

ISSUES IN CODE GENERATION

The most important criterion for a code generator is that it produces correct code. The following issues arise during the code generation phase:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

Input to code generator:

The input to the code generation consists of the intermediate representation of the source program produced by front end, together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be:

- a. Linear representation such as postfix notation
- b. Three address representation such as quadruples, triples, indirect triples
- c. Virtual machine representation such as byte code and stack machine code
- d. Graphical representations such as syntax trees and DAG's.

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 (reduced instruction set computer), CISC (complex instruction set computer), and stack based.

- A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
- In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

In a stack-based machine, operations are done by pushing operands onto 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. Stack-based machines almost disappeared because the stack organization was too limiting and required too many swap and copy operations.

- 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.

- Producing a relocatable machine-language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

Memory management:

Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator. Labels in three-address statements have to be converted to addresses of instructions.

For example,

`j : goto i` generates jump instruction as follows :

- if $i < j$, a backward jump instruction with target address equal to location of code for quadruple i is generated.
- if $i > j$, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j . When i is processed, the machine locations for all instructions that forward jumps to i are filled.

Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size
- For example, every three-address statement of the form $x = y + z$, where x , y , and z are statically allocated, can be translated into the code sequence

```
LD  R0, y      // R0 = y      (load y into register R0)
ADD R0, R0, z  // R0 = R0 + z (add z to R0)
ST  x, R0      // x = R0      (store R0 into x)
```

- This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

```
a = b + c
d = a + e
```

Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two sub problems:
 - Register allocation - the set of variables that will reside in registers at a point in the program is selected
 - Register assignment - the specific register that a variable will reside in is picked

- Finding an optimal assignment of registers to variables is difficult

Example 1: Certain machines require register-pairs (an even and next odd numbered register) for some operands and results. For example, for integer multiplication and integer division involve register pairs.

The multiplication instruction is of the form

M x, y

- where x, the multiplicand, is the even register of an even/odd register pair
- the multiplier y, is the odd register.
- The product occupies the entire even/odd register pair.

The division instruction is of the form

D x, y

- Where, the dividend occupies an even/odd register pair whose even register is x;
- the divisor is y.
- After division,
 - even register holds the remainder
 - the odd register the quotient.

Evaluation order:

- The order in which the computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.

Example

