Compiler Design

Samit Biswas

samit@cs.iiests.ac.in

Department of Computer Science and Technology,
Indian Institute of Engineering Science and Technology, Shibpur

November 6, 2017

Code Generation

Code Generation :

- The final phase of a compiler is code generator.
- It receives an intermediate representation (IR) with supplementary information in symbol table.
- Produces semantically equivalent target program.
- Code generators - Main Tasks:
  - Instruction selection.
  - Register allocation and selection.
  - Instruction Ordering.

Code Generation Issues

- The most important criterion is that it produces correct target code.
- **Input to code generator**
  - IR + Symbol Table.
  - We assume that front-end produces low level IR; i.e. values in it can be directly manipulated by the machine instructions.
- **The target Program**
  - Prerequisite: target machine and its instruction set.

Instruction Selection

Every three address statement of the form :

$$x = y + z$$

translated into the following code sequences:

```
MOV y, R0    /* load y into register R0 */
ADD z, R0     /* add z to R0    */
MOV R0, x    /*  store R0 to x  */
```

The desired quality of target code:

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

would be translated into :

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

A Simple Target Machine Model

- Byte addressable machine with four bytes to a word.
- n general purpose registers, $R0, R1, \ldots, Rn-1$
- It has the address instruction of the form

  op *source destination*

  op is an op-code; *source* and *destination* are data field.
- It has the following op-codes (among others)
  - MOV (move source to destination)
  - ADD (add source to destination)
  - SUB (subtract source to destination)

Table: Address modes

Mode	Form	Address
absolute	M	M
register	R	R
indexed	c(R)	$c + contents(R)$
indirect register	*R	$contents(R)$
indirect indexed	*c(R)	$contents(c + contents(R))$

$contents(a)$ denotes the contents of register or memory address represented by a.

Assignment Statement: d = (a - b) + (a - c) + (a - c) might be translated into the following TAC:

$$
\begin{aligned}
t &= a - b \\
u &= a - c \\
v &= t + u \\
d &= v + u
\end{aligned}
$$

with d live at end.
Generated Code:

Statements	Code Generated	Register Descriptor	Address Descriptor
		Register empty	
t = a - b	MOV a, R0 SUB b, R0	R0 Contains t	t in R0
u = a - c	MOV a, R1	R0 contains t	t in R0
	SUB c, R1	R1 contains u	u in R1
v = t + u	ADD R1, R0	R0 contains v	t in R0
		R1 contains u	v in R0
d = v + u	ADD R1, R0	R0 contains d	d in R0
	MOV R0, d		d in R0 and Memory

Code Sequences for indexed assignments

Indexing operations in three address statements are handled in the same manner as binary operations. Consider the following indexed assignments:

```
a = b[ i ]
a[ i ] = b
```

Code Sequences for indexed assignments:

Statement	i in Register R_i Code	i in Memory M_i Code
a = b[i]	MOV b(Ri), R	MOV Mi, R MOV b(R), R
a[i] = b	MOV b, a(Ri)	MOV Mi, R MOV b, a(R)

Code Sequences for pointer assignments

Consider the following pointer assignments:

a = ∗p

∗p = a

Statement	p in Register R_p Code	p in Memory M_p Code
a = *p	MOV ∗R_p, a	MOV Mp, R MOV *R, R
*p = a	MOV a, ∗R_p	MOV Mp, R MOV a, *R

Code Sequences for Conditional Statements

- For branch instruction set a condition code to to indicate whether the last quantity computed or loaded into a register is negative, zero or positive.
- Compare instruction (CMP in our case) has the desirable property that it set the condition code without actually computing a value.
- That is *CMP x, y* sets a condition code to positive *if* $x > y$, and so on. a conditional-jump machine instruction makes the jump if a designated condition $<, =, >, \leq, \neq, \geq$ is met.

$$\text{if } x < y \text{ goto z}$$

This can be translated to

```
CMP x, y
CJ<   z     /*jump if the condition code is negative or zero*/
```

Example Conditional Statement

```
x = y + z
if x < 0 goto z
```

After translation:

```
MOV y, R0
ADD z, R0
MOV R0, x
CJ<  z
```

The condition code was determined by x after **ADD z, R0**.

Basic Block and Flow Graphs

- A graph representation of TAC called a **flow graph**.
- Nodes in flow graph represent computations.
- The edges represents the flow of controls.
- Some register assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

Basic Block and Flow Graphs

- **Basic Block:** A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.
- A name in a basic block is said to be live at a given point if its value is used after that point in the program, perhaps in another basic block.

Algorithm: Partition into Basic Block

Input: A sequence of Three Address Statement.
Output: A list of Basic Blocks with each TAC in exactly one block.

Method:

1. Determine the leaders, the first statements of Basic blocks.

   i) The first statement is a leader.

   ii) Any statement that is the target of a conditional or unconditional goto is a leader.

   iii) Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

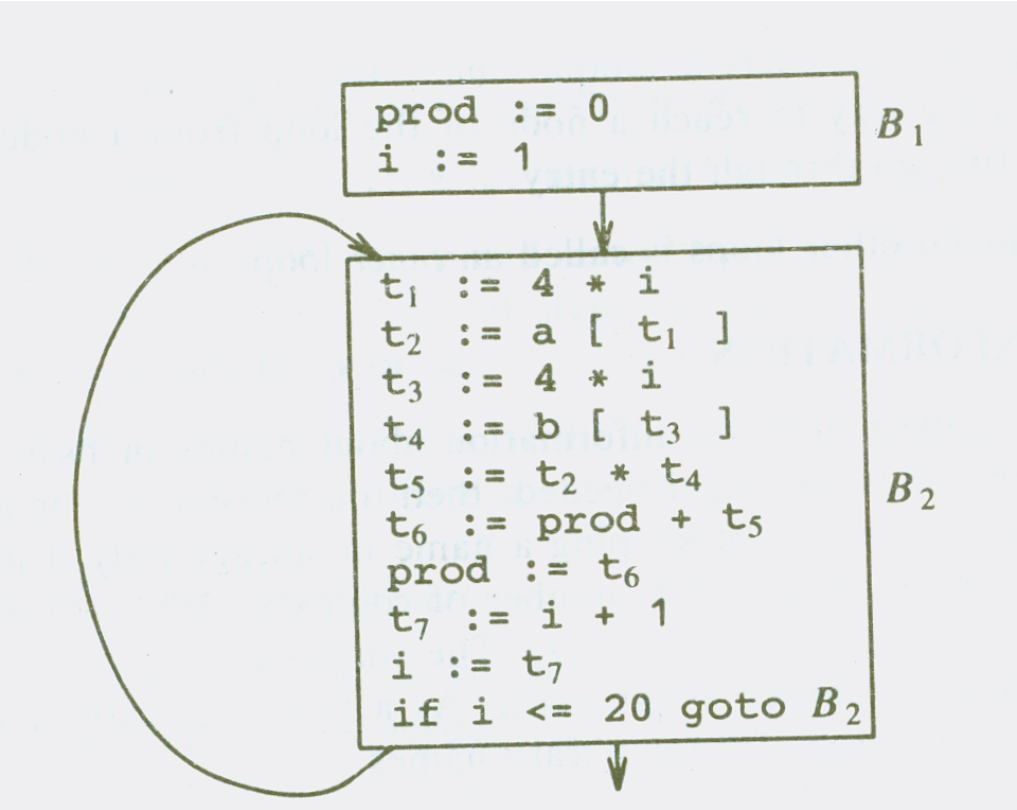
Consider the following code fragment - It computes the dot product of two vectors *a* and *b* of length 20.

```
begin
      prod := 0;
      i := 1;
      do begin
          prod := prod + a[i] * b[i];
          i := i + 1
      end
      while i <= 20
end
```

TAC to computing the dot product:

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]          /* compute a[i] */
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]          /* compute b[i] */
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
```

Flow graph for the program:



Register and address Descriptors

- A register descriptor keeps track of what is currently in each register. It is consulted whenever a new register is needed.
- An address descriptor keeps track of the location where the current value of the name can be found at runtime.

A Code Generation Algorithm

The Code generation Algorithm takes as input a sequence of three-address statements constituting a basic block. For each three address statement ($x = y$ op z) perform the following:

1. Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.
2. Consult address descriptor. If the value of y is not already in L generate instruction **MOV y, L** to place a copy of y in L.
3. Generate the instruction **OP z, L**. If L is a register, update descriptor to indicate that it contains the value of x, and remove x from all other register descriptors.
4. If the current values of y and / or z have no next uses, they are not live on exit from the block. Alter the register descriptor.

The function *getreg*

getreg returns the location L to hold the value of x for the assignment ($x = y$ op z).

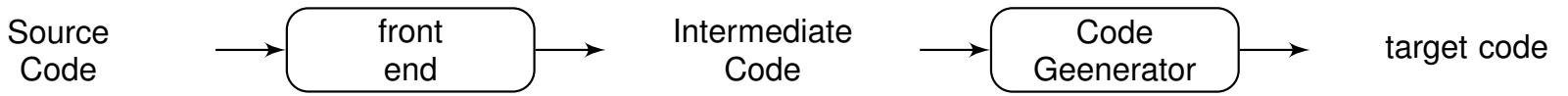
1. If the name y is in a register that holds the value of no other names and If the y is not Live and has no next use after execution of ($x = y$ op z) then return the register of y for L. Update the address descriptor of y that y is no longer in L.
2. Failing (1), return an empty register for L if there is one.
3. Failing (2), find an occupied register R, to be get freed.
4. If x is not used in the block, or no suitable occupied register can be found select memory location of x as L.

Code Optimization

Code Optimization

- A transformation to a program to make it run faster and/or take up less space.
- Optimization should be safe, preserve the meaning of a program.
- Code optimization is an important component in a compiler.

Getting Better Performances



User can
profile program, change algorithm, transform loops

Compiler can
improve loops, procedure calls, address calculations

Compiler can
use registers, select instructions, do peephole transformation

Levels:

- Window - Peephole Optimization
- Procedural - Global (Control flow graph)

Peephole Optimization

- examining a short sequence of target instructions and replacing these by faster sequences.
- Peephole is a small moving window on the target program.

Characteristics of Peephole Optimization

- redundant-instruction Elimination
- flow-of-control optimiztions
- algebraic simplifications
- use of machine idioms

Copy folding

```
x = 32;          |  Becomes  |  x =64;
x = x+32;        |           |
```

Unreachable Code:

```
goto L2;
x = x+1;          unneeded
```

flow - of - Control Optimization

- **algebraic simplifications**
  $x = x + 0 \quad \leftarrow$ Unneeded

- **Dead Code**
  $x = 32 \quad \leftarrow$ where x not used after statement
  $x = x + y \quad \rightarrow$ x = y + 32

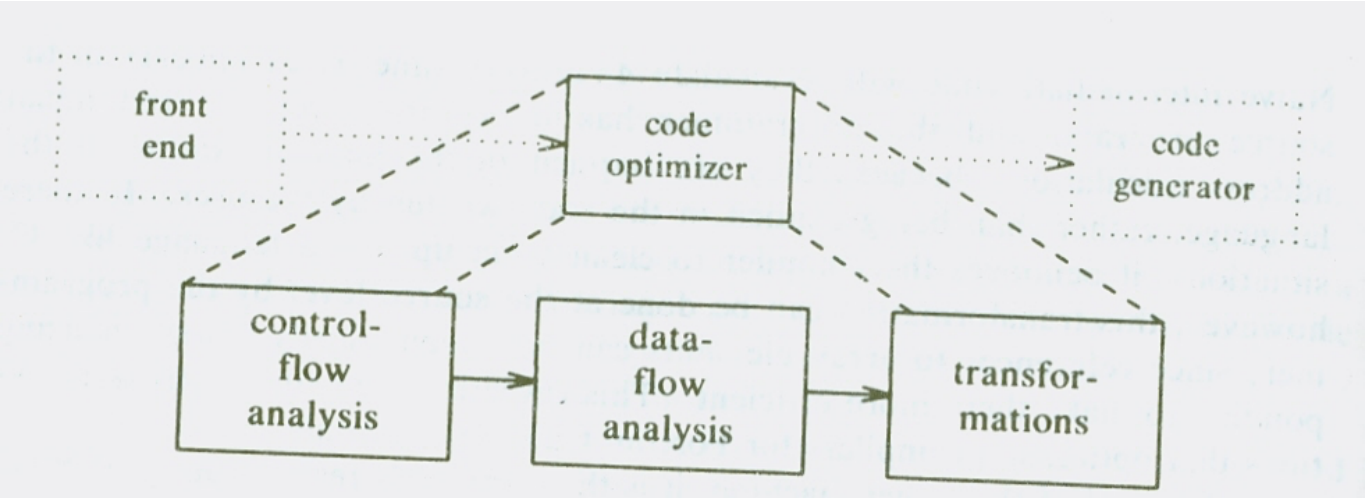
- **Reduction in Strength** - replace an expensive operation by a cheaper one

  $x = x * 2 \quad \rightarrow$ x = x + x

Peephole Optimization – Limitations

- Local in Nature.
- Pattern Driven.
- Limited by the size of the window.

Optimizing Compiler (Code Optimizer)



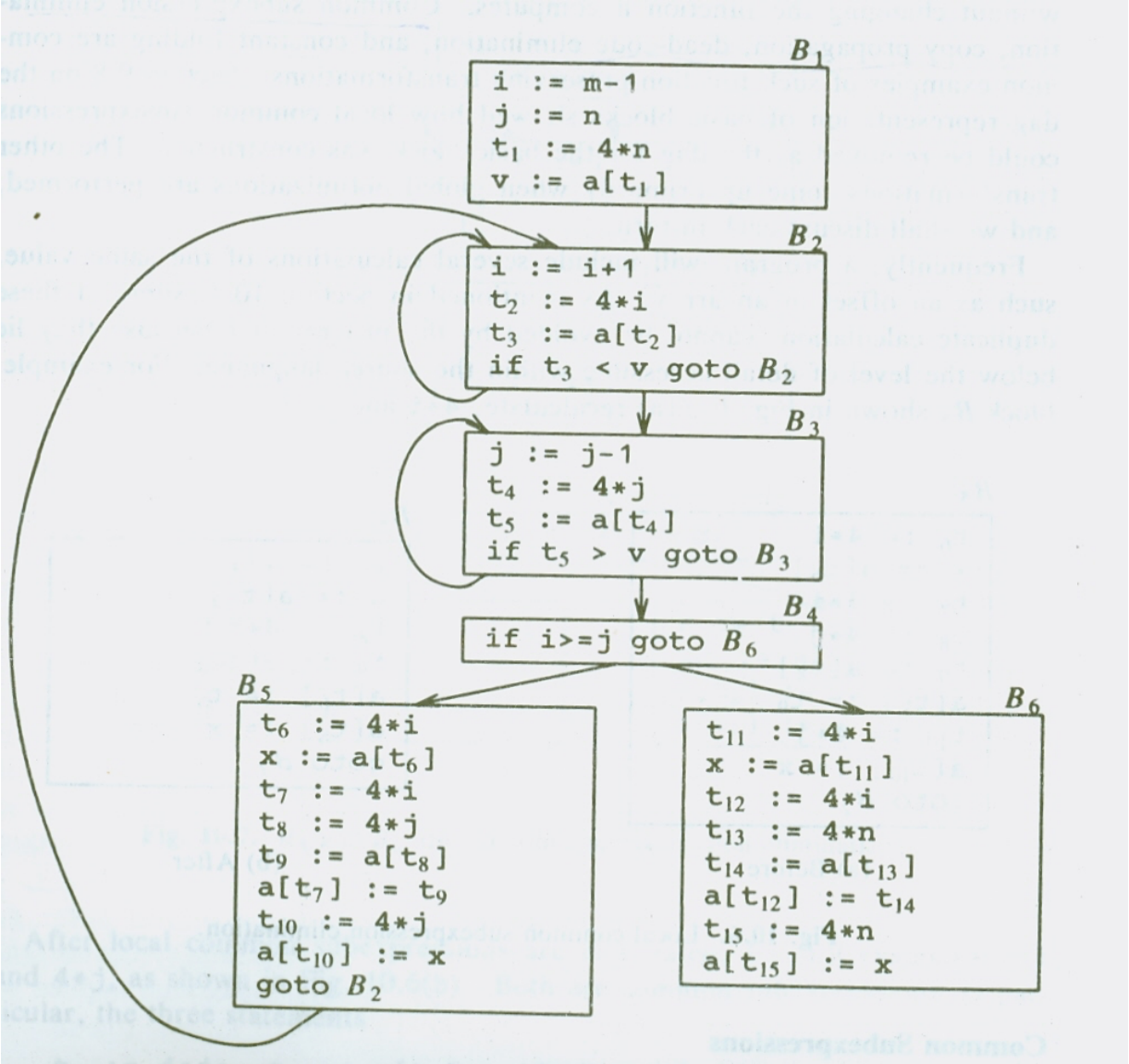
C Code for quicksort

```
void quicksort(m,n)
int m,n;
{
    int i,j;
    int v,x;
    if ( n <= m ) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while(1) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if (i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
```

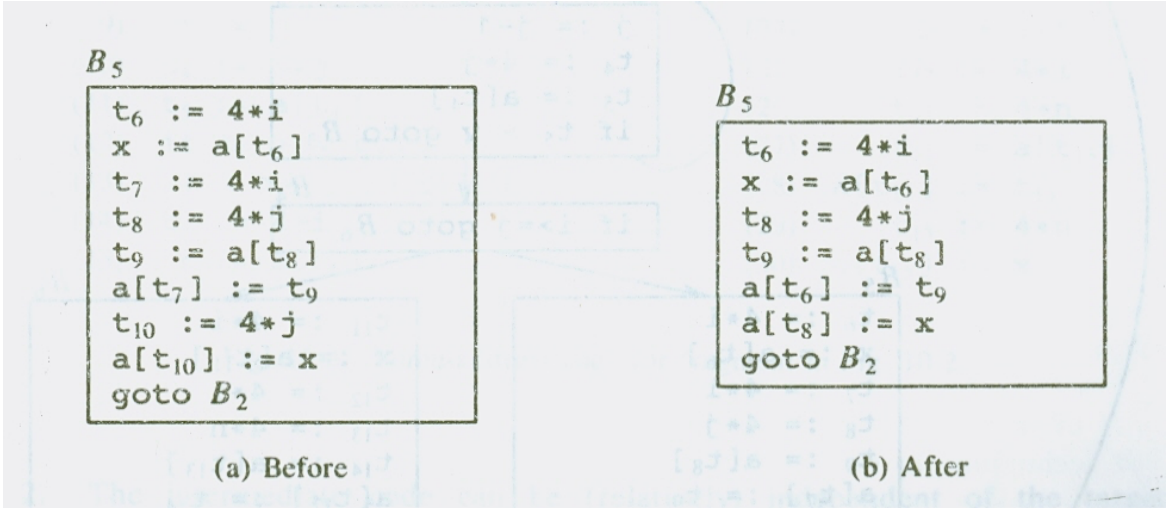
Optimizing Compiler (Code Optimizer)

```
(1)   i := m-1
(2)   j := n
(3)   t1 := 4*n
(4)   v := a[t1]
(5)   i := i+1
(6)   t2 := 4*i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)   j := j-1
(10)  t4 := 4*j
(11)  t5 := a[t4]
(12)  if t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)  x := a[t6]
(16)  t7 := 4*i
(17)  t8 := 4*j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4*j
(21)  a[t10] := x
(22)  goto (5)
(23)  t11 := 4*i
(24)  x := a[t11]
(25)  t12 := 4*i
(26)  t13 := 4*n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4*n
(30)  a[t15] := x
```

Flow Graph


B1
i := m-1
j := n
t1 := 4*n
v := a[t1]
B2
i := i+1
t2 := 4*i
t3 := a[t2]
if t3 < v goto B2
B3
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
B4
if i>=j goto B6
B5
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
B6
t11 := 4*i
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x

Common Subexpressions



- t_7 and t_{10} have common subexpressions $4 * i$ and $4 * j$ respectively in B_5.
- Eliminated by using t_6 instead of t_7 and t_8 instead of t_10.

- After common subexpressions are eliminated B_5 still evaluates $4 * i$ and $4 * j$. Both are common subexpressions ; in particular, the three statements $t_8 = 4 * j; t_9 = a[t_8]; a[t_8] = x$ in B_5 can be replaced by $t_9 = a[t_4]; a[t_4] = x$

Copy Propagations

Loop Optimizations

Mostly used loop optimization techniques are :

- **Code Motion** - which moves code outside the loop.
- **Induction variable elimination** - apply to eliminate i and j from the inner loops B_2 and B_3
- **Reduction in Strength** - which replaces an expensive operation by a cheaper one, such as multiplication by an addition.

Code Motion

```
while (i <= limit-2)    /* statement does not limit
```

Code motion will result in the equivalent of

```
t = limit - 2;
while (i <= t)    /* statement does not limit or t
```

Induction variables and Reduction in Strength