Module 6 – Code Generation

Issues in Code Generation

Issues in Code Generation

- **Issues in Code Generation are:**
    1. Input to code generator
    2. Target program
    3. Memory management
    4. Instruction selection
    5. Register allocation
    6. Choice of evaluation
    7. Approaches to code generation

Input to code generator

- Input to the code generator consists of the intermediate representation of the source program.

- Types of intermediate language are:
    1. Postfix notation
    2. Quadruples
    3. Syntax trees or DAGs

- The detection of semantic error should be done before submitting the input to the code generator.

- The code generation phase requires complete error free intermediate code as an input.

Target program

- The output may be in form of:

    1. Absolute machine language: Absolute machine language program can be placed in a memory location and immediately execute.

    2. Relocatable machine language: The subroutine can be compiled separately. A set of relocatable object modules can be linked together and loaded for execution.

    3. Assembly language: Producing an assembly language program as output makes the process of code generation easier, then assembler is require to convert code in binary form.

Memory management

- Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator.

- We assume that a name in a three-address statement refers to a symbol table entry for the name.

- From the symbol table information, a relative address can be determined for the name in a data area.

Instruction selection

- Example: the sequence of statements

    a := b + c

    d := a + e

- would be translated into

    MOV   b, R0

    ADD   c, R0

    MOV   R0, a

    MOV   a, R0

    ADD   e, R0

    MOV   R0, d

- Here the fourth statement is redundant, so we can eliminate that statement.

Register allocation

- The use of registers is often subdivided into two sub problems:

- During register allocation, we select the set of variables that will reside in registers at a point in the program.

- During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

- Finding an optimal assignment of registers to variables is difficult, even with single register value.

- Mathematically the problem is NP-complete.

Choice of evaluation

- The order in which computations are performed can affect the efficiency of the target code.

- Some computation orders require fewer registers to hold intermediate results than others.

- Picking a best order is another difficult, NP-complete problem.

Approaches to code generation

- The most important criterion for a code generator is that it produces correct code.

- The design of code generator should be in such a way so it can be implemented, tested, and maintained easily.

Target Machine

Target machine

- We will assume our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps.

- The underlying computer is a byte-addressable machine with n general-purpose registers, $R_0, R_1, \ldots, Rn$

- The two address instruction of the form: *op source, destination*

- It has following opcodes:

  MOV (move source to destination)

  ADD (add source to destination)

  SUB (subtract source to destination)

Instruction Cost

- The address modes together with the assembly language forms and associated cost as follows:

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k +contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1
Immediate / Literal mode	#C	-NA-	1

- The instruction cost can be computed as one plus cost associated with the source and destination addressing modes given by "extra cost".

Instruction Cost

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k +contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1
Immediate / Literal mode	#C	-NA-	1

- Calculate cost for following:

MOV B,R0 ADD C,R0 MOV R0,A	

Instruction Cost

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k +contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

- Calculate cost for following:

MOV *R1 ,*R0 MOV *R1 ,*R0	

Next Use Information

Computing Next Uses

- The next-use information is a collection of all the names that are useful for next subsequent statement in a block.

- The **use of a name** is defined as follows,

- Consider a statement,

$$x := i$$

$$j := x \text{ op } y$$

- That means the statement j uses value of x.

- The next-use information can be collected by making the backward scan of the programming code in that specific block.

Storage for Temporary Names

- For the distinct names each time a temporary is needed. And each time a space gets allocated for each temporary.

- To have optimization in the process of code generation we pack two temporaries into the same location if they are not live simultaneously.

- Consider three address code as,

t1=a*a
t2=a*b
t3=4*t2
t4=t1+t3
t5=b*b
t6=t4+t5

Register and Address Descriptors

- The code generator algorithm uses descriptors to keep track of register contents and addresses for names.

- **Address descriptor** stores the location where the current value of the name can be found at run time. The information about locations can be stored in the symbol table and is used to access the variables.

- **Register descriptor** is used to keep track of what is currently in each register. The register descriptor shows that initially all the registers are empty. As the generation for the block progresses the registers will hold the values of computation.

Register Allocation & Assignment

Register Allocation & Assignment

- Efficient utilization of registers is important in generating good code.
- There are four strategies for deciding what values in a program should reside in a registers and which register each value should reside.
- Strategies are:
  1. Global Register Allocation
  2. Usage Count
  3. Register assignment for outer loop
  4. Register allocation for graph coloring

Global Register Allocation

- Global register allocation strategies are:

- The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop.

- Another strategy is to assign some fixed number of global registers to hold the most active values in each inner loop.

- The registers are not already allocated may be used to hold values local to one block.

- In certain languages like C or Bliss programmer can do the register allocation by using register declaration to keep certain values in register for the duration of the procedure.

- Example:

```
{
    register int  x;
}
```

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 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,

$$\sum_{block\ B\ in\ L} (use(x, B) + 2 * live(x, B))$$

- Where $use(x, B)$ is number of times x used in block B prior to any definition of x

- $live(x, B) = 1$ if x is live on exit from B; otherwise $live(x) = 0$.

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.

Loop L_1

Loop L_2 $\}$ L_1-L_2
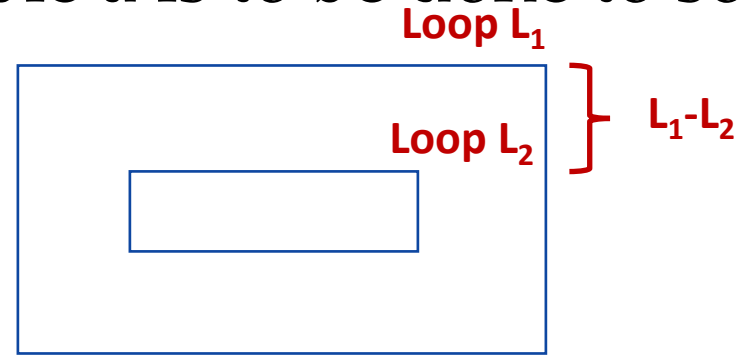
- Following criteria should be adopted for register assignment for outer loop,
- If a is allocated in loop $L2$ then it should not be allocated in $L1 - L2$.
- If a is allocated in $L1$ and it is not allocated in $L2$ then store a on entrance to $L2$ and load a while leaving $L2$.
- If a is allocated in $L2$ and not in $L1$ then load a on entrance of $L2$ and store a on exit from $L2$.

Register allocation for graph coloring

- The graph coloring works in two passes. The working is as given below:

- In the first pass the specific machine instruction is selected for register allocation. For each variable a symbolic register is allocated.

- In the second pass the register inference graph is prepared.

- In register inference graph each node is a symbolic registers and an edge connects two nodes where one is live at a point where other is defined.

- Then a graph coloring technique is applied for this register inference graph using k-color.

- The k-colors can be assumed to be number of assignable registers.

- In graph coloring technique no two adjacent nodes can have same color. Hence in register inference graph using such graph coloring principle each node is assigned the symbolic registers so that no two symbolic registers can interfere with each other with assigned physical registers.