

Code Generation

- Code generation is the Final phase.
- Input is IR from the front end + symbol table.
- Back end = optimization + code generation.
- Optimization = IR to IR. Can be in multiple phases (does several passes). Optimization is optional!

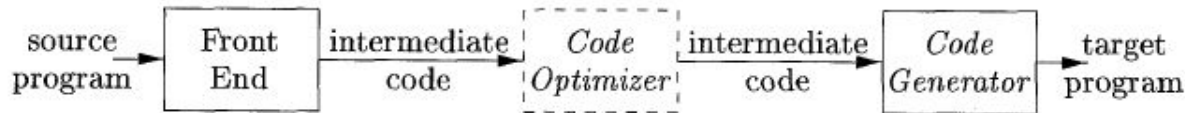


Figure 8.1: Position of code generator

Objectives of code generation

- Correctness. Semantic meaning of the source program should be preserved.
- It should be of high quality; that is, it must make effective use of the available resources of the target machine.
- Moreover, the code generator itself must run efficiently.

Objectives of code generation...

- Important criteria for CG: CORRECTNESS.
- Others
 - Ease of implementation
 - Testing to ensure correctness
 - Ease of maintenance
- We assume
 - In IR there are no syntactic or semantic errors.
 - Type checking completed, type conversion operators are embedded (as needed).

Challenges

- The problem of generating an optimal target program for a given source program is hard.
 - Sub-problems like register allocation (to be done efficiently), itself is intractable.
- We must be content with heuristics (which evolved over time with experience). And this does a good job in practice.

Primary tasks of CG

- Instruction selection
 - Choosing appropriate one from available
- Register allocation and assignment
 - What values to keep in which registers
- Instruction ordering
 - Deciding in what order to schedule the execution of instructions.

Issues in the design of a CG

- Depends on the form of IR, the target language, and the run-time system.
- The tasks mentioned, instruction selection, register allocation and assignment, and instruction ordering are encountered in any CG design.

- Various IRs we consider are:
 - three-address code,
 - trees (syntax trees), and
 - DAGs.

The target program

- The most common target-machine architectures are **RISC** (reduced instruction set computer) , **CISC** (complex instruction set computer), and **stack based**.
- RISC
 - Many registers
 - Three-address instructions
 - Simple addressing modes
 - Simple instruction-set architecture.

- CISC
 - Few registers
 - Two-address instructions
 - A variety of addressing modes
 - Several register classes
 - Variable length instructions
 - Instructions with side effects.

- Stack-based machine
 - Push operands onto a stack
 - Perform operations on the operands at the top
 - Top of the stack is kept in registers for high performance
 - Stack-based machines, almost disappeared, since it demands too many swap and copy operations.
 - However, they are revived with JVM (Java Virtual Machine)

- The common alternatives to stack machines are **register machines**, in which each instruction explicitly names specific registers for its operands and result.
- JVM simulates stack machine.
- Recall, PDA with two stacks can simulate a Turing machine.

Absolute vs relocatable target code

- Absolute machine-language program need to be placed at a fixed location in memory.
- Compilation time, execution time can be lesser.
- For embedded programs, like in a car, which needs to run in real time, this approach is suitable.
- Entire source program needs to be compiled at once. {you can not compile one function at a

Absolute vs relocatable target code

- Relocatable machine-language program (called object module) allows subprograms to be compiled separately.
- Flexibility. You can use precompiled tools/libraries.

3rd option

- Producing an assembly-language program as output makes the process of code generation easier.
- Price paid is: this is not target code. Assembler need to translate this to target code.
- We use this one, in our CG.

Instruction selection

- CG maps IR to a sequence of code of target m/c.
- Complexity of this depends on
 - The level of the IR
 - The nature of the instruction-set architecture
 - The desired quality of the generated code.

The level of the IR

- If IR is of high level, each IR statement can be translated into a sequence of m/c code using code templates.
- Such statement-by-statement CG is like interpretation and the resulting m/c code is slow.
 - M/c dependent code optimization is needed.
- If IR is closer to the target m/c, this results in efficient code.

- If each data type (basic data types) is supported in a uniform way, then it is easy ...since we do not worry about type!
- In principle, assume there is only one data type, like int, then every three-address statement of the form $x = y + z$, can be translated :

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

$$\begin{aligned}a &= b + c \\ d &= a + e\end{aligned}$$

would be translated into

```
LD  R0, b      // R0 = b
ADD R0, R0, c   // R0 = R0 + c
ST  a, R0      // a = R0
LD  R0, a      // R0 = a
ADD R0, R0, e   // R0 = R0 + e
ST  d, R0      // d = R0
```

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if *a* is not subsequently used.

Quality of the generated code

- Speed (to complete one IR stmt) & Size (of the code sequence).
- On most machines, a given IR can be implemented by many different code sequences.
 - Each quite different from other, with significant speed and size differences.
- Eg: If INC instruction is available, then $a = a+1$ can be implemented more efficiently than

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

Register allocation

- A key problem is, deciding what to store in which register.
- Registers are the fastest memory. But very few in number.
 - Some instructions, forces that its operands must be kept in the registers.

Register allocation

The use of registers is often subdivided into two subproblems:

1. *Register allocation*, during which we select the set of variables that will reside in registers at each point in the program.
2. *Register assignment*, during which we pick the specific register that a variable will reside in.

Example 8.1: Certain machines require *register-pairs* (an even and next odd-numbered register) for some operands and results. For example, on some machines, 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 and y, the multiplier, 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, the even register holds the remainder and the odd register the quotient.

Evaluation order

- The order in which computations are done can affect the efficiency
 - Some computation orders require fewer registers
- Picking the best order, in general, is NP-complete.
 - DAG analysis (we see this) can help.
- This is studied in detail in m/c dependent code optimization.

The Target Language

- Instruction set architecture of the target m/c is a must to be well understood by the compiler writer.
- But, we make many assumptions on the target m/c that simplifies the CG discussion.

A simple target m/c model

- We have n general-purpose registers, $R0, R1, \dots, R_{n-1}$.
- Limited set of instructions we consider.
 - We assume all operands are integers.

- *Load* operations:

LD $dst, addr$ $dst = addr$

LD r, x loads the value
in location x into register r .

LD r_1, r_2 *register-to-register copy*

- *Store* operations:

ST x, r $x = r$.
stores the value in register r into
the location x .

- *Computation* operations $OP\ dst, src_1, src_2$

OP is a operator like ADD or SUB.

SUB r_1, r_2, r_3 $r_1 = r_2 - r_3$.

Unary operators that take only one operand do not have a src_2 .

Jumps

- *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.

Other instructions are, like `BGTZ` (greater than), `BLEZ` (less or equal), `BGEZ` (greater or equal), `BEQZ` (equal to).

Addressing modes, we assume are:

- In instructions, a location can be a variable name x referring to the memory location that is reserved for x (that is, the l -value of x).
- Accessing array elements. $a(r)$ is located at memory address $a+r$ where a is the base address of an array and r is the index stored in a register.

For example, the instruction `LD R1, a(R2)` has the effect of setting $R1 = contents(a + contents(R2))$, 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. For example, `LD R1, 100(R2)` has the effect of setting $R1 = \text{contents}(100 + \text{contents}(R2))$, that is, of loading into R1 the value in the memory location obtained by adding 100 to the contents of register R2.
- We also allow 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 . For example, `LD R1, *100(R2)` has the effect of setting $R1 = \text{contents}(\text{contents}(100 + \text{contents}(R2)))$
- Finally, we allow 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.

`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
```

That is, the second step computes $8i$, and the third step places in register R2 the value in the i th element of `a` — the one found in the location that is $8i$ bytes past the base address of the array `a`.

`a[j] = c`

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


if $x < y$ goto L

LD	R1, x	// R1 = x
LD	R2, y	// R2 = y
SUB	R1, R1, R2	// R1 = R1 - R2
BLTZ	R1, M	// if R1 < 0 jump to M

Here, M is the label that represents the first machine instruction generated from the three-address instruction that has label L.