Code generation

introduction

- The final phase of compilation
- A careful code generation algorithm can easily produce code that can runs twice as fast as code produced by an ill considered algorithm
- The output of code generator is the object program
  - A variety of forms
  1. An absolute machine-language program
  2. A relocatable machine-language program
  3. An assembly language program
  4. Or a program in some other programming language

- If the output is
- absolute machine language program
  - has the advantage that it can be placed in a fixed location in memory and immediately executed
- Relocatable machine language program
  - Allows subprograms to be compiled separately
  - A set of relocatable object modules can be linked together and loaded for execution by a linking loader
  - More flexibility; we can call other previously compiled programs from an object module
- Many commercial compilers produce relocatable object modules

- Input to the code generator is in the form of a three-address code, usually a quadruple
- Assumptions is made when we reached this stage i.e.,
  - All necessary semantic checking has been done
  - Data type of each operator and operand is known
  - Data areas and offsets has been determined for each name
  - Intermediate text has been partition into basic blocks

Problems in code generation

- There are three main sources of difficulty
  - Deciding what machine instructions to generate
  - Deciding the order of computations
  - Deciding which registers to use

Deciding what machine instructions to generate

- Most machines permit certain computations to be done in a variety of ways
- Example
- If our target machine has “add-one-to-storage” instruction(AOS),
  - then for the three address statement A:=A+1
  - We might generate the single instruction

  AOS A

  instead of the following usual three instructions

  LOAD A

  ADD #1

  STORE A

Deciding the order of computations

- Some computations orders require fewer registers to hold intermediate results than others
- Hence picking the best order is a very difficult problem in general
- We shall simply generate code for the three address statements in the order in which they have been produced by the semantic routines

Deciding which registers to use

- Final problem is register assignment i.e. deciding in which register each computation should done
- The problem is further complicated because certain machines require *registers-pairs* for some operands and results
- Example IBM System/370 machines
- Integer multiplication and integer division involve registers pairs
- Example: a division instruction
  - D X, Y
  - The 64 bit dividend occupies an even/odd registers pair whose even register is X, Y represent the divisor
  - After division the even register holds the remainder and the odd register holds the quotient

A Machine model

- Good code generation requires an intimate knowledge of the target machine.
- An example machine we will use is similar to DEC PDP-11
- It is a byte-addressable machine with two bytes per word
- Memory size **2^{16}** bytes i.e 65536 bytes implies 524288 bits
  - Therefore if 16 bits is the size of a/each word then we have 32768 words in this memory ie **2^{15}** 16-bit words of memory
- 8 general-purpose registers, *R*0 to *R*7(16-bit)
- The binary operators are of the form

  Opcode (4-bit) Source(6-bit) , Destination (6-bit)
  - 6 bits not sufficient if source/destination is a memory word.
  - certain bit patterns in these fields indicate that words following an instruction will contain operand and/or addresses
  - Actual address can be in another word.

Op-codes used among others

- MOV (move source to destination)
- ADD (add source to destination)
- SUB (subtract source from destination)

Addressing Modes

- *used in the source or destination*
- *r* (register addressing)
  - Register *r* contains the operand
- **r* (indirect register)
  - Register *r* contains the address of the operand
- X(r) (indexed mode)
  - Value X is added to the contents of register r to produce the address of the operand
- *X(r) (indirect indexed mode)
  - Value X is added to the contents of register r to produce the address of the word containing the address of the operand
- *X* (absolute address)
  - The address of *X* follows the instruction
- #X (immediate)
  - The word following the instruction contains the literal operand *X*

Evaluation of Cost

- Instruction length is used to measure cost
- fetching takes more time
- minimize length minimize execution time

Examples of machine instruction

1. MOV R0,R1
   - copies the contents of register 0 to register 1.
   - the instruction has **cost one** since it occupies only one word of memory
2. MOV R5, M
   - copies the contents of R5 into memory location M
   - This instruction has **cost two**, since the address location M is in the word following the instruction
3. ADD #1, R3
   - Adds the constant 1 to the contents of R3
   - It has **cost one**, since constant 1 must appear in the next word
4. SUB 4(R0),*5(R1)
   - Subtracts ((R0)+4) from (((R1)+5)) where (X) denotes the contents of register or location X
   - The result is stored at the destination *5(R1)
   - **Cost is 3**, since constants 4 and 5 are stored in the next two words following the instruction

Example

- For a quadruple of the form A:=B+C where B and C are simple variables in distinct memory locations of the same name
- We can generate a variety of code sequences

- **Case 1 : Values of *B* and *C* are in memory**
  - **Cost is 6**

```
MOV B, R0
ADD C, R0
MOV R0, A
     OR
MOV B, A
ADD C, A
```

- **Case 2 : registers R0 ,R1 and R2 contains the address of A, B and C resp.**
  - **Cost is 2**

```
     MOV *R1, *R0
     ADD *R2, *R0
```

- **Case 3 : Values of B and C are in registers *R1* and *R2***
  - ***Cost is 3***

```
ADD R2, R1
MOV R1, A
```

A simple code generator

- A sequence of quadruples are given as input
- We generate code for each quadruple
- Also remembering if any of the operands of the quadruple are currently in registers and taking advantage of that fact
- We assume for simplicity that for each operator in the quadruple there is a corresponding machine code operator
- Leave the computed result in the register as long as possible
- Store them only if the register is
  - required for other computation
  - Before a procedure call, jump or labeled statement
- Store everything when moving across block boundaries as well when procedure calls are made

- Example A:=B+C
- ADD Rj, Ri (cost=1)
  - Provided C is in Rj and B is in Ri and B is <u>not live</u> after the statement and leaving A in register Ri
- ADD C, Ri (cost =2)
  - C is in memory location, B is in Ri and B is <u>not live</u> after the statement
- MOV C, Rj
- ADD Rj, Ri (cost=3) provided if C is subsequently used

Next-Use Information

- Making informed decisions **concerning register allocation**
- definition
  - Suppose quadruple *i* assigns a value to A. If quadruple *j* has A as an operand , and control can flow from quadruple *i* to *j* along a path that has no intervening assignments to A, then we say quadruple *j* ***uses*** the value of A computed at *i.*
- For each quadruple A:=B op C what the next uses of A, B and C are

Algorithm for Next-Use Information

- make a backward pass over each block
- Determine whether A has next use, if not whether live on exit
- Data flow analysis tells us which variables are live on exit from each block
- If not done, assume all non-temporary are live on exit and mark temporaries that might be live

Quadruple i say *i:* A:=B **op** C

- set A to “not live” and “no next use”
- set B and C to “live” and “next use to *i*”

Register and Address Descriptor

- Register Descriptors
  - To perform register allocation we need a Register Descriptor to keep track of what is *currently in each register*.
  - This Register Descriptor is consulted whenever a new register is required
- Address Descriptors
  - For each **name** in the block there is an address descriptor that keeps track of the location(s) where the current value of the name may be found at run time
  - The location might be a register, stack, memory address or some set of these
  - This info is stored in the symbol table and is used to determine the accessing method for a name

Algorithm for Code Generation

For every three-address statement of the form *A* = *B op C* in the basic block

do{

1. Call GETREG() to obtain the location *L* in which the computation *B op C* should be performed.

   /* Pass the index of the quadruple as a parameter to GETREG() */
2. Obtain the current location of the operand *B* by consulting its address descriptor, and if the value of *B* is currently both in the memory location as well as in the register, then prefer the register. If the value of *B* is currently not available in *L*, then generate an instruction MOV B, L (where *B* as assumed to represent the current location of *B*).
3. Generate the instruction OP C, L, and update the **address descriptor** of *A* to indicate that *A* is now available in *L*, and if *L* is in a register, then update **its descriptor** to indicate that it will contain at run-time the value of *A*.
4. If the current values of *B* and /or *C* are in the register, and we have no further uses for them, and they are not live at the end of the block, then alter the register descriptor to indicate that after the execution of the statement *A* = *B op C*, those registers will no longer contain *B* and /or *C*.

}

Store all the results.

Algorithm for Code Generation

For every three-address statement of the form *A* := *B* in the basic block

do{

}

Store all the results.

- If B is in a register, change the register and address descriptors to record that the value of A is now found only in the register holding A, provided B has no next use and not live on exit
- If B is only in memory then
    - We must use getreg to find a register to load B and make that register a location for A
    - If A has no next use in the block, then MOV B,A is preferable

- Once all quadruples are processed we store by MOV instructions those names that are live on exit and not in their memory locations
    - Register descriptor is used to find out what names are left in the register
    - Address descriptor to determine that the same name is not already in memory locations
    - Live variable information to determine whether the name is to be stored

Function GETREG()

1. First, it searches for a register already containing the name *B*. If such a register exists, and if *B* has no further use after the execution of *A* = *B op C*, and if it is not live at the end of the block and holds the value of no other name, then return the register for *L*.
2. Otherwise, GETREG() searches for an empty register; and if an empty register is available, then it returns it for *L*.
3. If no empty register exists, and if *A* has further use in the block, or *op* is an operator such as indexing that requires a register, then getreg() finds a suitable, occupied register. The register is emptied by storing its value in the proper memory location *M*, the address descriptor is updated, the register is returned for *L*. (The least-recently used strategy can be used to find a suitable, occupied register to be emptied.)
4. If *A* is not used in the block or no suitable, occupied register can be found, getreg() selects a memory location of *A* and returns it for *L*.

Example

The three-address statement for

W:=(A-B) + (A-C) + (A-C)

is

T:=A – B

U:=A – C

V:=T + U

W:=V + U

Code sequence

Statement	**Code generated**	**Register descriptor**	**Address descriptor**
		Register is empty	
T:=A-B	MOV A,R0 SUB B,R0	R0 contains T	T in R0
U:=A-C	MOV A,R1 SUB C,R1	R0 contains T R1 contains U	T in R0 U in R1
V:=T+U	ADD R1,R0	R0 contains V R1 contains U	U in R0 V in R0
W:=V+U	ADD R1,R0 MOV R0,W	R0 contains W	W in R0 W in R0 and memory