

UNIT-VIII

CODE GENERATION

Machine dependent code generation, object code forms, generic code generation algorithm, Register allocation and assignment. Using DAG representation of Block.

- The final phase in our compiler model is the code generator. It takes intermediate representation (IR) produced by the front end of the compiler as input along with relevant symbol table information and produces semantically equivalent target program as output.
- The following diagram shows the position of code generator.



MACHINE DEPENDENT CODE GENERATION

- A code generator has three primary tasks:
 - Instruction selection
 - Register allocations and assignment
 - Instruction ordering
- Instruction selection involves choosing appropriate target machine instructions to implement the intermediate representation statements.
- Register allocation and assignment involves in deciding what values to keep in which registers.
- Instruction ordering involves in deciding what order to schedule the execution of instructions.
- The code generators use certain algorithms to translate the intermediate representation into a sequence of target language instructions for simple register machine.
- Many code generators partition intermediate representation instructions into “basic blocks” which consists of sequences of instructions that are always executed together.

- There are several Issues in the design of code generator:

- ❖ **Input to the Code Generator**

- The input to the code generator is the intermediate representation of the source program produced by the front end along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the intermediate representation.
- There are so many choices for intermediate representation (IR) which include three-address representations such as quadruples, triples, indirect triples, virtual machine representation such as byte code and stack-machine code, linear representation such as postfix notation and graphical representation such as syntax trees and DAG.

- ❖ **The Target Program**

- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code.
- The most common target machine architectures are RISC, CISC and stack based.
RISC → Reduced Instruction Set Computer
CISC → Complex Instruction Set Computer
- RISC consists of many registers, three address instructions, simple addressing mode and a relatively simple instruction-set architecture.
- CISC consists of a few registers, two address instructions, a variety of addressing modes, several register classes, variable-length instructions.

- In stack based machine, operations are done by pushing operands on to a stack and then performing the operations on the operands at the top of the stack. To achieve high performance, the top of the stack is typically kept in registers.
- But stack based architectures are almost disappeared because of it's limitation and required too many swap and copy operations.
- However stack based architectures were revived(re-energized) with introduction of the JVM. The JVM is a software interpreter for java byte codes, and also it is an intermediate language produced by java compilers.
- To achieve high performance, JIT compilers are used which can translate the byte codes during run time to the native hardware instruction set of the target machine.

❖ **Register Allocation**

- A key problem in code generation is deciding what values to hold in what registers.
- Registers are the fastest computational unit on the target machine but we usually do not have enough of them to hold all values.
- Values that can not be able to hold in the registers need to reside in memory.
- The use of registers is often sub divided into two sub problems 1) register allocation and 2) register assignment.

❖ **Evaluation order**

- The order in which computations are performed can affect the efficiency of the target code. Because, as we know that some computations orders requires few registers to hold the intermediate results than the other. However picking the best order is some what difficult.

❖ **Instruction selection**

- The uniformity and completeness of instruction set is an important factor for the code generator. The selection of instructions depends upon the instruction set of the target machine.
- The speed of instruction and machine idioms are two important factors in selection of instructions.

MACHINE DEPENDENT CODE OPTIMIZATIONS

- Machine-dependent optimization uses information about the limits and special features of the target machine to produce code which is shorter or which executes more quickly on the machine.
- The code produced by the compiler should take advantage of the special features of the target machine. For example, consider code intended for machines of the PDP-11 family.
- These computers have auto increment and auto decrement modes for instructions.

- When an instruction is given in the auto increment mode, the contents of the register are incremented after being used. The register is incremented by one for byte instructions and by two for word instructions.
- The use of instructions in these modes reduces the code necessary for pushing and popping stacks.
- The PDP-11 computers also have machine-level instructions to increment (INC), or to decrement (DEC), by one, values stored in memory.
- Whenever possible, the INC and DEC operations should be used instead of creating a constant with value 1 and adding or subtracting this constant from the value stored in memory.
- The PDP-11 machines have left- and right-shift operations. Shifting the bits one position to the left is equivalent to multiplying by 2. Since shifting is faster than multiplication or division, more efficient code is generated if multiplication and division by multiples of 2 are implemented with shift operations.

OBJECT CODE FORMS

- The output of code generation is an object code or machine code. Which is normally classified into different forms.
 1. Absolute Code
 2. Relocatable machine Code
 3. Assembler Code

- **Absolute Code:-** Producing an absolute machine language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed.
- Programs can be compiled and executed quickly.
- **Relocatable Machine Code:-** Producing a relocatable machine language program (often called as object module) as output allows sub programs to be compiled separately.
- For example a set of relocatable object modules can be linked together and loaded for execution by linking loader.
- If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program segments.
- **Assembler Code:-** Producing an assembly-language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use macro facilities of the assembler to help in generation of code. But generating assembler code as an output makes code generation process slower because of it needs assembling, linking and loading.

GENERIC CODE GENERATION ALGORITHM

- A simple code generation algorithm is a one that generates code for a single basic block.
- It considers each three-address instruction in turn and keeps track of what values are in what registers so it can avoid generating unnecessary loads and stores.
- One of the primary issues during code generation is how to use registers effectively.
- There are four principle uses of registers:
 - ⊕ In most machine architectures some or all of the operands of an operation must be in registers in order to perform the operation.
 - ⊕ Registers make good temporaries – i.e. places to hold the result of sub expression while a larger expression is being evaluated.
 - ⊕ Registers are used to hold values that are computed in one basic block and used in other blocks.
 - ⊕ Registers are often used to help with run-time storage management, for example registers are used to manage the run-time stack.
- Let us assume that some set of registers is available to hold the values that are used within the block.
- Typically this set of registers does not include all the registers of the machine since some registers are reserved for global variables and managing the stack.

- But our code generation algorithm considers each three address instruction in turn and decides what loads are necessary to get the needed operands into registers.
- After generating the loads, it generates the operation itself. And if there is a need to store the result into a memory location, it also generates that store.
- In order to make needed decisions we require a data structure that tells us what program variables currently have their value in a register and in which register(s) if so.
- The desired data structure has the following descriptors:
- For each available register, a register descriptor keeps track of the variable names whose current value is in that register.
- For each program variable an address descriptor keeps track of the location or locations where the current value of that variable can be found. Where the location may be a stack location or a register or a memory address.
- An essential part of the algorithm is the function `getReg(I)` which selects registers for each memory location associated with the three address instruction `I`.
- The function `getReg` has access to the register and address descriptors for all the variables of the basic block.

Machine Instructions for operations

- For a three address instruction such as $x = y + z$, do the following:
 1. Use `getReg(x = y + z)` to select registers for `x`, `y` and `z`. let these registers are `Rx`, `Ry`, and `Rz`.

1. If y is not in R_y then issue an instruction $LD R_y, y'$, where y' is one of the memory locations for y .
 2. Similarly if z is not in R_z , issue an instruction $LD R_z, z'$, where z' is a location for z .
 3. Issue the instruction $ADD R_x, R_y, R_z$.
- In the above three address instruction $x = y + z$ we shall treat $+$ as a generic operator and ADD as the equivalent machine instruction.
 - Thus when we implement the operation, the value of y must be in the second register and z must be the third register in the ADD instruction.

Managing Register and Address Descriptors

- As the code generation algorithm issues load, store and other machine instructions, it needs to update the register and address descriptors. The rules are as follows:
1. For the instruction $LD R, x$
 - a) Change the register descriptor for register R so it holds only x .
 - b) Change the address descriptor for x by adding register R as an additional location.
 2. For the instruction $ST x, R$ change the address descriptor for x to include its own memory location.
 3. For an operation such as $ADD R_x, R_y, R_z$ implementing a three address instruction $x = y + z$.
 - a) Change the register descriptor for R_x so that it holds only x .

- b) Change the address descriptor for x so that it's only location is Rx.
- c) Remove Rx from the address descriptor of any variable other than x.

REGISTER ALLOCATION AND ASSIGNMENT

- Register operands are faster and shorter than memory operands, which means that proper use of registers help in generating the good code.
- There are certain strategies adopted by the compiler for register allocation and assignment.
- The most commonly used strategy to register allocation and assignment is to assign specific values to specific registers.
- Some of the strategies used in register allocation and assignment are:

Global Register Allocation

- While generating the code the registers are used to hold the values for the duration of single block and All the live variables are stored at the end of each block.
- For the variables that are used again and again (consistently) , we can allocate specific set of registers. Hence allocation of variables to specific registers that is consistent across the block boundaries is called as global register allocation.
- Following are the strategies adopted while doing the global register allocation.
 1. The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop.

2. Another strategy is to assign some fixed number of global registers to hold the most active values in each inner loop.
3. The registers that are not already allocated may be used to hold values local to one block.

Register Assignment for Outer Loop

- Consider that there are two loops L1 is outer loop and L2 is an inner loop. And allocation of variable a is to be done to some register.
- Following strategies should be adopted for register assignment for outer loop.
 1. If the variable a is allocated in loop L2 then it should not be allocated in between L1 and L2.
 2. If the variable a is allocated in L1 and it is not allocated in L2 then store a at the beginning to L2 and load the variable a while leaving L2.
 3. If the variable a is allocated in L2 and not in L1 then load a at the beginning of L2 and store a on exit from L2.

Usage Count

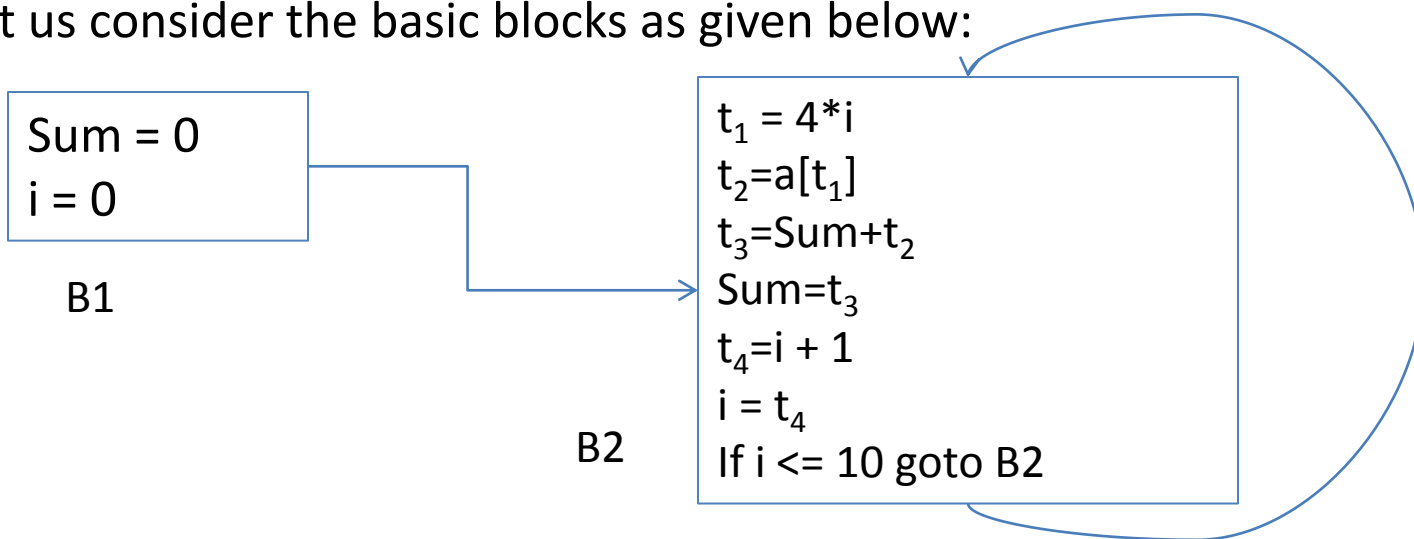
- The usage count is the count for the use of some variable x in some register used in any basic block.
- The usage count gives the idea about how many units of cost(time) can be saved by selecting a specific variable for global register allocation.
- The approximate formula for usage count for the loop L in some basic block B can be given as
$$\sum_{\text{Block B in L}} (\text{use}(x, B) + 2 * \text{live}(x, B))$$

Block B in L

- Where $use(x, B)$ specifies the number of times x used in block B , and $live(x, B)=1$ if x is live on exit from B otherwise $live(x, B)=0$.

DAG Representation of Block

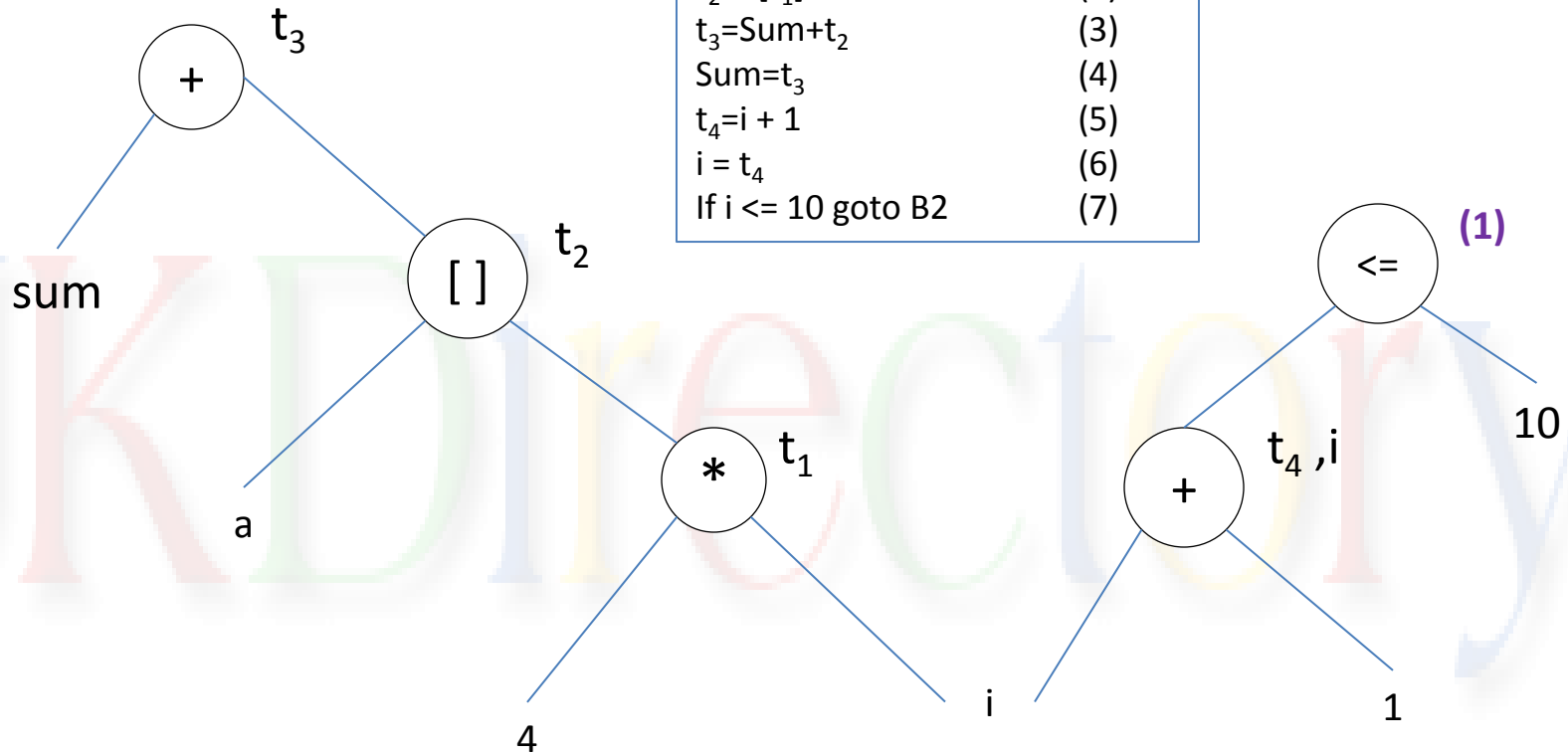
- The **D**irected **A**cyclic **G**raph is used to apply transformations on the basic block. To apply transformations on basic block using DAG, it is constructed from three address statement.
- A DAG can be constructed for the following type of labels on nodes:
 1. Leaf nodes are labeled by identifiers or variable names or constants.
 2. Interior nodes store operator values.
- The DAG and flow graphs are two different pictorial representation. But each node of the flow graph can be represented by DAG since each node of the flow graph is a basic block.
- Let us consider the basic blocks as given below:



- The DAG representation for basic block B2 is given as below:

- Consider the numbering as \rightarrow

$t_1 = 4 * i$	(1)
$t_2 = a[t_1]$	(2)
$t_3 = \text{Sum} + t_2$	(3)
$\text{Sum} = t_3$	(4)
$t_4 = i + 1$	(5)
$i = t_4$	(6)
If $i \leq 10$ goto B2	(7)



- Construction of DAG for the basic block is done by applying the following steps:
 - If y is undefined in the three address statement $x = y \text{ op } z$ then create y similarly create z if undefined.
 - Create a node (op) whose left child is node (y) and right child is node (z). Also check for common subexpressions if any.
 - Check for a node labeled op , which has a child node (y), such that node n will be node (y)
 - Delete x from the list of identifiers for node (x). And append x to the list of attached identifiers for node n found in 2.

Applications of DAG

- DAG is used in determining the common subexpressions.
- DAG is used in determining which names are used inside the block and computed outside the block.
- DAG is used in determining which statements of the block could have their computed value outside the block.
- DAG is used in simplifying the list of quadruples by eliminating the common subexpression and not performing the assignment of the form $x=y$ unless it is necessary.

JKD **T** **H** **E** Directory!
hink +ve work ard ducate,
E **N** **D**
ncourage yourself but ever iscourage yourself or others