Unit 5

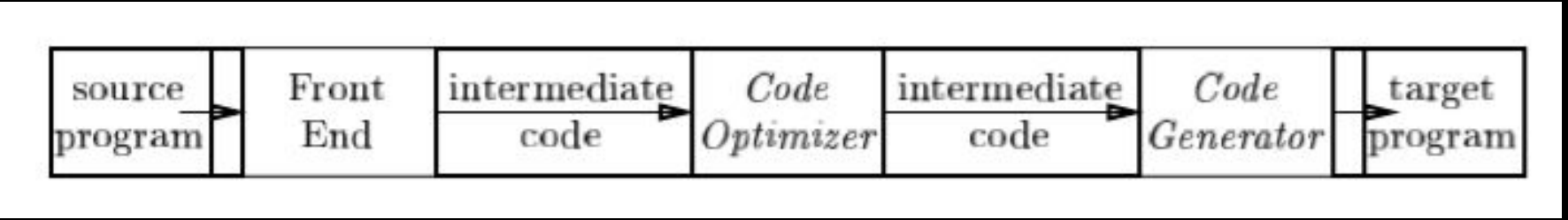
Code Generation

SINI ANNA ALEX

5/7/2020
10:06 AM

Code Generation

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



- The code optimization and code-generation phases of a compiler, often referred to as the back end, may make multiple passes over the IR before generating the target program.

Task of a code Generator

- Target code should be of high quality
  - execution time or space or energy
- Code generator itself should run efficiently.
- A code generator has three primary tasks:
  - Instruction selection,
  - Register allocation and assignment,
  - and Instruction ordering.
- **Instruction selection** involves choosing appropriate target-machine instructions to implement the IR statements.
- **Register allocation and assignment** involves deciding what values to keep in which registers.
- **Instruction ordering** involves deciding in what order to schedule the execution of instructions.

Code Generator in Reality

- The problem of generating an optimal target program is undecidable.
- Several subproblems are NP-Hard (such as register allocation).
- Need to depend upon
  - Approximation algorithms
  - Heuristics
  - Conservative estimates

Issues in the design of a code generator

- Code generation depends on the intermediate representation, the target language, and the run-time system, tasks such as instruction selection, register allocation and assignment, and instruction ordering.

- Main issues involving in the design of a code generator are
  - 1. Input to the code generator
  - 2. The Target Program
  - 3. Instruction Selection
  - 4. Register allocation
  - 5. Evaluation Order

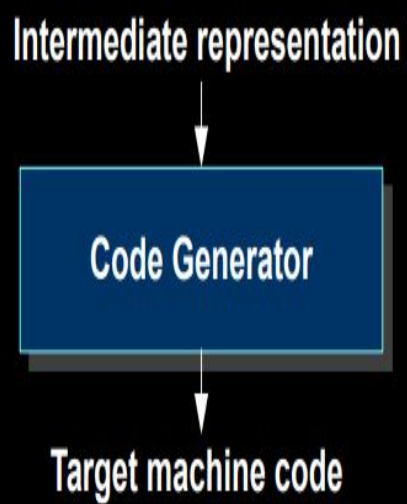
Input to the code generator

- 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 IR.
- The choices for the IR include three-address representations such as **quadruples, triples, indirect triples;** virtual machine representations such as **bytecodes** and **stack-machine code**; linear representations such as **postfix notation**; and graphical representations such as **syntax trees and DAG's.**
- Front end has scanned, parsed, and translated the source program into a relatively low-level IR, so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate, such as integers and floating-point numbers.

The Target Program

- Common target-machine architectures are **RISC** (reduced instruction set computer), **CISC** (complex instruction set computer), and **stack based**.
- 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.
- 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.
- Here we use a very simple RISC-like computer as our target machine.

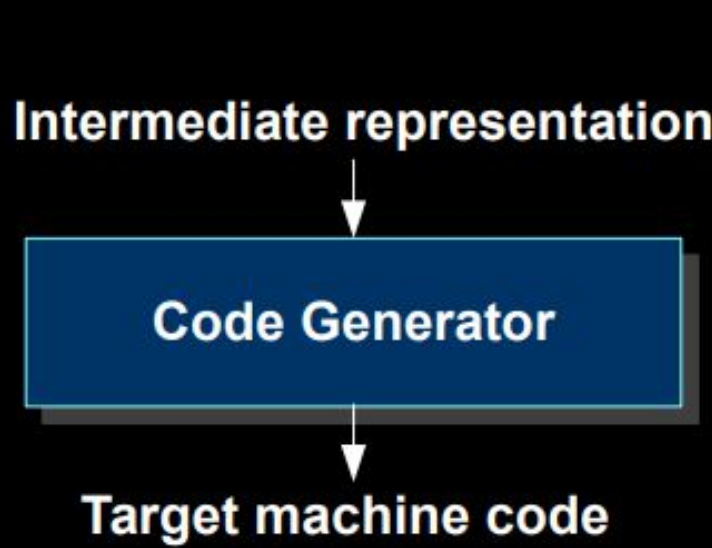
Input + Output



- **3AC** (Quadruples / Triples / Indirect triples)
- **VM** instructions (bytecodes / stack machine codes)
- **Linear** representations (postfix)
- **Graphical** representation (syntax trees / DAGs)

- **RISC** (many registers, 3AC, simple addressing modes, simple ISA)
- **CISC** (few registers, 2AC, variety of addressing modes, several register classes, variable length instructions, instructions with side-effects)
- **Stack** machine (push / pop, stack top uses registers, used in JVM, JIT compilation)

IR and Target Code




```
R0 = y
R0 = R0 + z
x = R0
```

```
LD R0, y
ADD R0, R0, z
ST x, R0
```

Generate code for
a = b + c
d = a + e

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

Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by factors such as
  - **the level of the IR**
    - Low-level IR can help generate more efficient code.
    - If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement by-statement code generation, however, often produces poor code that needs further optimization.
  - **the nature of the instruction-set architecture**
    - Uniformity and completeness of ISA affects the code
      - e.g., floats required to be loaded in special registers.

Instruction Selection contd..

- **the desired quality of the generated code.**
  - Context and amount of information to process affects the code quality.
- if the target machine has an "increment" instruction (INC), then the three-address statement **a=a+1** can be implemented more efficiently by the single instruction **INC a**,

  rather than by a more obvious sequence

  that loads a into a register, adds one to the register,

  and then stores the result back into a.

```
LD   R0, a       // R0 = a
ADD  R0, R0, #1  // R0 = R0 + 1
ST   a, R0       // a = R0
```

Register allocation

- 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 not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.
- The use of registers is often subdivided into two subproblems:
- 1. **Register allocation:** which variables to be put into registers .
- 2. **Register assignment:** which register to use for a variable.

- Finding an optimal assignment of registers to variables is NP-Complete.

Evaluation Order

- 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. Have to provide the best evaluation order .
- Instruction order affects execution efficiency.
- Picking the best order is NP-complete.
- Optimizer / Code generator needs to look at multiple instructions at a time.

Machine Code for : t = a + b

d = c + t

```
1: R0 = a
2: R1 = b
3: R2 = c
4: R3 = R0 + R1
5: R4 = R2 + R3
6: d = R4
```

```
1: R0 = a
2: R1 = b
4: R2 = R0 + R1
3: R0 = c
5: R3 = R2 + R0
6: d = R3
```

The Target Language - A Simple target Machine Model

- 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, R0,R1,……Rn-1.
- Load operations: The instruction **LD dst , addr** loads the value in location addr into location dst.
  - An instruction of the form **LD r1,r2** is a register-to-register copy in which the contents of register r2 are copied into register r1.
- Store operations: The instruction **ST x,r** stores the value in register r into the location x. This instruction denotes the assignment x = r.
- Computation operations of the form **OP dst,src1,src2** ; where OP is a operator like ADD or SUB, and dst, src1, and src2 are locations, not necessarily distinct.

Target Machine Model contd..

- Unconditional jumps: The instruction **BR L** causes control to branch to the machine instruction with label L. (BR stands for branch.)
- Conditional jumps of the form **Bcond r, L ;** where r is a register, L is a label, and cond stands for any of the common tests on values in the register r.
  - For example, **BLTZ r , L** causes a jump to label L if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.

Addressing Modes

- The instruction **LD R1,a(R2)**
  - R1 = contents(a + contents(R2)), where contents(x) denotes the contents of the register or memory location represented by x.
- A memory location can be an integer indexed by a register.
  - **LD R1,100(R2)** ➡ R1 = contents(100 + contents(R2)), loading into R1 the value in the memory location obtained by adding 100 to the contents of register R2.
- Two indirect addressing modes:
  - ***r** means the memory location found in the location represented by the contents of **register, r** and
  - ***100(r)** means the memory location found in the location obtained by adding **100 to the contents of r**.
    - **LD R1,*100(R2)** ➡ R1 = contents(contents(100 + contents(R2))), loading into R1 the value in the memory location stored in the memory location obtained by adding 100 to the contents of register R2.
- An immediate constant addressing mode. The constant is prefixed by #. The instruction **LD R1,#100** loads the integer 100 into register R1, and **ADD R1,R1,#100** adds the integer 100 into register R1.
- Comments at the end of instructions are preceded by //.

5/7/2020
10:28 AM

Instruction Type	Example
Load	LD R1, x
Store	ST R1, x
Computation	SUB R1, R2, R3
Unconditional Jump	BR main
Conditional Jump	BLTZ R1, main

Addressing Mode	Example
Direct	LD R1, 100000
Named / Variable	LD R1, x
Variable Indexed	LD R1, a(R2)
Immediate Indexed	LD R1, 100(R2)
Indirect	LD R1, *100(R2)
Immediate	LD R1, #100

Machine Instructions: Generate code for the three-address statements

- 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
```