in macros they are not typed. So, they are generic. The max() macro given example will perform for floats and also for ints.

Disadvantages of a parameterized macro:

a. The compiled code will often be larger, mostly when macros are nested (e.g., max(a,max(b,max(c,d)))) because each instance of the macro is expanded in line.
b. A macro may calculate its arguments more than once, producing subtle faults and it is not achievable to have a pointer to a macro.

Q1 d . **There are 3 representations of three address code namely**

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple – It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

2. Triples – This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

3. Indirect Triples – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Write quadruple, triples and indirect triples for following expression : (x + y) * (y + z) + (x + y + z)

Explanation – The three address code is:

t1 = x + y

t2 = y + z

t3 = t1 * t2

t4 = t1 + z

t5 = t3 + t4

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

List of pointers to table

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

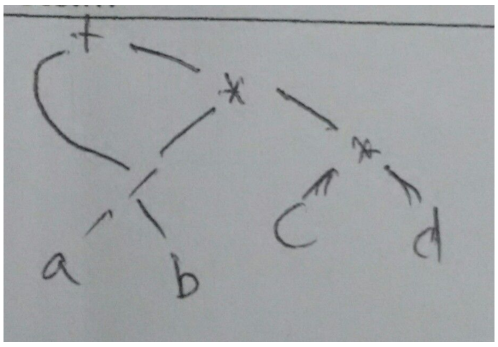
#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

Q1 e. DAG notation for (a/b)+(a/b)*(c*d).



Q1 f. Functions of a Loader

The loader is responsible for the activities such as allocation, linking, relocation and loading

1. It allocates the space for program in the memory, by calculating the size of the program. This activity is called **allocation.**
2. It resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses. This activity is called **linking.**

3. There are some address dependent locations in the program, such address constants must be adjusted according to allocated space, such activity done by loader is called **relocation.**
4. Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution, this activity is called **loading.**

Q 2 a. Different issues in design of code generation.

1. **Input to code generator** is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's etc. Assume that they are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.
2. **Target program** is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.
    - Absolute machine language as an output has advantages that it can be placed in a fixed memory location and can be immediately executed.
    - Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader.
    - Assembly language as an output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code.
3. **Memory Management** – Mapping the names in the source program to addresses of data objects is done by the front end and the code generator. A name in the three address statement refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.
4. **Instruction selection** – Selecting best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered.But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, the respective three-address statements would be translated into latter code sequence as shown below:

P:=Q+R
S:=P+T
	MOV Q, R0
	ADD R, R0
	MOV R0, P
	MOV P, R0
	ADD T, R0
	MOV R0, S

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

5. **Register allocation issues** – Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:

- ○ During **Register allocation** – we select only those set of variables that will reside in the registers at each point in the program.
- ○ During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

As the number of variables increase, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example : M a, b

These types of multiplicative instruction involve register pairs where a, the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

6. **Evaluation order** – The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in general case is a difficult NP-complete problem.
7. **Approaches to code generation issues:** Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:
    - ○ Correct
    - ○ Easily maintainable
    - ○ Testable
    - ○ Maintainable

Q2 b. Different types of code optimization techniques in compiler design

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives :

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

When to Optimize?

Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Types of Code Optimization –The optimization process can be broadly classified into two types :

1. **Machine Independent Optimization** – This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.
2. **Machine Dependent Optimization** – Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references

rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

Code Optimization is done in the following different ways :

1. **Compile Time Evaluation :**

(i) A = 2*(22.0/7.0)*r
Perform 2*(22.0/7.0)*r at compile time.

(ii) x = 12.4
y = x/2.3
Evaluate x/2.3 as 12.4/2.3 at compile time.

2. **Variable Propagation**

//Before Optimization	//After Optimization

c = a * b
x = a
till
 	
d = x * b + 4
	c = a * b
	x = a
	till
	d = a * b + 4

Hence, after variable propagation, a*b and x*b will be identified as common sub-expression.

3. **Dead code elimination :** Variable propagation often leads to making assignment statement into dead code

c = a * b
x = a
till
d = a * b + 4
	//After elimination :
	c = a * b
	till
	d = a * b + 4

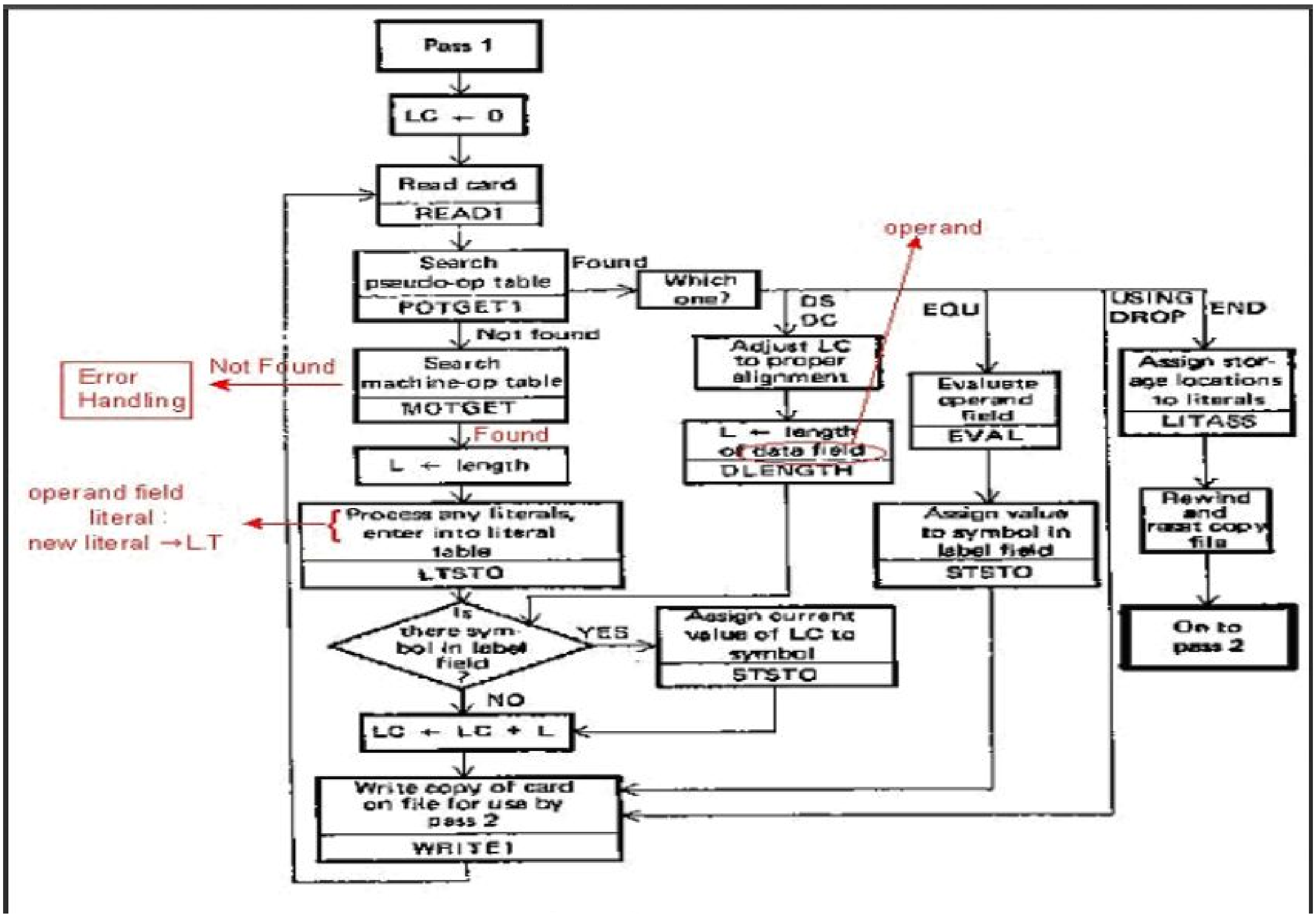
4. **Code Motion :**
   - Reduce the evaluation frequency of expression.
   - Bring loop invariant statements out of the loop.

a = 200;
 while(a>0)
 {
    b = x + y;
    if (a % b == 0}
    printf("%d", a);
  }
	//This code can be further optimized as
	a = 200;
	b = x + y;
	while(a>0)
	 {
	    if (a % b == 0}
	    printf("%d", a);
	  }

5. **Induction Variable and Strength Reduction :**
   - An induction variable is used in loop for the following kind of assignment i = i + constant.
   - Strength reduction means replacing the high strength operator by the low strength.

i = 1;
 	
while (i<10)
 	
{
 	
   y = i * 4;
}
	//After Reduction
	i = 1
	t = 4
	{
	  while(t<40)
	  y = t;
	  t = t + 4;
	}

Q3 a. Flow chart for pass 1 of two pass assembler.



Q3 b.

Source Prog	Copy File	Pass2 output
	LC	LC
PG3 START	0	0
USING *, BASE	0	0
SR 1,1	0 SR 1,1	0 SR 1,1

Solution for SPCC Test 2 / Academic Year 2018-19 / Department of Computer Engineering

	L	1, FOUR	2	L	1, ____	2	L	1, 18(0,8)
	A	1, FIVE	6	A	1, ____	6	A	1, 22(0,8)
	A	1, =F'3'	10	A	1, _____	10	A	1, 32(0,8)
	ST	1, TEMP	14	ST	1, _____	14	ST	1, 26(0,8)
FOUR	DC,	F'4'	18	'4'		16	0100	
FIVE	DC,	F'5'	22	'5'		20	0101	
BASE	EQU	8	26	8		24		
TEMP	DS	'1'F	26			24		
	END		30			28		
						32	0011	

MOT :

POT :

ST:

Symbol	Value	Length	R/A
PG3	00	1	R
FOUR	18	4	R
FIVE	22	4	R
BASE	26 / 8	1	A
TEMP	26	4	R

LT:

Literal	Value	Length	R/A
=F'3'	32	04	R

BT

Register	Availability Indicator	Contents
1	'N'	
. .	'N'	
8	'Y'	00
. .	'N'	
15	'N'	