4.1 Introduction

Object Code Generation or Code Generation phase works for the generation of object code from the intermediate representation. This phase assumes:

1. Lexical Analysis (LA) phase, Syntax Analysis (SA) and Intermediate Code Generation (ICG) phase are completed without errors.
2. The Code Optimization phase might have improved the intermediate code.

So, the efficient use of registers generates the efficient object codes from the intermediate code.

Example:

$T_1 = A + B$

$T_2 = C + D$

$T_3 = T_1 + T_2$

Sample Object Codes:

```
LOAD      A
ADD       B
STORE     T1
LOAD      C
ADD       D
STORE     T2
LOAD      T1
ADD       T2
STORE     T3
```

If we make use of some registers:

```
LOAD      A, R0
ADD       B, R0
```

```
LOAD      C, R1
ADD       D, R1
ADD       R0, R1
```

Now, the object code is a set of machine instructions of the form:

OPCODE	OPERAND

The object code generation deals with:

→ Locating a Register to the current operation or statement.
→ Generating appropriate machine instruction.
→ The use of machine specific *idioms*.

Every variable name in the intermediate code of a block is associated with an address descriptor which contain the location where the value can be found at run time.

4.2 Register and Address Descriptors

During object code generation, the algorithm uses the descriptors (Register and Address) to keep track of register contents and addresses for names.

So, a **register descriptor** keeps track of what is currently in each register. The code generation algorithm consult it whenever a new register is needed. Initially the register descriptor shows all the registers are empty and as the code generation started, each register will hold the value of zero or more names at any given time.

An **address descriptor** keeps track of the location (or locations) where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address or some set of these. This information can be store in the symbol table and is used to determine the accessing method for a name.

4.3 Object Code Generation Algorithm

Let the intermediate code are in the form (three-address) of `A = B op C` and the code generation algorithm takes as input a sequence of three-address statements constituting a basic block.

Step – 1: Get a location (*L*) using `GETREG()` function where the result of `B op C` will be stored. Usually *L* will be a register but could be a memory location.

Step – 2: If the address descriptor of *B* does not contain *L*, generate the the below instruction to place a copy of *B* in *L*.

`MOVE B', L` ;where *B*' → preferably register where *B* is found at run time.

If the address descriptor of *B* contains the current location of *B*, prefer the register for *B'* if the value of *B* is currently both in memory and a register.

Step – 3: Generate the below instruction where *C'* is a current location of *C*.

`OP C', L`

Update the address descriptor of *A* to indicate that *A* is in location *L*.

Step – 4: The register descriptor of *L* is set to *A*, i.e. the address descriptor of *A* now contain *L*. If *A* has *no_next_use* and not *LIVE* then, Generate Instruction

`MOVE L, A`

and update register descriptor of *L* by removing *A* and also remove *L* from the address descriptor of *A*.

GETREG()

Step – 1: Consult address descriptor of *B* to check whether *B* is in a Register (*L*) and the register descriptor of *L* does not contain any other except *B*.
⇒ Return (*L*).

Step – 2: Failing (1), Locate a new Register (*L*).
⇒ Return (*L*).

Step – 3: Failing (2),

(a) Find a Register (*L*) of which the register descriptor contains names haveing no_next_use and not_live then
⇒ Return (*L*).

(b) Move the content of the Register (*L*) to the equivalent memory location and,
⇒ Return (*L*).

Step – 4: Failing 3,
⇒ Return the address of *B* as *L*.

Example:

Let $X = (A–B) + (A–C) + (A–C)$ be a statement then the intermediate representation for the sentence is:

$T_1 = A - B$
$T_2 = A - C$
$T_3 = T_1 + T_2$
$T_4 = T_3 + T_2$
$X = T_4$

Intermediate Code	Register	Address and Register Descriptor		Code
$T_1 = A - B$	R_0	*A* is in R_0.	R_0 contains A.	MOV A,R_0
		T_1 is in R_0.	R_0 contains T_1.	SUB B,R_0
$T_2 = A - C$	R_1	A is in R_1.	R_1 contains A.	MOV A,R_1
		T_2 is in R_1.	R_1 contains T_2.	SUB C,R_1
$T_3 = T_1 + T_2$	R_0	T_2 is in R_1. T_3 is in R_0.	R_0 contains T_3. R_1 contains T_2.	ADD R_1,R_0
$T_4 = T_3 + T_2$	R_0	T_4 is in R_0.	R_0 contains T_4.	ADD R_1,R_0
$X = T_4$				MOV R_0,X

H/W: Change this algorithm to generate code from *A = op B.*

4.4 Peephole Optimization

(a). Elimination of Redundant LOAD and STORE operation.

e.g.

```
1:    ADD        B
2:    STORE      A
3:    LOAD       B
```

(b). Reduction of Strength

[Reduction of strong *opcode* by weak *opcode* if possible]

(c). Removal of Unreachable Code

Any unlabeled instruction following an unconditional jump is unreachable and this should be removed during peephole optimization.

e.g.

```
20:   ............
21:   ............
22:   goto 20
23:   ............
24:   ............
25:   ............
26:   ............
27:   goto 25
28:   ............
```

(d). Elimination of Multiple Jumps

e.g.

```
    if a < b goto L1
    ...................
L1: goto L2
    ...................
L2: ...................
```

4.5 Use of Machine Idioms

In some machines, a set of special operation codes are used to generate cost-effective instructions.

e.g.: for a sentence like, `I = I + 1`

```
LOAD     I
ADD      1
STORE    I
```

We can replace this set of codes by:

```
AOS      I
```

i.e. Add one to the storage. To get optimized code, we need to make use of machine idioms.