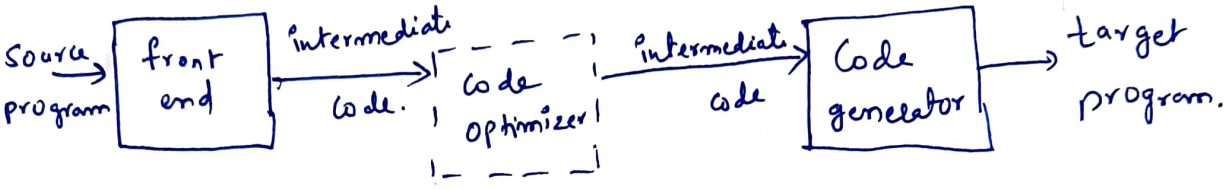
Unit-IV.

Chapter-2.

Code Generation.

Compiler that needs to produce efficient target code, includes an optimization phase before code generation. The code optimization and code generation phases of a compiler are known as back end.



A code generator has 3 main tasks.

1. instruction selection.
2. Register Allocation and assignment.
3. ~~Code~~ Instruction Ordering.

→ Instruction selection involves choosing appropriate target machine instructions to implement Intermediate Representation (IR) statements.

→ Register allocation and assignment involves deciding what values to keep in which register.

→ Instruction Ordering involves deciding the order to schedule the execution of instructions.

Issues in the Design of a Code Generator.

→ The most important criteria for code generator is to produce correct code. Given the premium on Correctness, designing a code generator can be easily implemented, tested and maintained

This Stores

1. Input to Code generator.

The i/p to code generator is IR of the source program produced by Source ~~program~~ Front end, along with info" in ~~Sou~~ Symbol table.

Various Choices for IR includes Three address representations such as quadruples, ~~triples~~, triples, indirect triples. And Virtual machine representations such as byte codes and stack machine code. Linear representations such as postfix notation, Graphical notations such as DAG's and Syntax Tree.

2. The Target program.

The most Common target machine architectures are RISC (Reduced Instruction Set Computer) and CISC (Complex instruction set Computer).

→ RISC machine has many registers, Three-address instructions, and simple addressing modes. and relatively simple instructions-set Architecture.

→ CISC has few registers, two address inst^n, a variety of addressing modes, and several register classes, variable length instructions, and instruction with side effects.

Instruction Selection.

Code generator maps, IR program into a code sequence that can be executed by the target machine.

The Complexity of mapping can be determined by

→ The level of the IR
→ The nature of instruction-set Architecture.
→. Desired quality of the generator code.

If the IR is high level, The code generator may translate each IR statement into sequence of small machine instructions.

Instruction speeds and machine idioms are also important factors. For each type of 3 address code, we can design target code as follows.

Consider the statement

$x = y + 3$, can converted to code sequence as follows.

```
LD   R0, y.   // R0 = y.
ADD  R0, R0, 3.  // R0 = R0 + 3.
ST   x, R0.   // x = R0.
```

This strategy produces redundant loads and stores. For eg consider three address statements

$$a = b + c$$
$$d = a + e.$$

which can be Translated to

```
LD   R0, b.
ADD  R0, R0, c
ST   a, R0.
LD   R0, a
ADD  R0, R0, e
ST   d, R0.
```

The fourth statement is redundant since it loads the value that has just been stored. A quality of generated code is determined by its size and speed.

Register Allocation.

A key problems in code generation is deciding what values to hold in what registers. Registers are fastest Computational units on target machine. But we do not have enough of them to hold all values. Values not held in registers must be reside in memory.

Instructions involving register operands are faster and shorter than those involving operands in memory.

The Use of registers is levided into two subproblems.

1. Register Allocation → During this we select the Variables that will reside in registers.
2. Register Assignment → During which we select Registees that has variables.

Evaluation Order.

The Order in which Computations are performed can affect efficiency of the target code. Some Computation orders requires fewer registers to hold intermediate Results than Others.

LD r_1, r_2 is a register-to-register copy, in which contents of r_2 are copied into contents of r_1.

2. Store Operation :- The instruction

ST x, r

→ Stores the value in register r, into location x.

3. Computation Operations :- These are of the form

Op dst, src1, src2

→ Where Op is Operator like ADD or SUB.

→ dst, src1, src2 are locations.

→ Operation is performed on values of src1, and src2, and result is placed in location dst.

For ex: SUB r_1, r_2, r_3 Compute $r_1 = r_2 - r_3$

→ Unary Operators takes only one Operand, They do not have src_2.

3. Unconditional Jumps :- The instruction

BR L

→ Causes control to transfer to instruction with label L (BR stands for branch).

* Conditional Jumps :- These are of the form

B cond r, L

→ Where r is register, L is label, and cond is test on values in r.

→ Ex:- BLTZ r, L.

→ Causes Jump to Label L, if the value in r is less than zero, if not control transfers to next instruction.

Addressing Modes.

* In instruction a location, can be a variable name x, referring to the memory location reserved for x.

→ A location can also be an indexed address of the form a(r), where a is variable and r is a register. The value of a is added to the value in register r.

Ex:- LD R1, a(R2), has the effect

R1 = contents (a + content (R2))

⇒ Memory location can be an integer indexed by a Register. Ex:-

LD R1, 100(R2)

$R1 = Contents(100 + Contents(R_2))$

→ Indirect addressing modes :-

→ *r means, memory location found in location represented as contentants of register r.

→ *100(r) → memory location found by adding 100 to the Contents of r.

Ex:- LD R1, *100 (R2) has the effect.

$R1 = Contents(Contents(100 + Contents(R_2)))$

→ Immediate constant addressing Mode.

The constant is prefixed by #.

The instruction LD R1, #100, loads.

100 in R1

ADD R1, R1, #100.

Adds 100 into register R1.

Comments at the end of instruction are preceded by //.

Example:- The three-address statement $x = y - z$ can be implemented by the machine instructions

```
LD   R1, y.    // R1 = y
LD   R2, z     // R2 = z
SUB  R1, R1, R2 // R1 = R1 - R2
ST   x, R1     // x = R1
```

⇒ Suppose a is an array, whose elements are 8 bytes, indexed starting at 0. The 3 address instruction b = a[i] in machine inst^n can be given as

b = a[i]

```
LD   R1, i.      // R1 = i.
MUL  R1, R1*8    // R1 = R1 * 8
LD   R2, a(R1)   // R2 = Contents(a + Contents(R1))
ST   b, R2       // b = R2
```

Similarly a[j] = c, in machine instruction can be given as

```
LD   R1, c       // R1 = c
LD   R2, j.      // R2 = j
MUL  R2, R2, 8   // R2 = R2 * 8
ST   a(R2), R1   // Contents(a + content(R2)) = R1
```

⇒ X = *P, can be given as

```
LD = R1, P.      // R1 = P
LD = R2, 0(R1)   // R2 = Content(0 + Content(R1))
ST   X, R2       // X = R2
```

⇒ if X < y goto L, can be given as.

```
LD  R1, X.   // R1 = X
LD  R2, Y.   // R2 = X
SUB R1, R1, R2 // R1 = R1 - R2
```

Addresses in the Target Code.

The names in Intermediate Representation (IR) can be converted to addresses in target code by using code generation for simple procedure calls, and returns using static and stack Allocation.

Each Executing programs logical Address space has partitioned into four code and data areas.

→ A statically determined code Area, which holds target code. The size can be determined at compile time.

→ A statically determined static area, which holds global constants and other data generated by the compiler. The size can be known at compile time.

→ A Dynamically managed area heap, which holds objects that are allocated and freed during program execution. The size of heap can't be determined during compile time.

→ A Dynamically managed area stack for holding activation records as they are created and destroyed during procedure calls and returns. Stack size can't be determined at compile time.

Static Allocation.

The Three address statements in procedure are

→ call callee.
→ return.
→ halt.
→ action

The code for implement simple case, Static Allocation is as follows.

The call callee There address statement can be implemented as two target-machine instructions as follows.

```
ST    Callee.staticArea , #here+20.
BR    Callee.codeArea.
```

→ ST stores return Address at the begining of activation record for callee, BR transfers the control to callee

→ Callee.staticArea is a constant text gives begining address of ~~callee~~ activation record for callee.

→ Callee.codeArea is the address of first instruction of called procedure callee.

→ The Opeand #here+20, in ST instruction is literal

return address, which is the Address of next instruction which follows BR.

Basic Blocks and Flow Graphs.

The intermediate code can be represented as a graph. This helps in register allocation

The Graph representation is constructed as follows.

1. partition the intermediate code into basic blocks, which are consequtive sequences of Three-Address instructions. With following properties.

   a). Flow of Control enters into basic block Through the first instruction in the block. There are no Jumps in the middle of the block.

   b). Control will leave the block without halting or branching.

2. Basic Blocks becomes nodes of a flow graph, whose edges indicates next Blocks.

Basic Blocks.

The steps in constructing Basic Blocks are as follows.

→ identify leaders, first statements of basic blocks.

The rules for finding leaders are.

1. The first Three Address instruction in the intermediate code is a leader.

2. Any instruction, which is target of Conditional or unConditional Jump is a leader.

3. Any instruction, That follows immediately a Conditional or unconditional jump is a leader.

Then for each leader, its basic block Consists of itself and all other instructions up to but not including next leader.

For exp Consider The intermediate code.

1) i=1
2) j=1
3) t1=10*i.
4) t2=t1+j
5) t3=8*t2
6) t4=t3-88.
7) a[t4]=0.0
8) j=j+1
9) if j<=10 goto (3).
10) i=i+1
11) if i<=10 goto (2).
12) i=1
13) t5=i-1
14) t6=88*t5
15) a[t6]=1.0.
16) i=i+1
17) if i<=10 goto (13)

→ In the above code, instruction 1 is leader by rule(1).

→ To find other leaders we first need to find Jumps. There are 3 Jumps at lines 9, 11, ~~13~~ and 17, By rule(2) The targets of these Jumps are leaders. Those are instructions ~~4, 11, 17~~ 3, 2, and 13. By rule (3), The instructions immediately after Jumps 10 and 12.

→ Finally The leaders are 1, 2, 3, 10, 12 and 13. And The Blocks are

B1
```
i=1
```

B2
```
j=1
```

B3
```
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
a[t4] = 0.0
j = j + 1
if j <= 10 goto B3
```

B4
```
i = i + 1
if i <= 10 goto B2
```

B5
```
i = 1
```

B6
```
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto B6
```

Flow Graphs.

→ Once an intermediate code program is partitioned into basic blocks, we represent flow of control between them by a flow graph.

→ The Nodes of flow graph are the basic blocks.

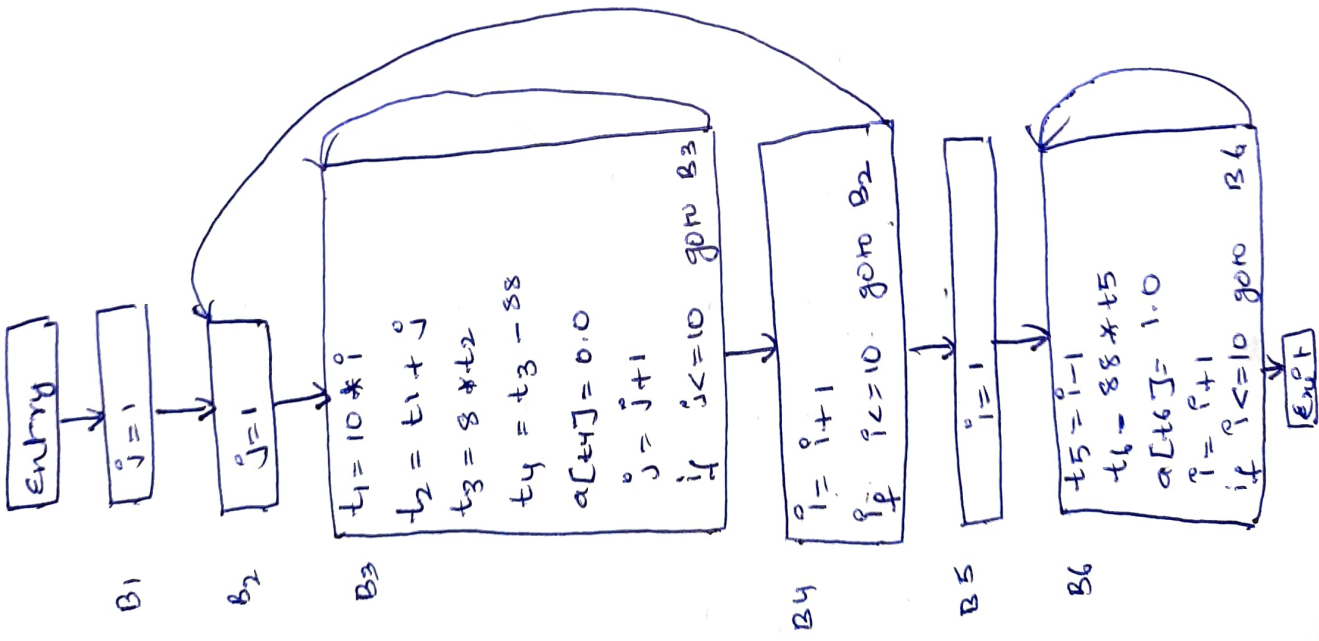
→ There is an edge from Block B to C, if first instruction in block C, immediately follows last instruction of Block B.

→ An Edge can be Added in two ways

→ There is a conditional or unconditional jump from the end of B to the begining of C.

→ C immediately follows B, in the Original Order of Three Address instruction.

→ we Say B as predecessor of and C as Successor of B.



Optimization of Basic Blocks

We can improve running time of code by performing local optimization within each basic block, and global optimization which shows how information flows among basic blocks of a program.

The Dag Representation of Basic Blocks.

local optimization within a Block is achieved by transforming a basic block into DAG (Directed Acylic Graph).

We construct DAG for a basic block as follows.

1. There is a node for each initial values of Variables
2. There is a node N fore each statement S within the block. The children of N are Statements prior to S, Operands used by S.
3. Node N is labeled by Operator applied to S. and also to N, The list of variables which it is the last definition within Block.
4. Some nodes are used as output nodes. Their values are computed at the exit of Block and used in another Block of flow graph.

→ The DAG has several code improving transformations as follows.

→ We can eliminate local common subexpressions. i.e; instructions that computes a value that has already computed.

→ We can eliminate dead code, instruction that computes a value that is never used.

→ We can reorder the statements that do not depend on one another.

→ We can apply algebraic laws to reorder the operands of three address instructions to simplify the computations.

Finding local common subexpressions.

Common sub expression can be detected by noticing, as a new node M is about to be added, and there exists an old node N, with same children (same order) and operator. N computes same value as M. Then M is and N are common and M can be removed.

or Conditional Jumps to

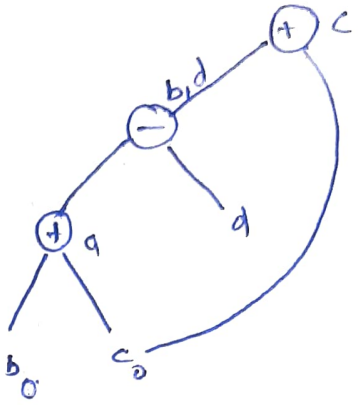
Ex1:- DAG for the BLOCK

$a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$

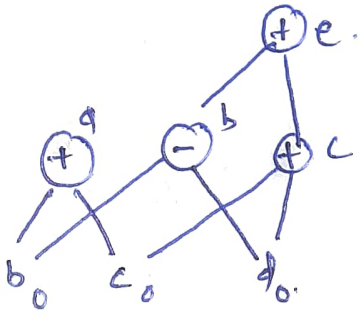


Ex:- DAG for $a = b + c$

$b = b - d$

$c = c + d$

$e = b + c$



Dead code elimination.

We can delete from DAG any root node with no ancestors that has no live variable attached

Peehole Optimization.

→ Peehole Optimization is a technique for locally improving the target code.

→ Peehole Optimization is done by examining a sliding window of target instructions (called as Peehole), and replacing the instructions within the Peehole by a shorter and faster sequence of inst^{n}.

→ Peehole Optimization can be used directly after the intermediate code generation to improve intermediate representation

→ Peehole is a small sliding window on a program

→ The following are Characteristics of Peehole Optimization.

1. Redundant - inst^{n} elimination.
2. Flow of control optimization.
3. Algebraic simplifications.
4. Use of machine idioms.

1. Eliminating Redundant Loads and Stores.

Concider the instruction ~~safe~~ sequence.

```
LD   R0, a
ST   a, R0
```

We can eliminate Store instruction, because whenever the instruction is executed, The first instruction will ensure that value of a has already been loaded into Register R0.

2. Eliminating unreachable code.

An unlabeled instruction immediately following an unconditional Jump may be removed. This operation can be repeated to eliminate a sequence of instructions.

consider the code sequence

```
if debug != 1 goto L2
print debugging information
L2:
```

if debug is set to 0 at the begining of the program, The code becomes

```
if 0 != 1 goto L2
print debugging information
L2:
```

Now first statement always evaluates to true, So statement can be replaced by goto L2. Then all statements that "prints debugging info" are unreachable and can be eliminated One at a time.

Flow-of-Control Optimizations.

Simple intermediate code generation algorithm Produce Jump to Jumps, Jumps to Conditional Jumps or Conditional Jumps to Jumps.

goto L1
...
L1: goto L2

Can be replaced by

goto L2
...
L1: goto L2

→ now no jumps to L1, Then it may be possible to eliminate L1: goto L2.

Algebraic Simplification and Reduction in Strength.

Algebraic identities used to simplify DAG's. These can also be used to by peephole optimizer. to eliminate statements in Three-address format Such as

$$X = X + 0$$

or

$$X = X * 1$$

Use of machine idioms.

Target machine may have hardware instructions to implement certain specific operations efficiently. So machines may have auto-increments and auto-decrement modes. Use of these modes improves quality of code.