

Code Generation

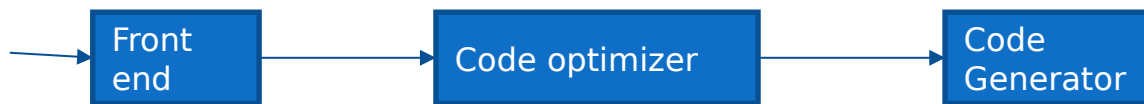
The final phase of a compiler is code generator

It receives an intermediate representation (IR) with supplementary information in symbol table

Produces a semantically equivalent target program

Main tasks:

- Instruction selection
- Register allocation and assignment
- Instruction ordering



Issues in the Design of Code Generator

- The most important criterion is that it produces correct code
- Input to the code generator
 - IR + Symbol table
 - We assume front end produces low-level IR, i.e. values of names in it can be directly manipulated by the machine instructions (e.g. Three address code).
 - Syntactic and semantic errors have been already detected
- The target program
 - Common target architectures are: RISC, CISC and Stack based machines

Complexity of mapping

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code

$x = y + z$

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

$a = b + c$

$d = a + e$

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

Register allocation

- Two subproblems
 - Register allocation: selecting the set of variables that will reside in registers at each point in the program
 - Register assignment: selecting specific register that a variable reside in
- Complications imposed by the hardware architecture

Instruction Selection

Straight forward code if efficiency is not an issue

```
a=b+c      Mov b, R0
d=a+e      Add c, R0
           Mov R0, a
           Mov a, R0    (redundant code, can be eliminated)
           Add e, R0
           Mov R0, d

a=a+1      Mov a, R0    (if there is an instruction like "Inc a", that can be used instead)
           Add #1, R0
           Mov R0, a
```

Partition the intermediate code into basic blocks

Basic blocks

sequence of statements in which flow of control enters at beginning and leaves at the end

Algorithm to identify basic blocks

- determine leader
 - first statement is a leader
 - any target of a goto statement is a leader
 - any statement that follows a goto statement is a leader
- for each leader its basic block consists of the leader and all statements up to next leader

Flow graphs

add control flow information to basic blocks

- nodes are the basic blocks
- there is a directed edge from B1 to B2 if B2 can follow B1 in some execution sequence, i.e.
 - either there is a jump from the last statement of B1 to the first statement of B2, or
 - B2 follows B1 in natural order of execution
- initial node: block with first statement as leader

The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.

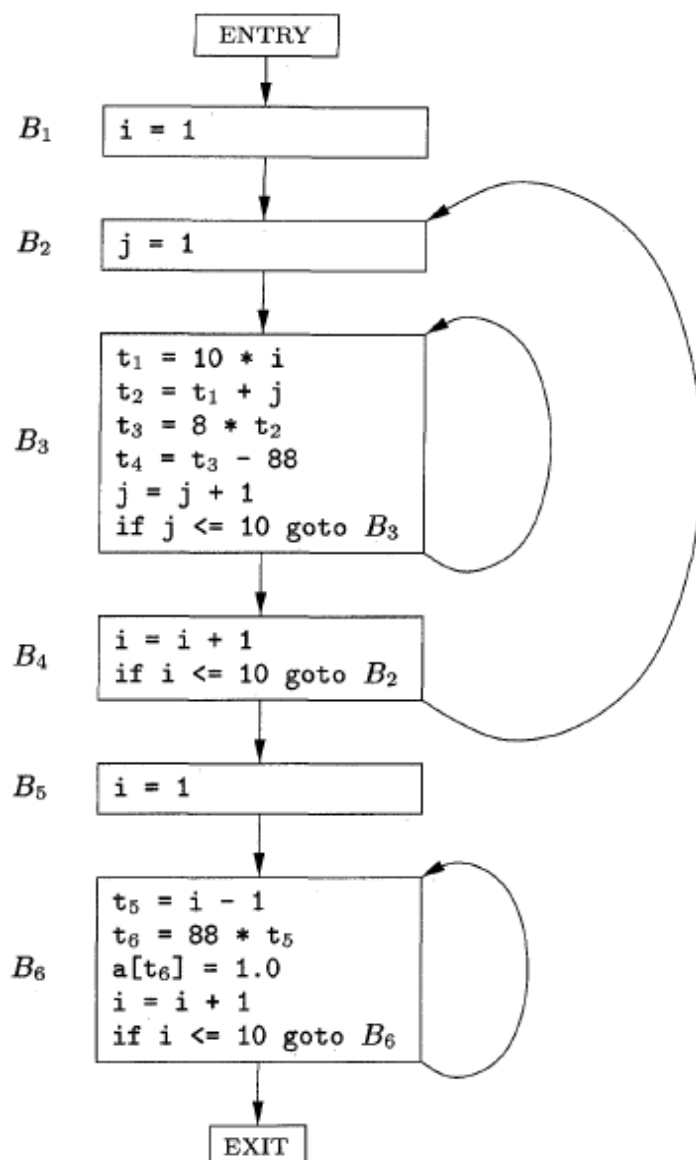
Control will leave the block without halting or branching, except possibly at the last instruction in the block.

Example: Intermediate code to set a 10*10 matrix to an identity matrix

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i,j] = 0.0;
for i from 1 to 10 do
    a[i,i] = 1.0;
```

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Flow graph based on Basic Blocks



If a register contains a value for a name that is no longer needed, we should re-use that register for another name (rather than using a memory location) . So it is useful to determine whether/when a name is used again in a block

Definition: Given statements i , j , and variable x , If i assigns a value to x , and j has x as an operand, and no intervening statement assigns a value to x , then j uses the value of x computed at i .

EXAMPLE:

(5) $X := \dots$
 ... (no ref to X) ...
(14) $\dots := \dots X \dots$

X is live at (5)

X is live at (5) because the value computed at (5) is used later in the basic block.

X 's next use at (5) is (14). It is a good idea to keep X in a register between (5) and (14).

EXAMPLE:

(12) $\dots := \dots X \dots$
 ... (no ref to X) ...
(25) $X := \dots$

X is dead at (12)

X is dead at (12) because its value has no further use in the block.

Don't keep X in a register after (12).

Liveness and next-use information

Requirement is to determine for each three address statement $x=y+z$ what the next uses of x , y and z are.

Algorithm:

- scan each basic blocks backwards
- assume all temporaries are dead on exit and all user variables are live on exit
- Example: Suppose we are scanning
 $i : X := Y + Z$ in backward scan
 - Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x , y , and z .
 - In the symbol table, set x to "not live" and "no next use."
 - In the symbol table, set y and z to "live" and the next uses of y and z to i .

Example

1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

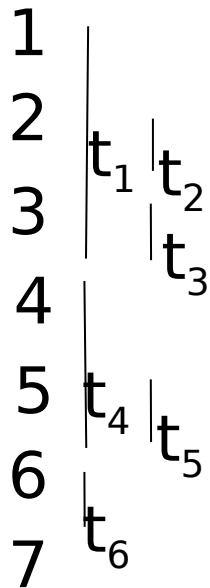
STATEMENT

7: no temporary is live
6: t_6 :use(7), t_4 t_5 not live
5: t_5 :use(6)
4: t_4 :use(6), t_1 t_3 not live
3: t_3 :use(4), t_2 not live
2: t_2 :use(3)
1: t_1 :use(4)

Symbol Table

t_1	dead	Use in 4
t_2	dead	Use in 3
t_3	dead	Use in 4
t_4	dead	Use in 6
t_5	dead	Use in 6
t_6	dead	Use in 7

Example of Reuse



1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_2 = 2 * t_2$
4: $t_1 = t_1 + t_2$
5: $t_2 = b * b$
6: $t_1 = t_1 + t_2$
7: $X = t_1$

Code Generator

- consider each statement
 - remember if operand is in a register
 - Register descriptor
 - Keep track of what is currently in each register.
 - Initially all the registers are empty
 - Address descriptor
 - Keep track of location where current value of the name can be found at runtime
 - The location might be a register, stack, memory address or a set of those
- for each $X = Y \text{ op } Z$ do
- invoke a function `getreg` to determine location L where X must be stored. Usually L is a register.

- Consult address descriptor of Y to determine Y'. Prefer a register for Y'. If value of Y not already in L generate
 Mov Y', L
- Generate op Z', L
- Again prefer a register for Z. Update address descriptor of X to indicate X is in L. If L is a register update its descriptor to indicate that it contains X and remove X from all other register descriptors.
- If current value of Y and/or Z have no next use and are dead on exit from block and are in registers, change register descriptor to indicate that they no longer contain Y and/or Z.

Function getreg

1. If Y is in register (that holds no other values) and Y is not live and has no next use after
 $X = Y \text{ op } Z$
 then return register of Y for L.
2. Failing (1) return an empty register
3. Failing (2) if X has a next use in the block or op requires register then get a register R, store its content v into M (by Mov v, M) and use it.
4. else select memory location X as L

(If we are not OK by one of the first two cases, then we need to generate the move instruction Mov v, M to place a copy of v in its own memory location. This operation is called a spill. - Register Spilling)

Example

Stmt	code	reg desc	addr desc
$t_1 = a - b$	mov a, R₀ sub b, R₀	R ₀ contains t ₁	t ₁ in R ₀
$t_2 = a - c$	mov a, R₁ sub c, R₁	R ₀ contains t ₁ R ₁ contains t ₂	t ₁ in R ₀ t ₂ in R ₁
$t_3 = t_1 + t_2$	add R₁, R₀	R ₀ contains t ₃ R ₁ contains t ₂	t ₃ in R ₀ t ₂ in R ₁
$d = t_3 + t_2$	add R₁, R₀ mov R₀, d	R ₀ contains d	d in R ₀ d in R ₀ and memory

Global register allocation

Previous algorithm does local (block based) register allocation

This results that all live variables be stored at the end of block

To save some of these stores and their corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally)

Some options are:

- Keep values of variables used in loops inside registers
- Use graph coloring approach for more globally allocation

(Global register allocation will not be discussed for this class)