EXP NO: 9 **DATE: 16.04.25**

DEVELOP THE BACK-END OF A COMPILER THAT TAKES THREE-ADDRESS CODE (TAC) AS INPUT AND GENERATES CORRESPONDING 8086 ASSEMBLY LANGUAGE CODE AS OUTPUT.

AIM:

To design and implement the back-end of a compiler that takes three-address code (TAC) as input and produces 8086 assembly language code as output. The three-address code is an intermediate representation used by compilers to break down expressions and operations, while the 8086 assembly code is a machine-level representation of the program that can be executed by a processor.

ALGORITHM:

1. Parse the Three-Address Code (TAC):

Input: Three-Address Code, which is an intermediate representation. For example:

```
t0 = b + c
t1 = t0 * d
a = t1
```

Output: 8086 assembly language code. For example:

```
MOV AX, [b]       ; Load b into AX
ADD AX, [c]       ; Add c to AX
MOV [t0], AX      ; Store result in t0
```

---

2. Process Each TAC Instruction:

1. **Initialize Registers**:
    - Set up the registers in 8086 assembly (e.g., AX, BX, CX, etc.) for storing intermediate results and final outputs.
    - Maintain a temporary register counter for naming temporary variables in TAC (e.g., t0, t1).
2. **For each TAC instruction**, based on its operation:
    - Identify the components: operands and operator.
    - Choose an appropriate register (AX, BX, etc.) for storing intermediate results.
    - If the operation involves multiple operands or temporary variables, map them to registers.

---

3. Translating TAC to 8086 Assembly:

- **Addition/Subtraction (e.g., t0 = b + c)**:
    - Load operands into registers and perform the operation:

```
MOV AX, [b]        ; Load b into AX
ADD AX, [c]        ; Add c to AX
MOV [t0], AX       ; Store result in t0
```

- **Multiplication (e.g., t1 = t0 * d)**:
  - Load operands into registers and perform the operation:

    ```
    MOV AX, [t0]       ; Load t0 into AX
    MOV BX, [d]        ; Load d into BX
    MUL BX             ; Multiply AX by BX (result in AX)
    MOV [t1], AX       ; Store result in t1
    ```

- **Assignment (e.g., a = t1)**:
  - Move the value from a temporary variable to the target variable:

    ```
    MOV [a], [t1]      ; Move value of t1 into a
    ```

- **Division (e.g., t2 = b / c)**:
  - Division is a bit more complex due to the 8086's limitations with the DIV instruction. For example, the result might need to be stored in AX or DX:AX (if it's a 32-bit result):

    ```
    MOV AX, [b]        ; Load b into AX
    MOV DX, 0          ; Clear DX (important for division)
    MOV BX, [c]        ; Load c into BX
    DIV BX             ; AX = AX / BX (quotient in AX, remainder in DX)
    MOV [t2], AX       ; Store quotient in t2
    ```

4. Manage Memory and Registers:

- **Variables**: Variables like a, b, c are stored in memory, so you will use memory addressing modes such as [variable_name] to access them.
- **Temporary Variables**: Temporary variables like t0, t1, t2, etc., are stored in registers (AX, BX, etc.) or memory if there are more variables than registers available.

---

5. Handle Control Flow (Optional):

If the TAC contains control structures (such as loops, if-else statements, or function calls), you will need to generate labels and jump instructions in 8086 assembly.

- **If Statements**: For example, if (x > 0) { y = 1; } could generate:

  ```
  MOV AX, [x]
  CMP AX, 0
  JG positive_case  ; Jump if greater
  JMP end_if
  ```

PROGRAM:

```
#include <stdio.h>

#include <string.h>

void generateAssembly(const char* tac) {

    char result[10], op1[10], op[2], op2[10];

    // Parse the TAC instruction

    sscanf(tac, "%s = %s %s %s", result, op1, op, op2);

    // Generate assembly code based on the operator

    if (strcmp(op, "+") == 0) {

        printf("MOV AX, [%s]\n", op1);

        printf("ADD AX, [%s]\n", op2);

        printf("MOV [%s], AX\n", result);

    } else if (strcmp(op, "-") == 0) {

        printf("MOV AX, [%s]\n", op1);

        printf("SUB AX, [%s]\n", op2);

        printf("MOV [%s], AX\n", result);

    } else if (strcmp(op, "*") == 0) {

        printf("MOV AX, [%s]\n", op1);

        printf("MOV BX, [%s]\n", op2);

        printf("MUL BX\n");

        printf("MOV [%s], AX\n", result);

    } else if (strcmp(op, "/") == 0) {

        printf("MOV AX, [%s]\n", op1);

        printf("MOV BX, [%s]\n", op2);

        printf("DIV BX\n");

        printf("MOV [%s], AX\n", result);

    } else {

        printf("Unsupported operation: %s\n", op);

    }

}

int main() {

    const char* tacInstructions[] = {
```

```
        "t0 = b + c",
        "t1 = t0 * d",
        "a = t1"
    };
    int numInstructions = sizeof(tacInstructions) / sizeof(tacInstructions[0]);
    for (int i = 0; i < numInstructions; i++) {
        generateAssembly(tacInstructions[i]);
        printf("\n");
    }
    return 0;
}
```

OUTPUT :

```
MOV AX, [b]
ADD AX, [c]
MOV [t0], AX

MOV AX, [t0]
MOV BX, [d]
MUL BX
MOV [t1], AX

MOV AX, [t1]
MOV [a], AX
```

RESULT:

Thus the above example provides a foundational approach to converting TAC to 8086 assembly using C. For a complete compiler back-end, you would need to handle additional aspects such as register allocation, memory management, and more complex control flow constructs.