UNIT 4

Code Generation and Instruction Selection

Optimization

- Optimization can be performed at two levels:
- 1. **Machine Dependent optimization:** it is performed through a better choice of instructions, better addressing modes and usage of machine registers.

- **2.Machine Independent optimization:** It is based on the use of semantics preserving transformations applied independent of the target machine. This includes optimizations like common sub-expression elimination and loop optimization.

Properties of Code transformation

- 1. Preserve the meaning of programs.
- 2. Improve the Speed efficiency.
- 3. Worth the effort.

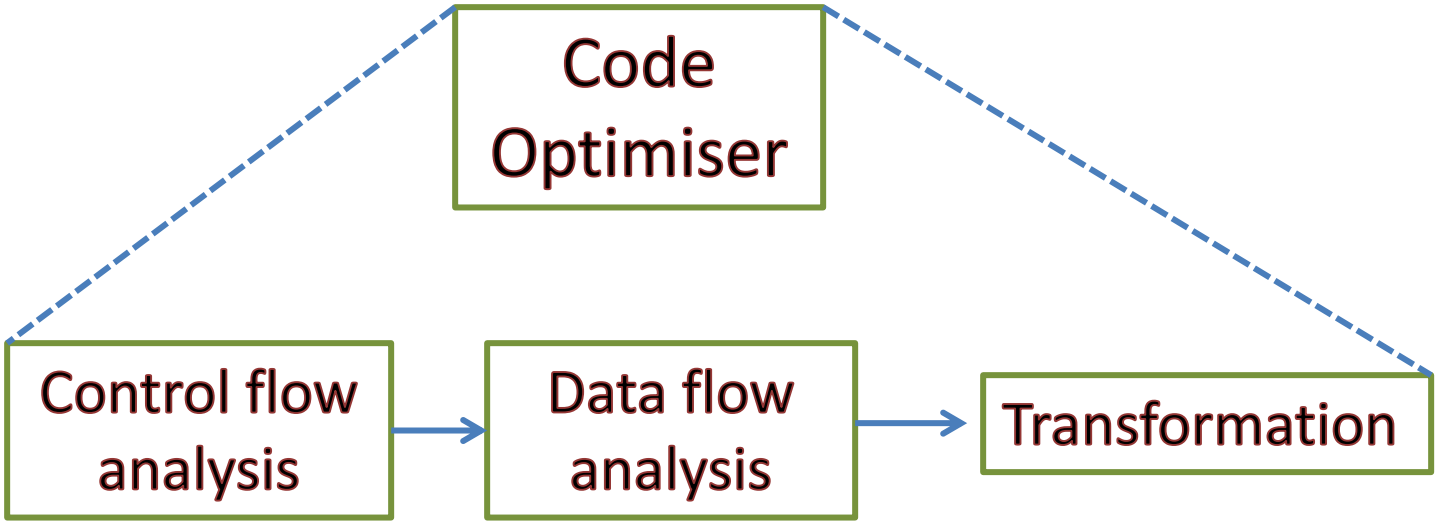


Fig: Types of Code Optimisation

Steps involved in the construction of flow graphs

1. Computation of TAC for the source code segment.
2. Construction of basic blocks.
3. Construction of flow graphs
4. Code optimisation.
5. Construction of DAGs for each basic block in the flow graph.
6. Optimisation.

Procedure for Construction of Basic Blocks

1. Determine the set of leaders.

The following rules are followed in determining the leaders:

a. First statement is a leader.

b. Any statement which is the target of conditional or unconditional goto is a leader.

c. Any statement which initially follows a conditional statement is a leader.

2. For each leader construct its **basic block** which consists of the leader and all statement upto but not including the next leader or the end of the program.

3. Any statement not placed is a block can never be executed and hence can be removed if desired.

Example

Three Address Code for Quick Sort

```
(1)  i=m-1
(2)  j=n
(3)  t1=4*n
(4)  v=a[t1]
(5)  i=i+1
(6)  t2=4*i
(7)  t3=a[t2]
(8)  if t3<v goto (5)
(9)  j=j-1
(10) t4=4*j
(11) t5=a[t4]
(12) if t5>v goto (9)
(13) if i>=j goto (23)
(14) t6=4*i
(15) x=a[t6]
```

```
(16) t7=4*i
(17) t8=4*j
(18) t9=a[t8]
(19) a[t7] = t9
(20) t10=4*j
(21) a[t10]=x
(22) goto (5)
(23) t11=4*i
(24) x=a[t11]
(25) t12=4*i
(26) t13=4*n
(27) t14=a[t13]
(28) a[t12]=t14
(29) t15=4*n
(30) a[t15]=x
```

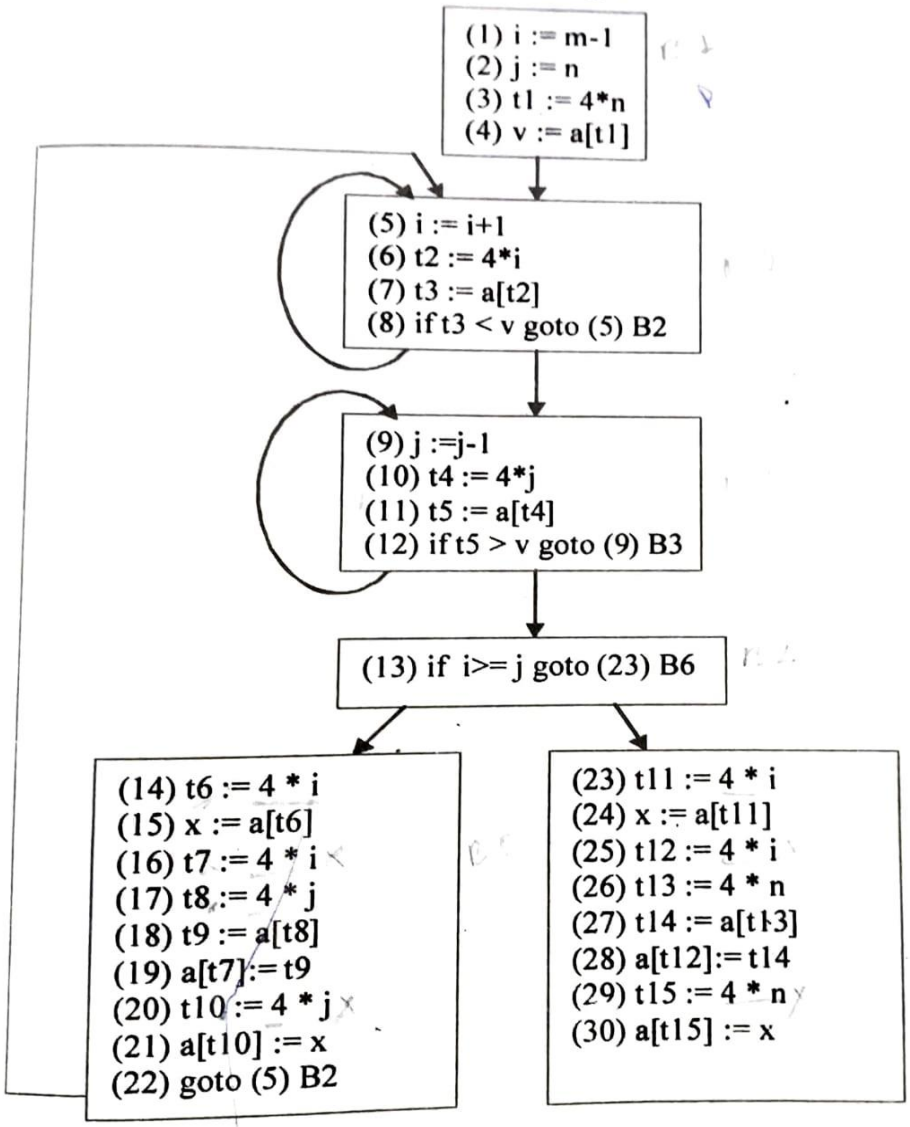
Basic Blocks and corresponding statements

Blocks	Statements
B1	1 to 4
B2	5 to 8
B3	9 to 12
B4	13
B5	14 to 22
B6	23 to 30

Construction of Flow Graphs

- A flow graph shows the directed relationship between the basic blocks and their successors.
- The nodes of the flow graph are the basic blocks.

Example 9.3: Draw the flow graph for TAC in fig 9.4.



Principle Source of Code Optimisation

1. Function-preserving Transformations:
   a. Common Sub-expression (CSE) Elimination
   b. Copy Propagation
   c. Dead Code Elimination
2. Loop Optimisation:
   a. Code Motion
   b. Induction Variable Elimination
   c. Reduction in Strength

Function Preserving Transformations

a. Common Sub-expression (CSE) Elimination

```
(14) t6=4*i
(15) x=a[t6]
(17) t8=4*j
(18) t9=a[t8]
(19) a[t6] = t9
(21) a[t8]=x
(22) goto (5)
```

```
(23)t11=4*i
(24) x=a[t11]
(26) t13=4*n
(27) t14=a[t13]
(28) a[t11]=t14
(30)a[t13]=x
```

Fig: B5 & B6 after elimination of common subexpressions

```
(15) x=t3

(19) a[t2] = t5
(21) a[t4]=x
(22) goto (5)
```

```
(24) x=t3

(27) t14=a[t1]
(28) a[t2]=t14
(30)a[t1]=x
```

Fig: B5 & B6 after global elimination of common subexpressions

b. Copy Propagation

Fig: Copy propagation in B5 and B6

c. Dead Code Elimination

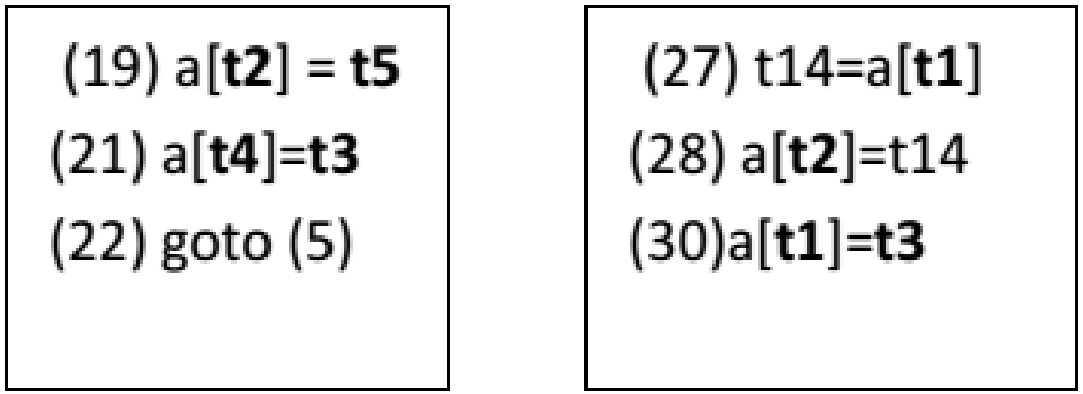
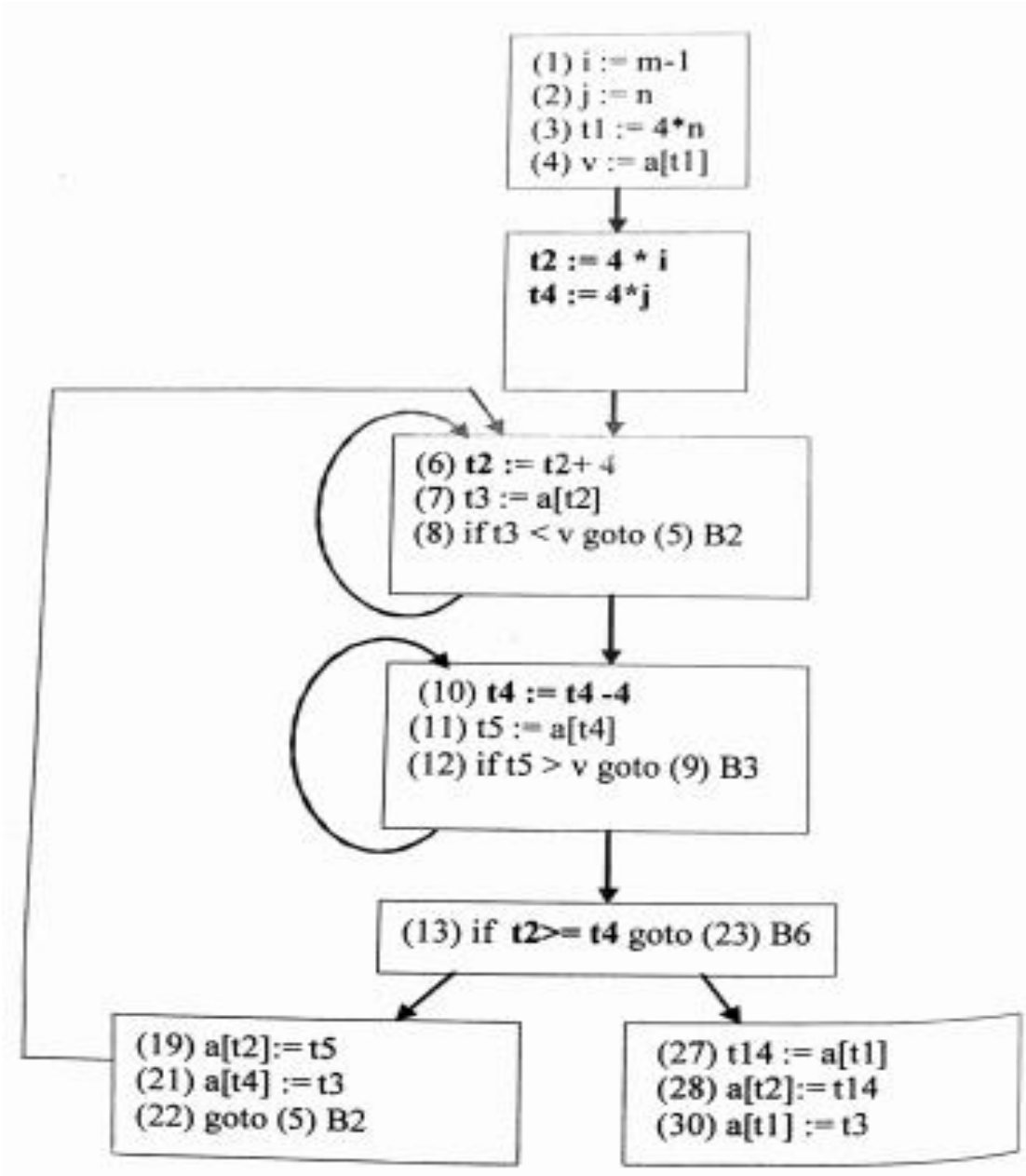


Fig: Elimination of dead code in B5 and B6


(1) i := m-1
(2) j := n
(3) t1 := 4*n
(4) v := a[t1]
t2 := 4 * i
t4 := 4*j
(6) t2 := t2+ 4
(7) t3 := a[t2]
(8) if t3 < v goto (5) B2
(10) t4 := t4 -4
(11) t5 := a[t4]
(12) if t5 > v goto (9) B3
(13) if t2>= t4 goto (23) B6
(19) a[t2]:= t5
(21) a[t4] := t3
(22) goto (5) B2
(27) t14 := a[t1]
(28) a[t2]:= t14
(30) a[t1] := t3

Loop Optimisation

Loop Optimisation

- Loop optimization is most valuable machine-independent optimization
  - program's inner loop takes more time to execute.

- If we decrease the number of instructions in an inner loop
  - then the running time of a program may be improved even if we increase the amount of code outside that loop.

a. Code Motion

- In this technique, loop invariants are removed from the loop and placed before it.
- Example:

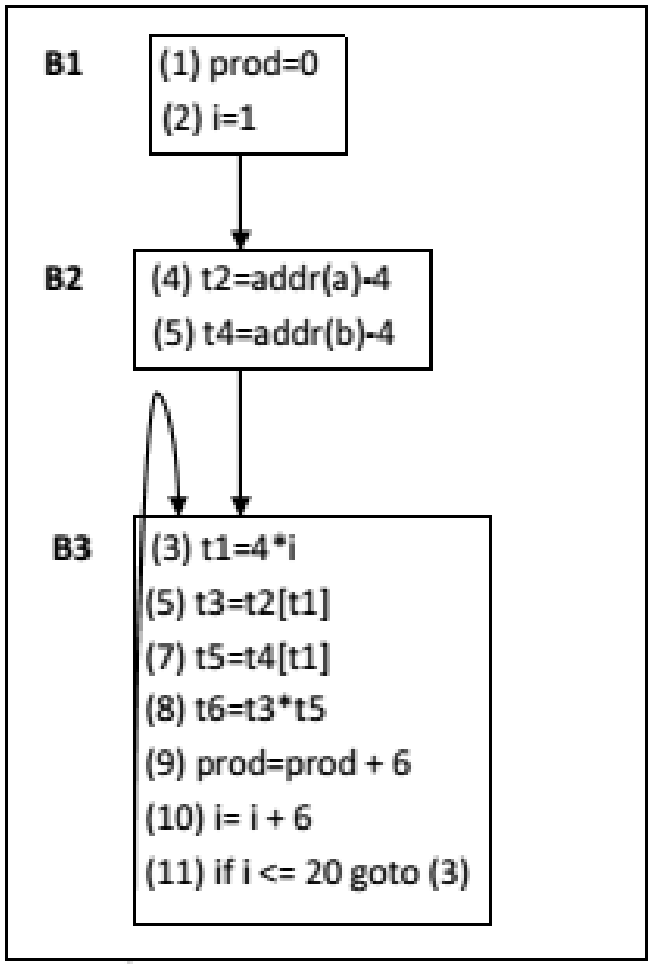
```
while(i<=n-3)
```

rewritten code:

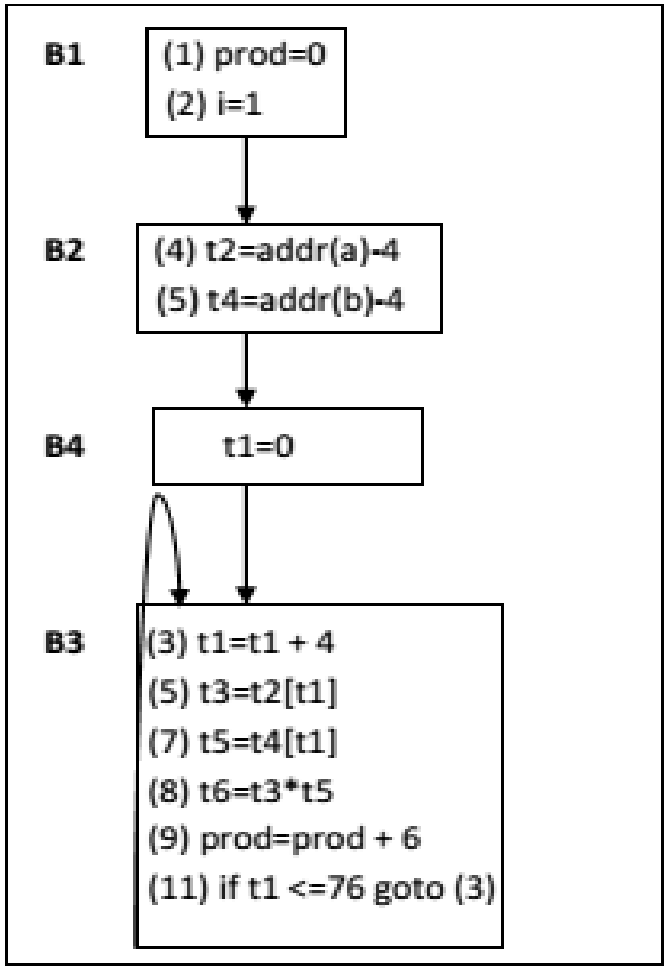
```
t=n-3;
while(i<t)
```

b. Induction Variable Elimination

- A variable is said to be Induction variable, if the value of the variable gets changed every time (incremented or decremented by constant).
- It is used to replace variable from inner loop.
- It can reduce the number of additions in a loop.
- It improves the code for both space and run time performance.



Before



After

c. Reduction in Strength

- Strength reduction is used to replace the expensive operation by the cheaper one on the target machine.
- Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.
- Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

- Example:

  L=strlen(S1 || S2) is more costly than

  L=strlen(S1) + strlen(S2)

Directed Acyclic Graphs (DAG)

- A Directed Acyclic Graph also called DAG is a directed graph that contains no cycle.
- A DAG is used to optimise basic blocks.
- A DAG is used to eliminate Common Sub-expression.
- DAG specifies how the value computed by each statement in a basic block is used in subsequent statements of the block.
- To apply transformation in a basic blocks, a DAG is constructed from 3 address code.

Applications of DAG

- 1. Determining the common Sub-expression
- 2. Determining which names are used inside the block & computed outside the block.
- 3. Determining which statements of the block could have their computed value outside the block.
- 4. Simplifying one list of Quadruples by eliminating Common Sub-expression.

Algorithm for the construction of DAG

- 1 (a) In a DAG, leaf node represent identifier,
  name or constants.
  (b)interior nodes represent operators.

- (2) While constructing DAG, a check is made to find if there is an existing node with the same children. A new node is created only when such a node does not exist. It helps to detect common sub-expression.

- (3) The assignment of the form x=y must not be preferred until and unless it is a result.

Example 1

- Construct DAG for the given expression

(a+b) * (a+b+c)

Sol: Three Address code for the expression is

t1= a + b

t2 = t1 + c

t3= t1 * t2

Example 2

1) t1=4*i
2) t2=addr(a)-4
3) t3=t2[t1]
4) t4=4*i
5) t5=addr(b)-4
6) t6=t5[t4]
7) t7=t3*t6
8) t8=prod + t7
9) prod=t8
10) t9=i+1
11) i=t9
12) if i<=20 goto (1)

Code Generation

- Code generation is the last phase in the compilation process.

Issues in the code generator

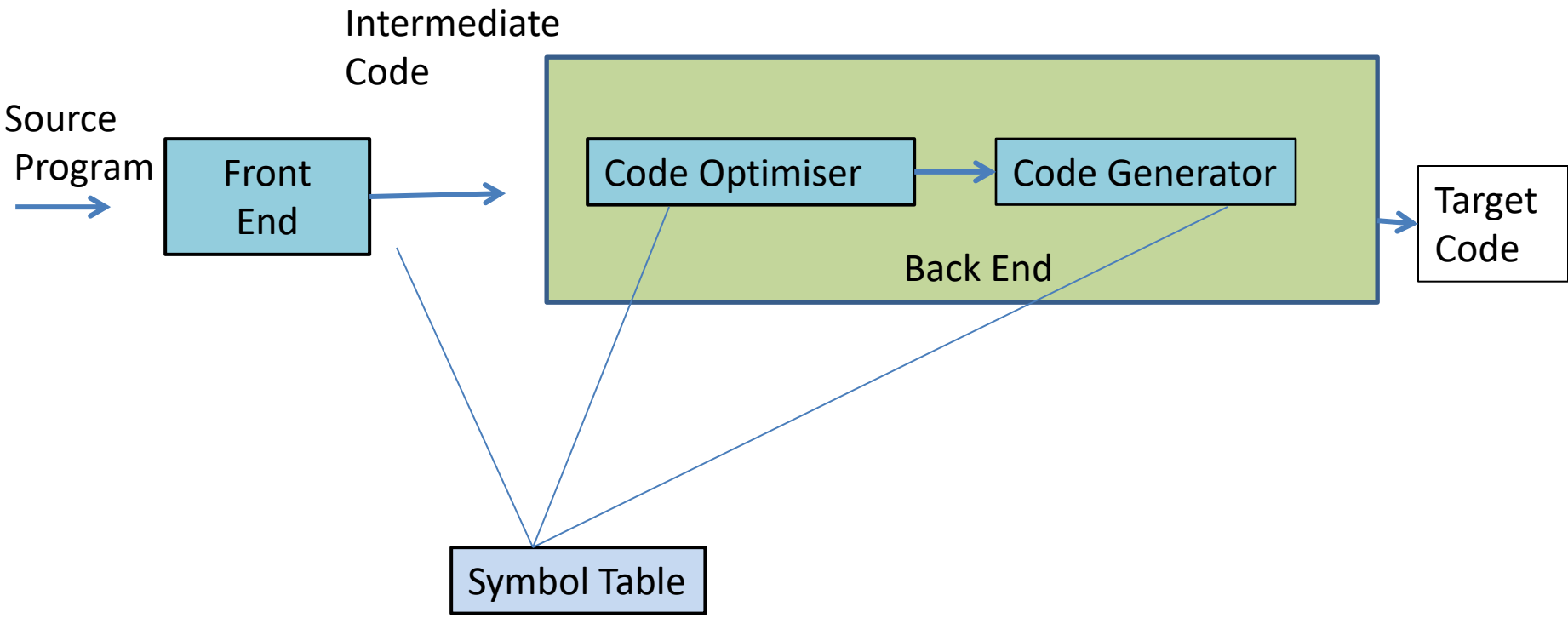


Figure: Role of Code Generator

- 1. Input to the Code Generator
- 2. Target Program
- 3. Memory Management
- 4. Instruction selection
- 5. Register allocation issues
- 6. Evaluation order
- 7. Generation of Correct code.

Target Machine

- Familiarity with the target machine and its instruction set are a prerequisite for designing a good code generator.
- Our target computer is:
  - Byte addressable
  - 4 bytes per word
  - N general purpose registers
  - It has address instructions
  - The addressing mode and cost of execution of the instructions.

- **Instruction Costs:** Each Instruction has a cost of 1 plus added costs for the source and destination

Cost of instruction = 1 + cost associated with the source and destination address mode.

This cost corresponds to the length (in words) of instruction.

A simple Code Generator

- The code generator strategy generates target code for a sequence of three address statement.
- The code generation algorithm uses descriptors to keep track of register contents and addresses for names.
  - Register Descriptor
  - Address Descriptor

A Code Generation Algorithm

- For each three address statements of the form

  x=y op z

  we perform the following action:

(1) Invoke a function *getreg to* determine the location *L,* where the result of computation *y op z* should be stored *'L'* will usually be a register, but it could also be a memory location.

(2) Consult the address descriptor for *'y'* to determine *'y'*, the current location of *'y'*. If the value of *'y'* is not already in *'L'*, generate the instruction *MOV y, L* to place a copy of *'y'* in *'L'.*

(3) Generate the instruction *op z, L* where *z* is a current locations of *z.* Update the address descriptor of *'x'* to indicate that it contains the value of *x* and remove *'x'* from all other register descriptors.

(4) if the current values of *'y'* and *'z'* have no next users and are in registers after the register descriptor to indicate that after execution of *x = y op z*, those registers no longer will contain *'y'* and/or *'z'*.

- Generate the target code for the following expression:

$$W = A - B + A - C + A - C$$

The TAC is given below:

$$T = A - B$$

$$U = A - C$$

$$V = T + U$$

$$W = V + U$$

Statement	Code Generated	Reg. Descriptor	Address Descriptor
T = A – B	MOV A,R_0 SUB R_0, B	R_0 has A R_0 has A – B = T	T is in R_0
U = A – C	MOV A,R_1 SUB R_1, C	R_1 has A R_1 has U = A – C	U is in R_1
V = T + U	ADD R_0 , R_1	R_0 has T R_1 has U	V is in R_0
W = V + U	ADD R_0, R_1 STORE R_0, W	R_0 has V R_1 has U	W is in R_0

Thank You