

PART A
(PART A : TO BE REFERRED BY STUDENTS)
Experiment No. 10(A)

Aim : To generate target code.

Objective: Understand intermediate code generation and code generation phase of compiler.
Develop a program to generate target code

Outcome: Students are able to implement a program to generate target code

Theory:

The last phase of compiler is the code generator. The intermediate representation of the source program is given as input to code generator and it produces target program as an output as shown in figure.

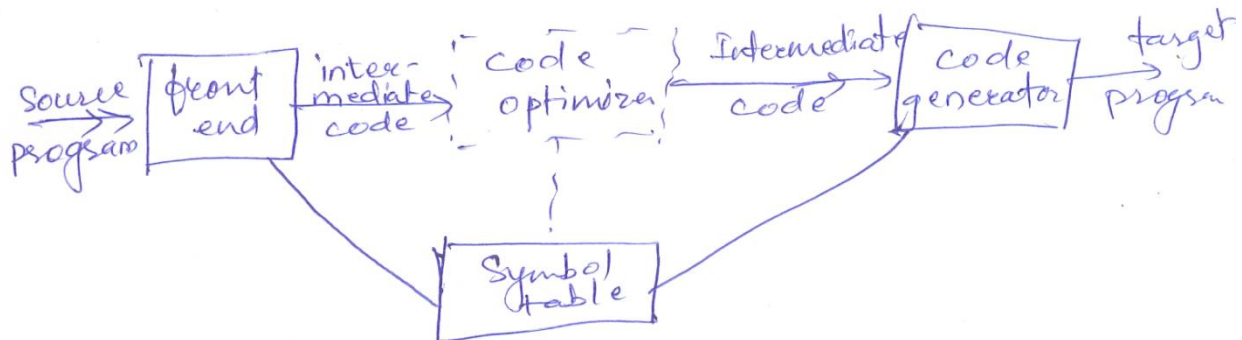


FIGURE Position of code generator

A Code-Generation algorithm-

The code generation algorithm takes a sequence of three-address statements as a input from a basic block. For each three-address statement of the form $x := y \text{ op } z$ we do the following actions:

1. To store the result of $y \text{ op } z$, determine a location L , by invoking **getreg** function. L may be a register or a memory location.
2. To determine y' , current location of y , consult address descriptor of y . If value of y is currently both in memory and register prefer register for y' . If value of y is not already in L then generate instruction **MOV L, y'** to place a copy of y' in L .
3. Generate instruction **$\text{op } z', L$** where z' is current location of z . If z is in both register and memory, prefer value from register. Update address descriptor of x to indicate value of x is in location L . If L is a register, update its register descriptor to indicate that it contains value of x and remove x from all other register descriptors.
4. If the current values of y and z are not used in future or they are not live after exit from the block and are in the register, alter the register descriptor to indicate that those register no longer contains the value of y and z .

The Function getreg

Function getreg returns the location for the value of x in the assignment $x = y \text{ op } z$.

1. If the name y is in a register that holds the value of no other names and y is not live and has no next use after execution of $x = y \text{ op } z$, then returns the register of y is no longer in L.
2. Failing 1, return an empty register for L if there is one.
3. failing 2, if x has a next use in a block, op is an operator, such as indexing, that requires a register, find an occupied register R. Store the value of R into a memory location (by MOV R,M) if it is not already in the proper memory location M, update the address descriptor for M, and return R. If R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable occupied register might be one whose value is also in memory.
4. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L.

PART B

(PART B : TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded at the end of the practical)

Roll. No. B48	Name: Aryan Unhale
Class: TE B COMPS	Batch: B3
Date of Experiment:29/3/25	Date of Submission:4/4/25
Grade:	

B.1 Software Code written by student:

```
#include <stdio.h>
#include <string.h>

// Function to generate target code from three-address code
void generateTargetCode(char op[], char arg1[], char arg2[], char result[]) {
    printf("Generating target code for: %s = %s %s %s\n", result, arg1, op, arg2);

    if (strcmp(op, "+") == 0)
        printf("ADD %s, %s\n", arg1, arg2);
    else if (strcmp(op, "-") == 0)
        printf("SUB %s, %s\n", arg1, arg2);
    else if (strcmp(op, "*") == 0)
        printf("MUL %s, %s\n", arg1, arg2);
    else if (strcmp(op, "/") == 0)
        printf("DIV %s, %s\n", arg1, arg2);
    else
        printf("Unsupported operation\n");

    printf("MOV %s, RESULT\n\n", result);
}

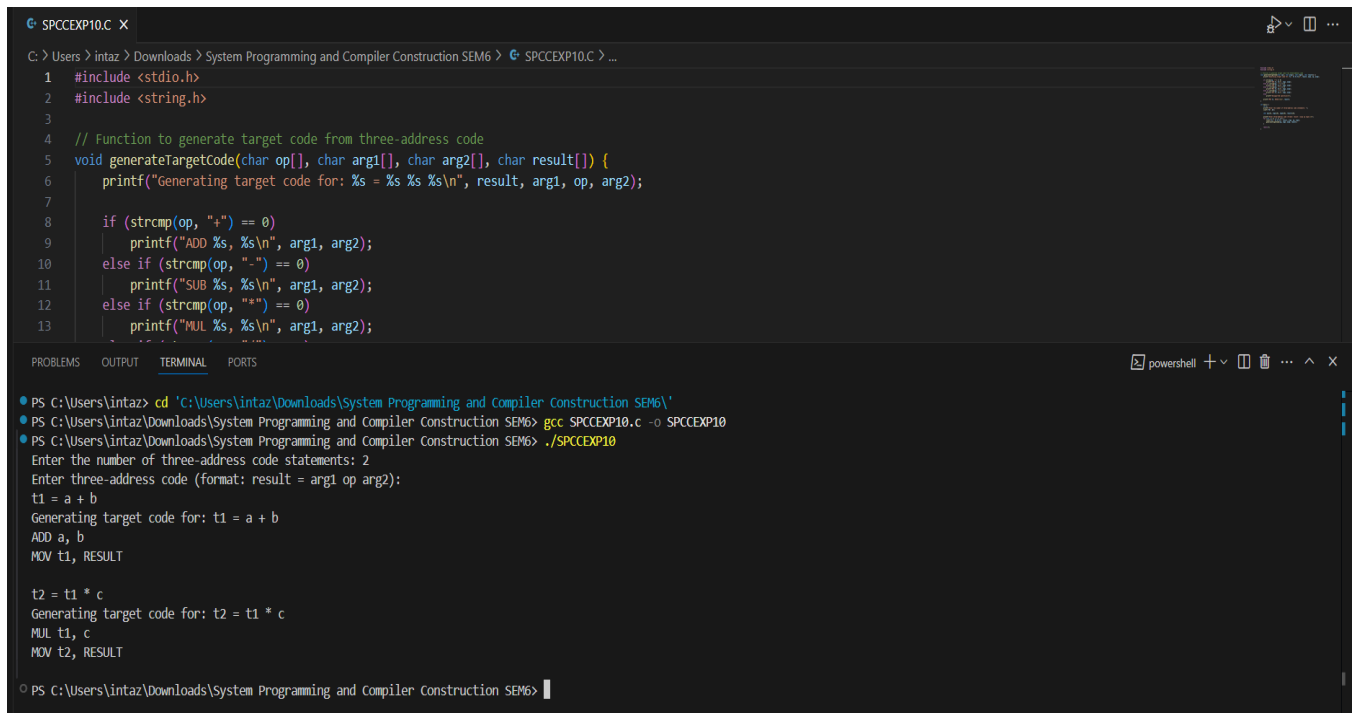
int main() {
    int n;
    printf("Enter the number of three-address code statements: ");
    scanf("%d", &n);

    char op[10], arg1[10], arg2[10], result[10];

    printf("Enter three-address code (format: result = arg1 op arg2):\n");
    for (int i = 0; i < n; i++) {
        scanf("%s = %s %s %s", result, arg1, op, arg2);
        generateTargetCode(op, arg1, arg2, result);
    }
}
```

```
    return 0;
}
```

B.2 Input and Output:



```
SPCCEXP10.C X
C:\Users\intaz> Downloads > System Programming and Compiler Construction SEM6 > SPCCEXP10.C > ...
1  #include <stdio.h>
2  #include <string.h>
3
4  // Function to generate target code from three-address code
5  void generateTargetCode(char op[], char arg1[], char arg2[], char result[]) {
6      printf("Generating target code for: %s = %s %s %s\n", result, arg1, op, arg2);
7
8      if (strcmp(op, "+") == 0)
9          printf("ADD %s, %s\n", arg1, arg2);
10     else if (strcmp(op, "-") == 0)
11         printf("SUB %s, %s\n", arg1, arg2);
12     else if (strcmp(op, "*") == 0)
13         printf("MUL %s, %s\n", arg1, arg2);
14 }
15
16 int main() {
17     char op[10], arg1[10], arg2[10], result[10];
18     printf("Enter the number of three-address code statements: ");
19     int n;
20     scanf("%d", &n);
21     for (int i = 0; i < n; i++) {
22         printf("Enter three-address code (format: result = arg1 op arg2): ");
23         scanf("%s = %s %s %s", result, arg1, op, arg2);
24         generateTargetCode(op, arg1, arg2, result);
25     }
26     return 0;
27 }
```

```
PS C:\Users\intaz> cd 'C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6\'
PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6> gcc SPCCEXP10.c -o SPCCEXP10
PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6> ./SPCCEXP10
Enter the number of three-address code statements: 2
Enter three-address code (format: result = arg1 op arg2):
t1 = a + b
Generating target code for: t1 = a + b
ADD a, b
MOV t1, RESULT

t2 = t1 * c
Generating target code for: t2 = t1 * c
MUL t1, c
MOV t2, RESULT

PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6>
```

B.3 Observations and learning:

During the implementation of the target code generation, the experiment demonstrated how an intermediate representation of a program is converted into machine-level instructions. The process involved generating target code from three-address code using the code generation algorithm and the `getreg` function.

B.4 Conclusion:

The experiment successfully demonstrated the code generation phase of a compiler, which translates an intermediate representation into efficient machine instructions. Through this process, we understood the role of register allocation, instruction selection, and optimization in generating high-performance target code.

B.5 Question of Curiosity

1. Differentiate Between Machine Dependent and Machine Independent code optimization.

Feature	Machine-Dependent Optimization	Machine-Independent Optimization
Definition	Optimization techniques that consider the underlying hardware architecture.	Optimization techniques that improve code without relying on a specific machine.
Scope	Focuses on specific processor architecture, instruction set, and hardware resources.	Works at the intermediate representation level, applicable to any machine.
Examples	Register allocation, instruction scheduling, pipeline optimization.	Common sub-expression elimination, dead code elimination, loop optimization.
Portability	Code optimized for one machine may not perform well on another.	Optimizations remain effective across different architectures.
Execution Efficiency	Leads to highly efficient execution on specific hardware.	Ensures code correctness and efficiency without hardware dependency.

2. Describe Peephole Optimization.

→ Peephole optimization is a type of Code Optimization performed on a small part of the code. It is performed on a very small set of instructions in a segment of code.

→ The small set of instructions or a small part of the code on which peephole optimization is performed is known as peephole or window.

→ It works on the theory of replacement in which a part of code is replaced by shorter and faster code without change in output. The peephole is machine-dependent optimization.

→ The objective of peephole optimization is:

1. To improve performance
2. To reduce memory footprint
3. To reduce code size

Peephole Optimization Techniques:

1. Redundant load and store elimination:

In this technique, redundancy is eliminated.

Initial code:

```
y = x + 5;
```

```
i = y;
```

```
z = i;
```

```
w = z * 3;
```

Optimized code:

`y = x + 5;`

`i = y;`

`w = y * 3;`

2. Constant folding:

The code that can be simplified by the user itself, is simplified.

Initial code:

`x = 2 * 3;`

Optimized code:

`x = 6;`

53. Strength Reduction:

The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code:

`y = x * 2;`

Optimized code:

`y = x + x;` or `y = x << 1;`

Initial code:

`y = x / 2;`

Optimized code:

`y = x >> 1;`

4. Null sequences:

Useless operations are deleted.

5. Combine operations:

Several operations are replaced by a single equivalent operation.