

## UNIT-5

### SYLLABUS

- **Code Generation:** -
- **Basic Blocks and Flow Graphs**-Basic Blocks, Next use Information, Flow Graphs, Representation of Flow Graph, Loops.
- **Optimization of Basic Blocks:**
- The DAG representation of basic blocks –Finding Local common subexpressions – Dead code elimination – The use of Algebraic Identities -
- **Machine independent code optimization** -Principle sources of Optimization- Causes of Redundancy, Running example: Quick Sort, Semantic Preserving transformations, Global common sub expressions, copy propagation, dead code elimination, code motion, induction variables, and reduction in strength.
- **Machine dependent code optimization:**
- Peephole optimization: Eliminating redundant loads and stores – Eliminating Unreachable code – Flow-of-control optimizations – Algebraic simplification and Reduction in strength - Use of Machine idioms
- Register allocation: Global Register Allocation

## THREE ADDRESS CODE

Fragment of Source Code	Three Address Code
begin	1. $p = 0$
$p := 0;$	2. $i = 1$
$i = 1;$	3. $t_1 = 4 * i$
do begin	4. $t_2 = a[t_1]$
$p = p + a[i] * b[i];$	5. $t_3 = 4 * i$
$i = i + 1$	6. $t_4 = b[t_3]$
end	7. $t_5 = t_2 * t_4$
while $i \leq 20;$	8. $t_6 = p + t_5$
$j = j + 1$	9. $p = t_6$
end	10. $t_7 = i + 1$
	11. $i = t_7$
	12. if $i \leq 20$ goto (3)
	13. $t_8 = j + 1$
	14. $j = t_8$

- A graph representation of intermediate code that is helpful for code generation.
- We can do a better job of register allocation if we know how values are defined and used.
- We can do a better job of instruction selection by looking at sequences of three-address statements.
- Transformations on flow graphs that turn the original intermediate code into optimized code from which better target code can be generated.
- The optimized code is turned into machine code using the code generation technique.

## BASIC BLOCKS

- First job is to partition the three address code instructions into basic blocks.
- We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump or a conditional jump and labels, control proceeds sequentially from one instruction to the next.

### Basic Blocks construction

**Algorithm 8.5:** Partitioning three-address instructions into basic blocks.

**INPUT:** A sequence of three-address instructions.

**OUTPUT:** A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

**METHOD:** First, we determine those instructions in the intermediate code that are *leaders*, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program. □

```

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

```

### Source code

```

1) i = 1 ← leader
2) j = 1 ← leader
3) t1 = 10 * i ← leader
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) ← leader
10) i = i + 1 ← leader
11) if i <= 10 goto (2) ← leader
12) i = 1 ← leader
13) t5 = i - 1 ← leader
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13) ← leader

```

```

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)

```

Figure 8.7: Intermediate code to set a  $10 \times 10$  matrix to an identity matrix

## Basic Blocks and Flow Graphs

The representation is constructed as follows:

1. Partition the intermediate code into *basic blocks*, which are maximal sequences of consecutive three-address instructions with the properties that
  - (a) 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.
  - (b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.
2. The basic blocks become the nodes of a *flow graph*, whose edges indicate which blocks can follow which other blocks.

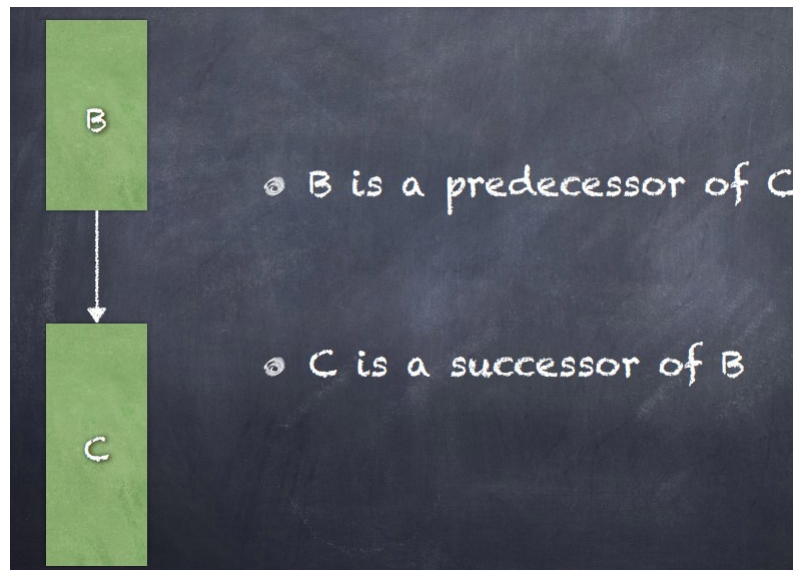
## FLOW GRAPHS

Each basic block is a **node** in the flow graph.

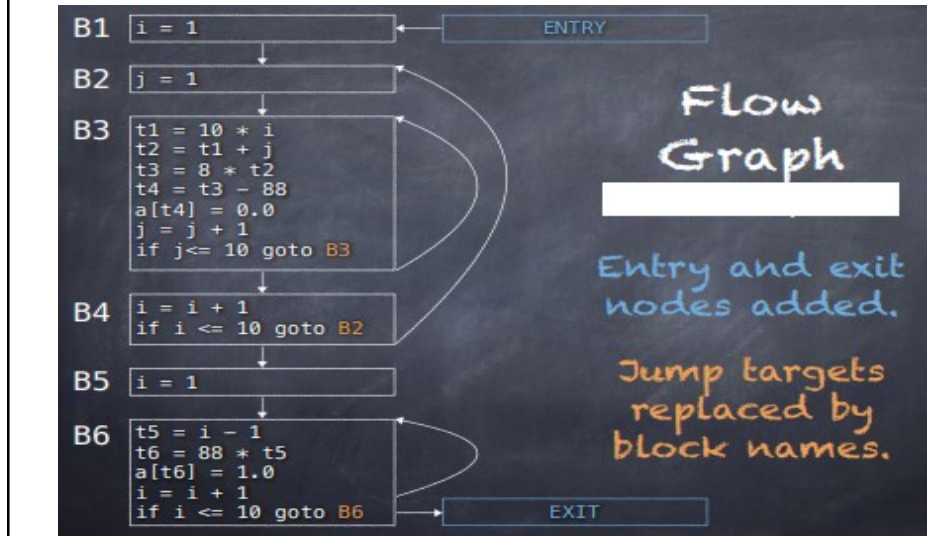
There is an **edge** between blocks B and C of the flow graph if:

1. there is a (conditional or unconditional) jump from from the end of B to the start of C, or
2. C immediately follows B and B does not end with an unconditional jump.

## TERMINOLOGY



- The Basic block of each leader contains all the instructions from itself until just before the next leader

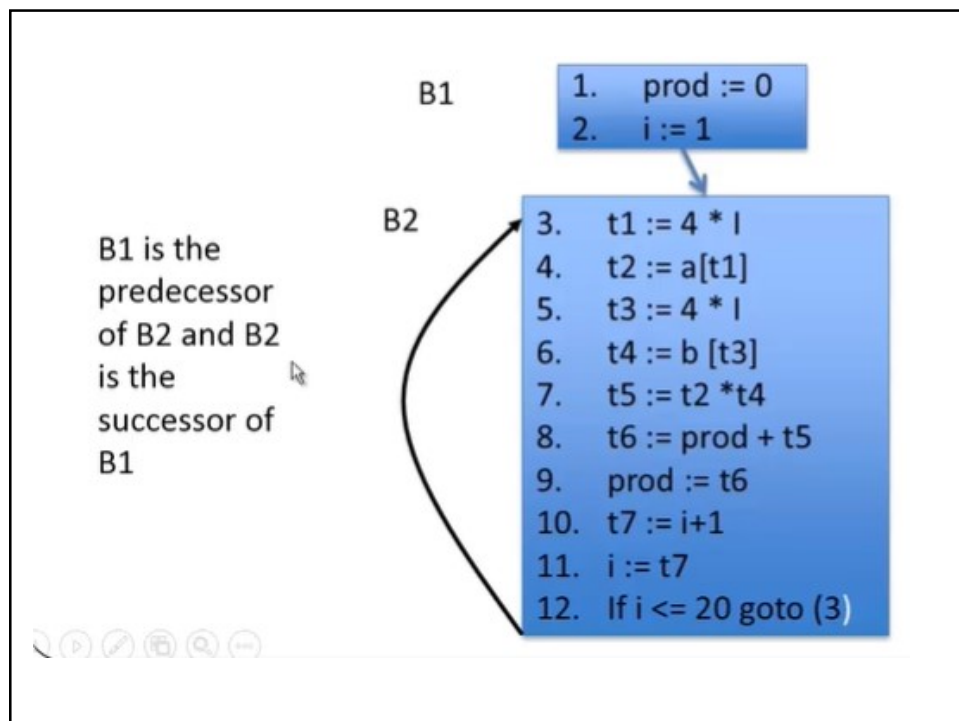


## EXAMPLE 2

<pre> Begin   prod := 0   i := 1;   do begin     prod := prod + a[i] *     b[i];     i = i + 1;   end   while i &lt;= 20 end </pre>	<pre> 1. prod := 0 2. i := 1 3. t1 := 4 * i 4. t2 := a[t1] 5. t3 := 4 * i 6. t4 := b[t3] 7. t5 := t2 * t4 8. t6 := prod + t5 9. prod := t6 10. t7 := i + 1 11. i := t7 12. If i &lt;= 20 goto (3) </pre>
---	--

## Identifying leaders

- (1) is the beginning – hence leader
- (3) is the target of a jump – hence leader
  - Lines (1) and (2) is a basic block
- Statement following (12) is a leader
  - Statements (3) to (12) is another basic block





## Next-Use Information.

- Knowing when the value of a variable will be used next is essential for **generating good code**.
- If the value of a variable that is currently in a register will never be referenced subsequently, then that **register can be assigned to another variable**.
- The use of a name in a three-address statement is defined as follows.
- Suppose three-address statement  $i$  assigns a value to  $x$ .
- If statement  $j$  has  $x$  as an operand, and control can flow from statement  $i$  to  $j$  along a path that has no intervening assignments to  $x$ , then we say statement  $j$  uses the value of  $x$  computed at statement  $i$ .
- We further say that  $x$  is live at statement  $i$ .

$i:$      $x = \dots$

·                    assuming there  
·                    are no assignments  
·                    to  $x$  between  $i$  and  $j$

$j:$      $\dots = x \text{ op } \dots$

Statement  $j$  uses  $x$ , and  $x$  is live at  $i$ .



### Algorithm for liveness and next use information

Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block  $B$  of three address instructions. Assume the symbol table initially shows all non-temporary variables in  $B$  as being live on exit.

OUTPUT: At each statement  $i: x = y + z$  in  $B$ , we attach to  $i$  the liveness and next-use information for  $x$ ,  $y$ , and  $z$ .

METHOD: We start at the last statement in  $B$  and scan backwards to the beginning of  $B$ . At each statement  $i: x = y + z$  in  $B$  do the following:

- 1) attach to statement  $i$  the information currently found in the symbol table regarding the next-use and liveness of  $x$ ,  $y$ , and  $z$ .
- 2) In the symbol table, set  $x$  to "not live" and "no next use".
- 3) In the symbol table, set  $y$  and  $z$  to "live" and the next uses of  $y$  and  $z$  to instruction  $i$ .

B

i: a = b + c

j: d = a + b

	a	b	c
Live	False	True	True
Nextuse	None	i	i

	a	b	c
Live	True	True	True
Nextuse	j	j	None

	a	b	d
Live	True	True	False
Nextuse	j	j	None

	a	b	d
Live	True	True	True
Nextuse	None	None	None

## LOOPS

Programming-language constructs like while-statements, do-while-statements, and for-statements naturally give rise to loops in programs. Since virtually every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops. Many code transformations depend upon the identification of “loops” in a flow graph. We say that a set of nodes  $L$  in a flow graph is a *loop* if

1. There is a node in  $L$  called the *loop entry* with the property that no other node in  $L$  has a predecessor outside  $L$ . That is, every path from the entry of the entire flow graph to any node in  $L$  goes through the loop entry.
2. Every node in  $L$  has a nonempty path, completely within  $L$ , to the entry of  $L$ .
  1.  $B_3$  by itself.
  2.  $B_6$  by itself.
  3.  $\{B_2, B_3, B_4\}$ .

The first two are single nodes with an edge to the node itself. For instance,  $B_3$  forms a loop with  $B_3$  as its entry. Note that the second requirement for a loop is that there be a nonempty path from  $B_3$  to itself. Thus, a single node like  $B_2$ , which does not have an edge  $B_2 \rightarrow B_2$ , is not a loop, since there is no nonempty path from  $B_2$  to itself within  $\{B_2\}$ .

The third loop,  $L = \{B_2, B_3, B_4\}$ , has  $B_2$  as its loop entry. Note that among these three nodes, only  $B_2$  has a predecessor,  $B_1$ , that is not in  $L$ . Further, each of the three nodes has a nonempty path to  $B_2$  staying within  $L$ . For instance,  $B_2$  has the path  $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2$ .  $\square$



## Code Optimization

- Elimination of unnecessary instructions in object code, or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called “code improvement” or “code optimization”.
- Types of optimization
  - 1: Machine independent optimization
  - 2: Machine dependent optimization
  - **Machine independent optimization:**
- The process of optimizing intermediate code instruction is called as machine independent optimization
  - Types of machine independent optimization
    - 1. Local optimization : optimization within each basic block by itself
    - 2. Global optimization : how information flows among the basic blocks of a program.
  - **1. Local optimization**
    - Local Common Subexpressions elimination
    - Dead Code Elimination
    - The Algebraic Optimization

## OPTIMIZATION OF BASIC BLOCKS

- **Optimization of Basic Blocks:**

- **Structure Preserving Transformation:**

- The DAG representation of basic blocks
- Finding Local common sub expressions
- Dead code elimination

- **Algebraic transformations**

- The use of Algebraic Identities
- Strength Reduction
- Constant folding

- A substantial improvement in the running time of code can be performed by **local optimization** within each block .
- More thorough **global optimization** which deals with how information flows among the basic blocks of a program

## The DAG Representation of Basic Blocks

We construct a DAG for a basic block as follows

Local Optimization begin by transforming basic block into DAG.

1. Node in DAG for each initial values of Variables
2. Node N associated with each Statement  $S$  within the block.
3. Node N is labeled by operator
4. Output nodes have Variables as live on exit.

## The DAG Representation of Basic Blocks

- Directed Acyclic Graphs
  - Determines common subexpressions
- 1) Leaves are labeled by variable names or constants. Initial values are subscripted with 0.
  - 2) Interior nodes are operators.
  - 3) Internal nodes also represent result of the expressions.

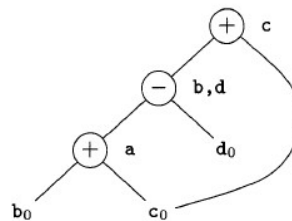
The DAG representation of a basic block lets us perform several code-improving transformations on the code represented by the block.

- a) We can eliminate *local common subexpressions*, that is, instructions that compute a value that has already been computed.
- b) We can eliminate *dead code*, that is, instructions that compute a value that is never used.
- c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
- d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

## 1. Finding Local Common Sub Expressions

$a = b + c$   
 $b = a - d$   
 $c = b + c$   
 $d = a - d$

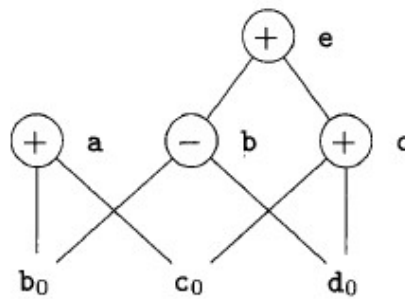
$a = b + c$   
 $d = a - d$   
 $c = d + c$



```

a = b + c;
b = b - d
c = c + d
e = b + c

```



## 2. Dead code elimination

- The code having 'no next use' or 'not live' are dead code
- Ex:  $x = a * b$

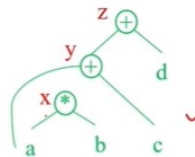
```

x = a * b
y = a + c
z = y + d

```

• 'x' have 'no next use' or 'not live'

- x is dead code
- After removal of dead code
- $y = a + c$
- $z = y + d$





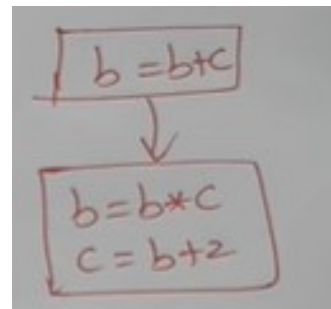
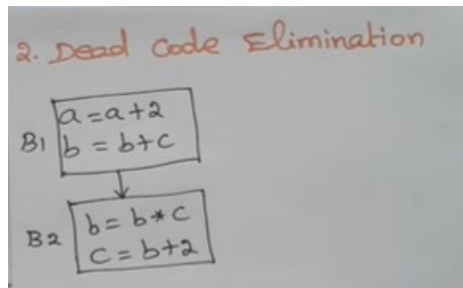
- Ex 2:

```

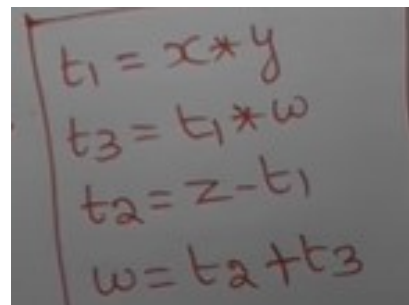
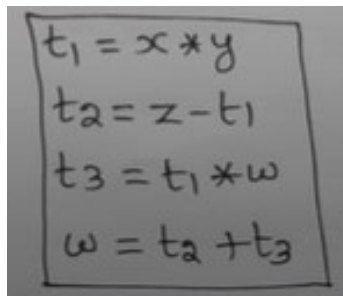
if(1)
    Print("TOC");
else
    print("TC");

```

**Ex 3:**



### 3. Reordering of statements



## Renaming of temporary variables

```

t1 = x * y
t2 = z - t1
t1 = t1 * w
w = t2 + t1

```

```

t1 = x * y
t2 = z - t1
t3 = t1 * w
w = t2 + t3

```

## Algebraic Transformations

**1. Algebraic identities**, another important class of optimizations on basic blocks. arithmetic identities, such as

$$x + 0 = 0 + x = x, \quad x - 0 = x$$

$$x * 1 = 1 * x = x, \quad x / 1 = x$$

**2. Strength reduction**, replacing a more expensive operator by a cheaper one.

*EXPENSIVE*      *CHEAPER*

$$x^2 = x * x$$

$$2 * x = x + x$$

$$x/2 = x * 0.5$$

**3. Constant folding**, Here we evaluate constant expressions at compile time and replace the constant expressions by their values.

$$2 * 3.14 \Rightarrow 6.28$$

```

for(i=0; i<=n; i++)
{
  X = 2*3 + y -> X = 6 + y
}

```

**4.algebraic Simplification**, Apply algebraic transformations such as commutativity and associativity

Ex 1:  $a = b + c$

$t = c + d$

$e = t + b$

•  $a = b + c$

•  $e = a + d$

If  $t$  is not needed outside this block, we can change this sequence to

•  $e = t + b$

$= c + d + b$

$= c + b + d$

$= b + c + d$

$= a + d$

Ex 2:  $x * y - x * z \Rightarrow x * (y - z)$

Ex 3:  $y = x + a$

$z = y - a$

$w = z * b$

Backward substitution

$w = z * b = (y - a) * b = (x + a - a) * b = x * b$

$w = x * b$

**5.Copy Propagation**, coping constant or variable from one statement to other .

Ex 1:  $x = 2$

$y = x * b$

### Representation of Array References

The proper way to represent array in a DAG is as follows.

1.  $x = a[i]$ , is represented by creating a node with operator  $[ ]$  and two children  $a$ , and index  $i$ . Variable  $x$  becomes a label of this new node.

2.  $a[j] = y$ , is represented by a new node with operator  $[ ] =$  and three children representing  $a$ ,  $j$  and  $y$ . There is no variable labeling this node.

What is different is that the creation of this node *kills* all currently constructed nodes whose value depends on  $a$ .

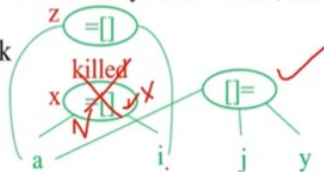
A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

Ex 1: The DAG for the basic block

$x = a[i]$

$a[j] = y$

$z = a[i]$



The node  $N$  for  $x$  is created first, but when the node labeled  $[ ] =$  is created,  $N$  is killed.

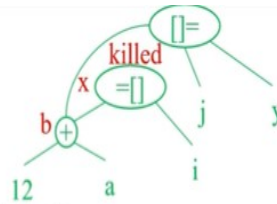
Thus, when the node for  $z$  is created, it cannot be identified with  $N$ , and a new node with the same operands  $a$  and  $i$  must be created instead.

Ex 2: The DAG for the basic block

$b = 12 + a$

$x = b[i]$

$b[j] = y$



A node can kill if it has a descendant that is an array.

If  $j$  and  $i$  represent the same value, then  $b[i]$  and  $b[j]$  represent the same location.

Therefore it is important to have the third instruction,  $b[j] = y$ , kill the node with  $x$  as its attached variable.

However, as both the killed node and the node that does the killing have  $a$  as a grandchild, not as a child.

Ex 3:

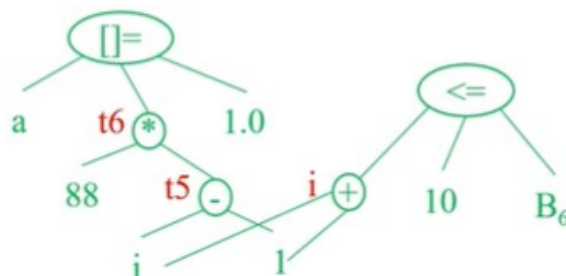
$t5 = i - 1$

$t6 = 88 * t5$

$a[t6] = 1.0$

$i = i + 1$

if  $i \leq 10$  goto  $B_6$



- **Machine independent code optimization** - Principle sources of Optimization- Causes of Redundancy, Running example: Quick Sort, Semantic Preserving transformations, Global common sub expressions, copy propagation, dead code elimination, code motion, induction variables, and reduction in strength.

### Code optimization

- Elimination of unnecessary instructions
- Replacement of one sequence of instructions by a faster sequence of instructions
- Local optimization
- Global optimizations
  - based on data flow analyses

Most global optimizations are based on *data-flow analyses*, which are algorithms to gather information about a program. The results of data-flow analyses all have the same form: for each instruction in the program, they specify some property that must hold every time that instruction is executed. The analyses differ in the properties they compute.

## Principle sources of Optimization

- A compiler optimization **must preserve the semantics of the original program.**
- once a programmer chooses and implements a particular algorithm, the compiler cannot understand enough about the program to replace it with a substantially different and more efficient algorithm.
- A compiler knows only how to apply relatively **low-level semantic transformations**, using general facts such as algebraic identities like  $i + 0 = i$  or program semantics such as the fact that performing the same operation on the same values yields the same result.

## Causes of redundancy

- Redundant operations are
  - at the source level
  - a side effect of having written the program in a high-level language
- Each of high-level data-structure accesses expands into a number of low-level arithmetic operations
- Programmers are not aware of these low-level operations and cannot eliminate the redundancies themselves.
- By having a compiler eliminate the redundancies
  - The programs are both efficient and easy to maintain.

## A Running Example: Quicksort

```

void quicksort(int m, int n)
    /* recursively sorts a[m] through a[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; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}

```

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

Figure 9.2: Three-address code for fragment in Fig. 9.1



Notice that every array access in the original program translates into a pair of steps, consisting of a multiplication and an array-subscripting operation. As a result, this short program fragment translates into a rather long sequence of three-address operations.

- $X = a[i]$ 
  - $T6 = 4 * i$
  - $X = a[t6]$
- $A[j] = x$ 
  - $T10 = 4 * j$
  - $A[t10] = x$

## Semantics-Preserving Transformations

- A number of ways in which a compiler can improve a program without changing the function it computes
  - Common-sub expression elimination
  - Copy propagation
  - Dead-code elimination
  - Constant folding

## Common Subexpressions

- Common subexpression
  - Previously computed
  - The values of the variables not changed
- Local:

```

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

```

B<sub>5</sub>

```

t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2

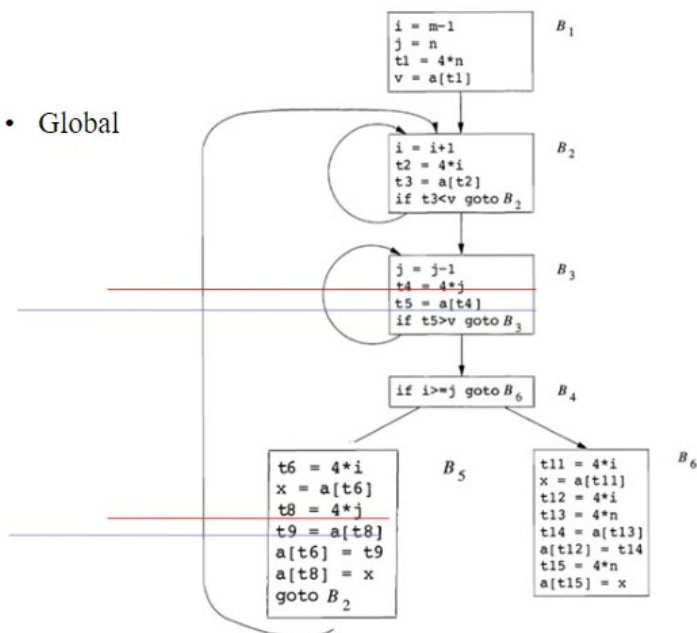
```

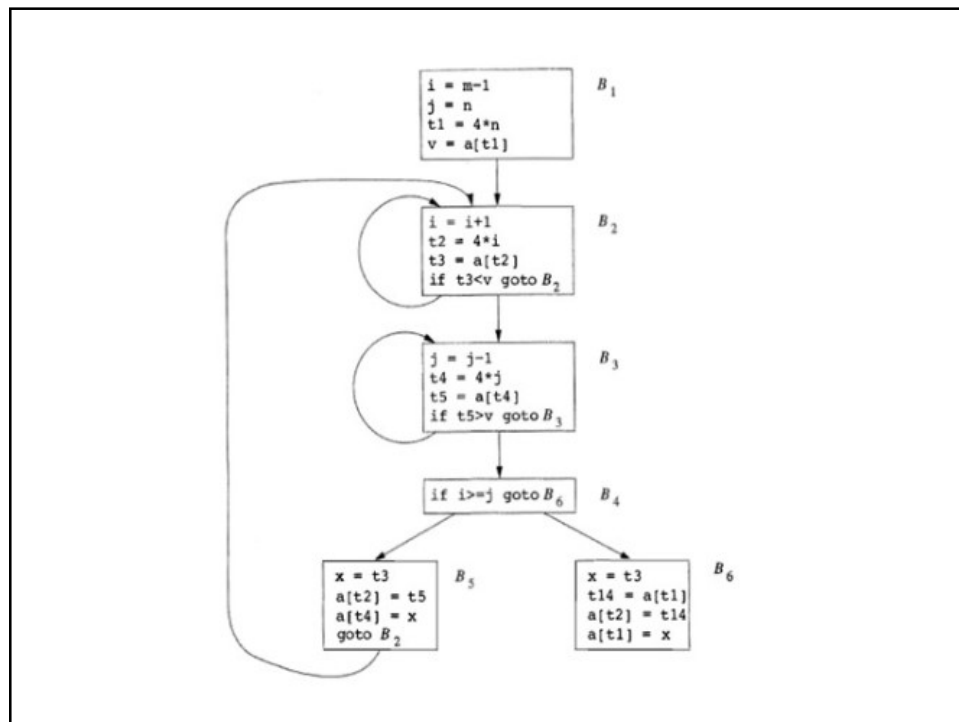
B<sub>5</sub>

(a) Before.

(b) After.

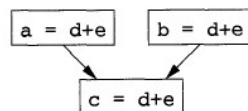
- Global



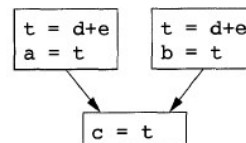


## 2. Copy Propagation

- Assignments of the form  $u = v$  called copy statements, or copies for short.
  - copy-propagation transformation is to use  $v$  for  $u$ .
  - Block  $B_5$  after copy propagation
- |                |                |
|----------------|----------------|
| • $x = t3$     | • $x = t3$     |
| • $a[t2] = t5$ | • $a[t2] = t5$ |
| • $a[t4] = x$  | • $a[t4] = t3$ |
| • goto B2      | • goto B2      |



(a)




(b)

Figure 9.6: Copies introduced during common subexpression elimination


### 3. Dead-Code Elimination


- A variable is *live* at a point in a program if its value can be used subsequently; otherwise, it is *dead* or *no next use* at that point.
- A *dead* (or *useless*) code statements compute values that never get used.
- Ex : *if (debug) print ...*  
`debug = FALSE`
- If copy propagation replaces `debug` by `FALSE`, then the `print` statement is dead because it cannot be reached.
- One advantage of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to `x` and transforms the code into

<pre>x = t3 a[t2] = t5 a[t4] = x goto B2</pre>		<pre>• x = t3 • a[t2] = t5 • a[t4] = t3 • goto B2</pre>	<pre>a[t2] = t5 a[t4] = t3 goto B2</pre>
--	---	---	--

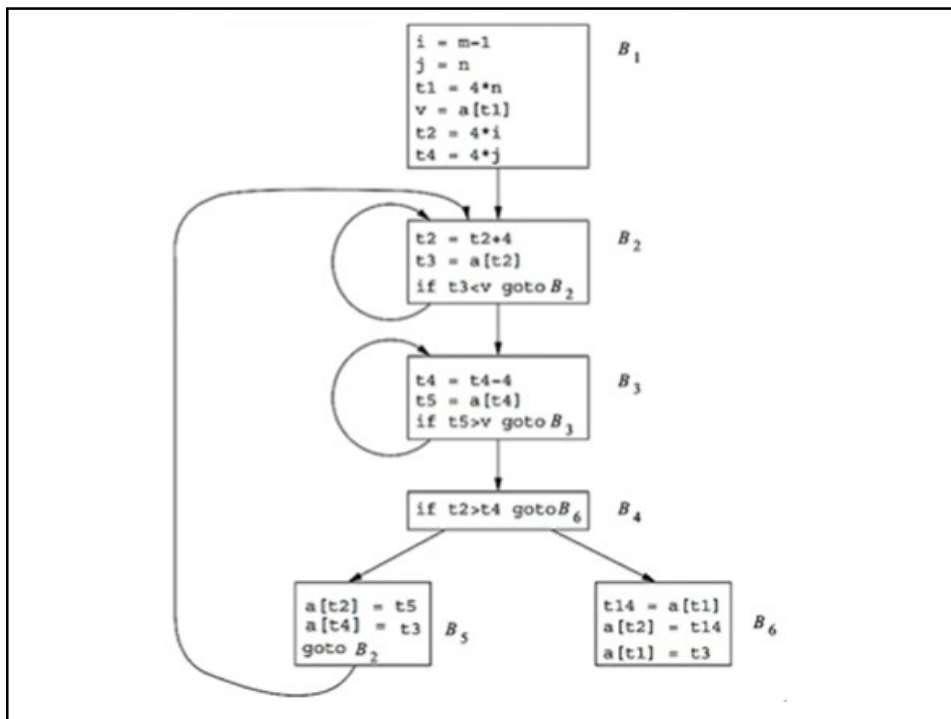
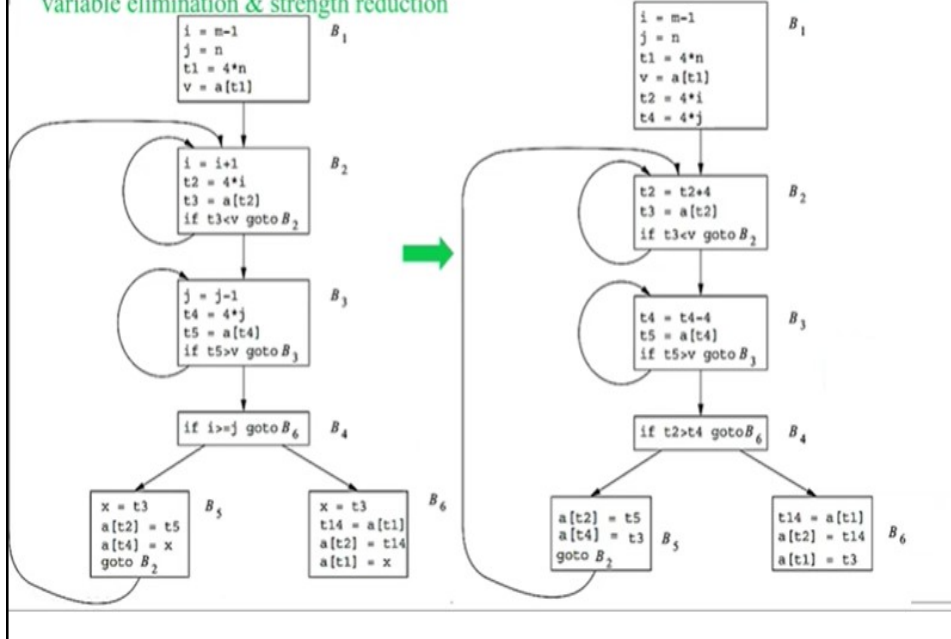
### 4. Code Motion

- Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time.
- The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Modification that decreases the amount of code in a loop is *code motion*.
- Ex 1:
 

<pre>• while (i &lt;= limit-2)</pre>		<pre>• t = limit-2 • while (i &lt;= t)</pre>
--------------------------------------	---	--
- Now, the computation of `limit-2` is performed once, before we enter the loop.

<pre>• Ex 2: • int i=1, a=2; • while (i &lt;= 50) • { •     int j=i+a*3; •     printf(j); •     i++; • }</pre>		<pre>• int i=1, a=2; • int b=a*3; • while (i &lt;= 50) • { •     int j=i+b; •     printf(j); •     i++; • }</pre>
--	---	---

- Flow graph after Copy Propagation, Dead-Code Elimination, Code Motion, induction-variable elimination & strength reduction



- **Machine dependent code optimization:**
- Peephole optimization: Eliminating redundant loads and stores – Eliminating Unreachable code – Flow-of-control optimizations – Algebraic simplification and Reduction in strength - Use of Machine idioms
- Register allocation: Global Register Allocation

- **Machine-independent optimization** phase tries to improve the intermediate code to obtain a better output.
- The optimized intermediate code does not involve any absolute memory locations or CPU registers.
- **Machine-dependent optimization** is done after generation of the target code which is transformed according to target machine architecture.
- This involves CPU registers and may have absolute memory references.

## Peephole optimization

- A simple but effective technique for locally improving the target code is *peephole optimization*,
  - which is done by examining a sliding window of target instructions (called the *peephole*) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible..
- Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation

- The peephole is a small, sliding window on a program.
- The code in the peephole need not be contiguous, although some implementations do require this.
- characteristic of peephole optimizations:
  - Redundant-instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms



## Eliminating Redundant Loads and Stores

- If the target code contains the instruction sequence:

```
MOV R, a
MOV a, R
```

- **Instruction 2** can always be removed if it **does not have a label**.
- If it is **labeled**, there is no guarantee that step 1 will always be executed before step 2.

```
goto L1
...
...
MOV R, a
L1: MOV a, R
```

## 2. Eliminating Unreachable code/unnecessary jumps

- Another opportunity for peephole optimization is the removal of unreachable instructions.
- An unlabeled instruction immediately following an unconditional jump may be removed.
- This operation can be repeated to eliminate a sequence of instructions.
- For example, for debugging purposes, a large program may have within it certain code fragments that are executed only if a variable debug is equal to 1.

- In the intermediate representation, this code may look like

```

        if debug == 1 goto L1
        goto L2
    L1: print debugging information
    L2:

```

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of `debug`, the code sequence above can be replaced by

```

        if debug != 1 goto L2
        print debugging information
    L2:

```

If `debug` is set to 0 at the beginning of the program, constant propagation would transform this sequence into

```

        if 0 != 1 goto L2
        print debugging information
    L2:

```

Now the argument of the first statement always evaluates to *true*, so the statement can be replaced by `goto L2`. Then all statements that print debugging information are unreachable and can be eliminated one at a time.

### 3. Flow-of-Control Optimizations

- Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps.
- These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.

```

        goto L1
        ...
    L1: goto L2

```

by the sequence

```

        goto L2
        ...
    L1: goto L2

```

```

        if a < b goto L1
        ...
L1: goto L2

```

can be replaced by the sequence

```

        if a < b goto L2
        ...
L1: goto L2

```

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

```

        goto L1
        . . .
L1: if a < b goto L2
L3:

```

may be replaced by the sequence

```

        if a < b goto L2
        goto L3
        . . .
L3:

```

#### 4. Algebraic simplification and Reduction in strength

- We discussed algebraic identities that could be used to simplify DAG's. These algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as
- $x = x + 0$       or       $x = x * 1$       in the peephole.

## Reduction-in-strength

- Similarly, reduction-in-strength transformations can be applied in the peep-hole to replace expensive operations by equivalent cheaper ones on the target machine.
- Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For ex-ample,  $x^2$  is invariably cheaper to implement as  $x * x$  than as a call to an exponentiation routine.
- Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be approximated as multiplication by a constant, which may be cheaper.

## 5.Use of Machine idioms

- The target machine may have hardware instructions to implement certain specific operations efficiently.
- Detecting situations that permit the use of these instructions can reduce execution time significantly.
- For example, some machines have auto-increment and auto-decrement addressing modes.
- These add or subtract one from an operand before or after using its value.
- The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing.
- These modes can also be used in code for statements like  $x = x + 1$ .

## Register Allocation and Assignment

- Instruction involving only register operands are shorter and faster than those involving memory operands. This also means that proper use of register help in generating the good code. This section presents various strategies for deciding what values in a program should reside in registers(register allocation)and in which register each value should reside (register assignment).
- One approach to register allocation and assignment is to assign specific value in an object program to certain registers.
- This approach has the advantage that it simplifies the design of a compiler.

- Disadvantage is that , applied too strictly , it uses registers inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated.
- Now we will discuss various strategies used in register and assignment and those
  - Global Register Allocation
  - Usage Counts
  - Register Assignment for Outer Loops
  - Register Allocation by Graph Coloring

## Global Register Allocation

- Generating the code the registers are used to hold the value for the duration of single block.
- All the live variables are stored at the end of each block.
- For the variables that are used consistently we can allocate specific set of registers.
- Hence allocation of variables to specific registers that is consistent across the block boundaries is called global register allocation.

## Following are the strategies adopted while doing the global register allocation

- The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop
- Another strategy is to assign some fixed number of global registers to hold the most active value in each inner loop.
- The registers not already allocated may be used to hold values local to one block
- In certain language like C or Bliss programmer can do the register allocation by using register declaration.