Code Optimization

Code Optimization

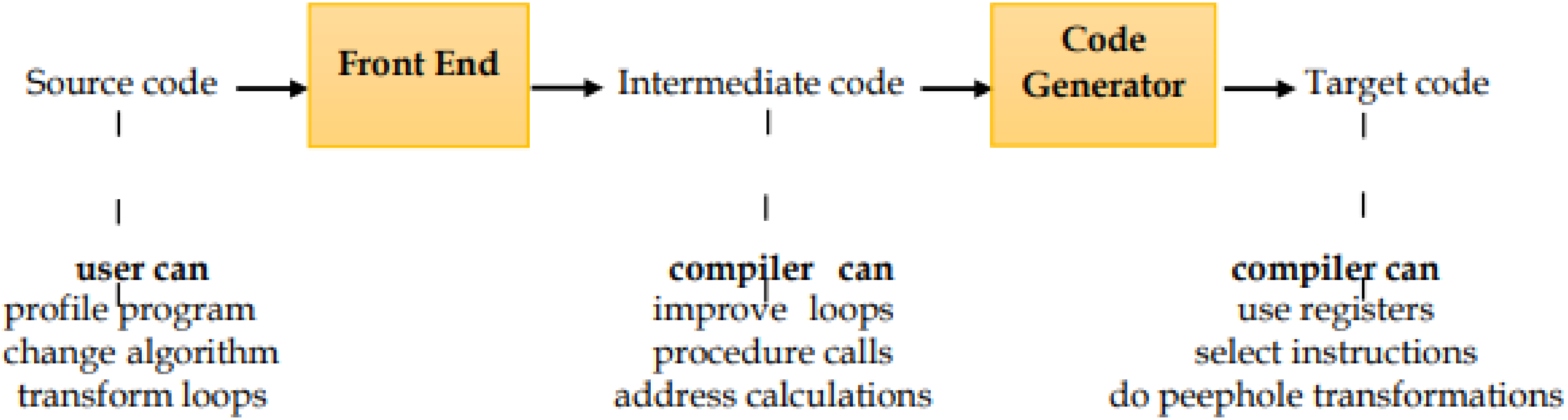
- Optimization is a program transformation technique that helps to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.
- In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.

Important Points

- Retain the semantics of the source code.
- Reduce time and/ or space.
- Reduce the overhead involved in the optimization process.

Optimization can be applied in 3 places.

- Source code
- Intermediate code
- Target code



Types of optimization techniques

- Machine independent optimization
- Machine dependent optimization

Machine Independent Optimization

- 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

- 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.

Steps before optimization

- Source program should be converted to Intermediate code
- Basic blocks construction
- Generating flow graph
- Apply optimization

Basic Block

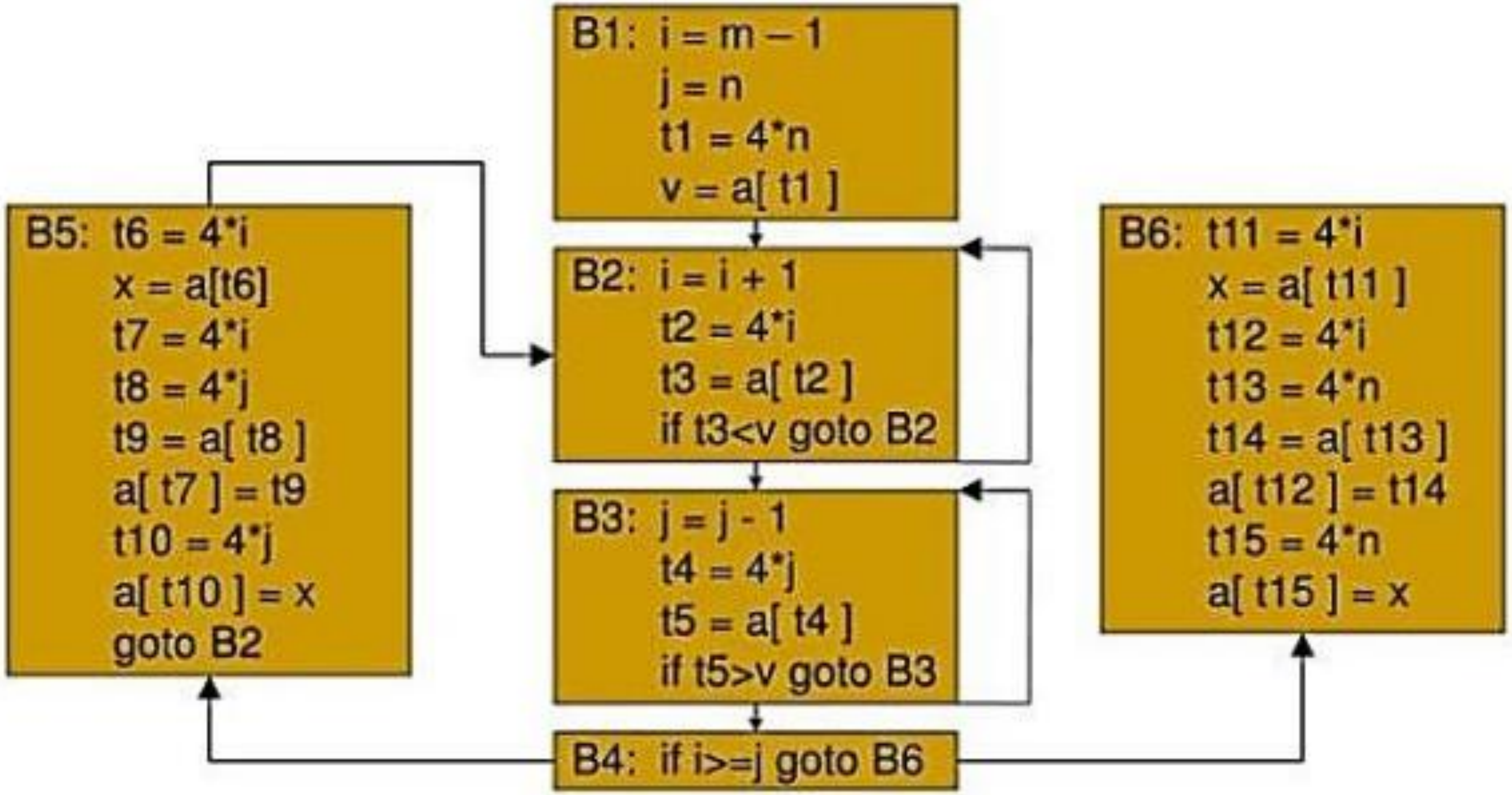
- Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code.
- These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.
- A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCHCASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Identification of basic blocks

- Search header statements of all the basic blocks from where a basic block starts. Following specifications denotes the header statement:
  - First statement of a program.
  - Statements that are target of any branch (conditional/unconditional).
  - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

- Basic blocks are important concepts from both code generation and optimization point of view.
- Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block.
- If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Flow Graph of Quicksort



```
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
```

Machine Independent Optimization

- The elimination of common sub-expressions
- Copy Propagation
- Constant Folding
- Dead code elimination
- Loop invariant code motion
- Strength reduction
- Procedure Inlining

Elimination of common sub-expressions

- Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it.
- Example :

```
a=b*c;
d=b*c+x-y;
```

We can eliminate the second evaluation of b*c from this code if none of the intervening statements has changed its value. Rewrite the code as

```
t1=b*c;
a=t1;
d=t1+x-y;
```

Contd.

Example

```
a=b*c;
b=x;
d=b*c+ x-y;
```

In above code, we can not eliminate the second evaluation of b*c because the value of b is changed due to the assignment b=x before it is used in calculating d.

Contd.

- Two expressions are common if they lexically equivalent i.e., they consist of identical operands connected to each other by identical operator.
- They evaluate the identical values i.e., no assignment statements for any of their operands exist between the evaluations of these expressions.
- The value of any of the operands use in the expression should not be changed even due to the procedure call.

Example :

```
        c=a*b;
        x=a;
        d=x*b;
```

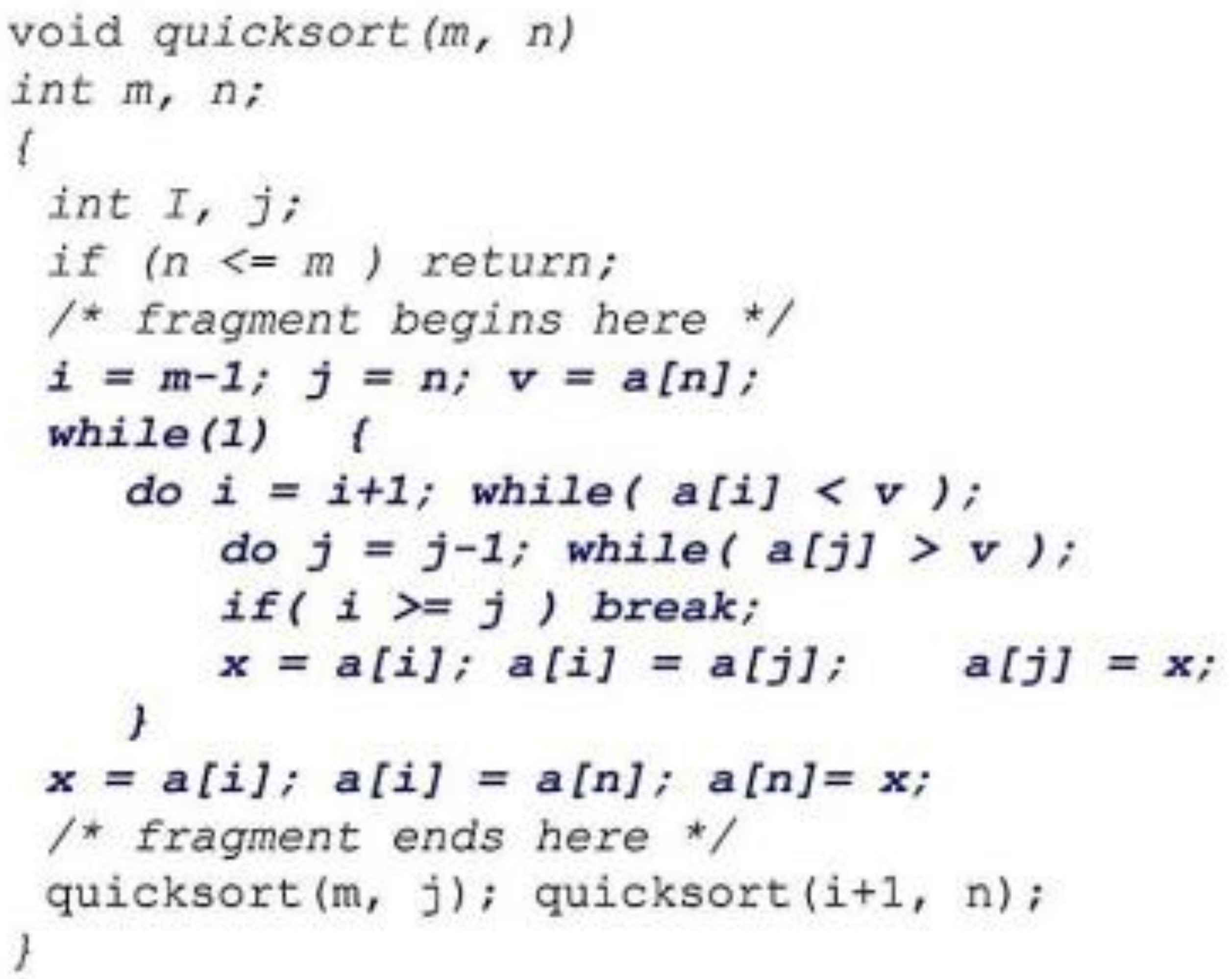
We may note that even though expressions a*b and x*b are common in the above code, they can not be treated as common sub expressions.

Types of common sub expression elimination

We can avoid recomputing the expression if we can use the previously computed value.

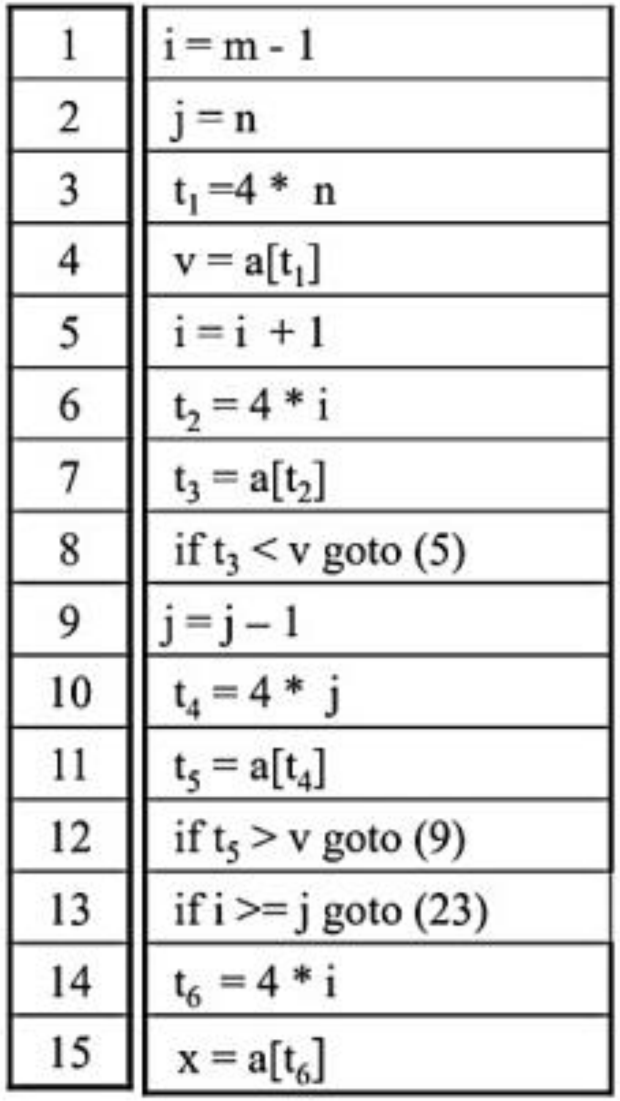
- Local common sub expression elimination
- Global common sub expression elimination

Quick Sort Code

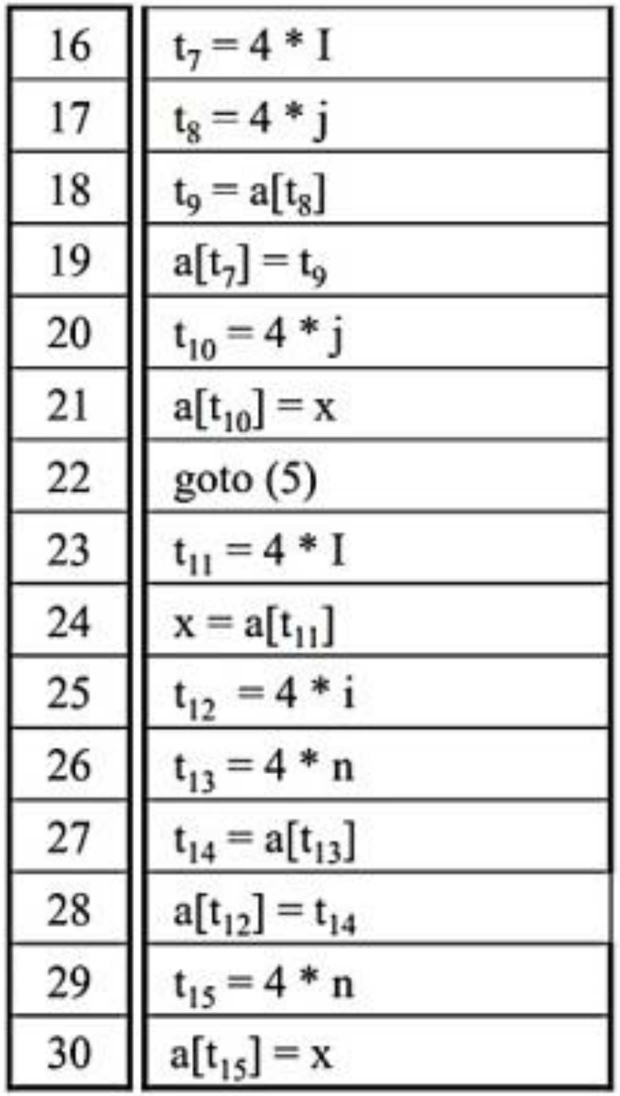


```
void quicksort(m, n)
int m, n;
{
 int I, j;
 if (n <= m ) return;
 /* fragment begins here */
 i = m-1; j = n; v = a[n];
 while(1)   {
    do i = i+1; while( a[i] < v );
        do j = j-1; while( a[j] > v );
        if( i >= j ) break;
        x = a[i]; a[i] = a[j];    a[j] = x;
    }
 x = a[i];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

Three Address Code for Quicksort

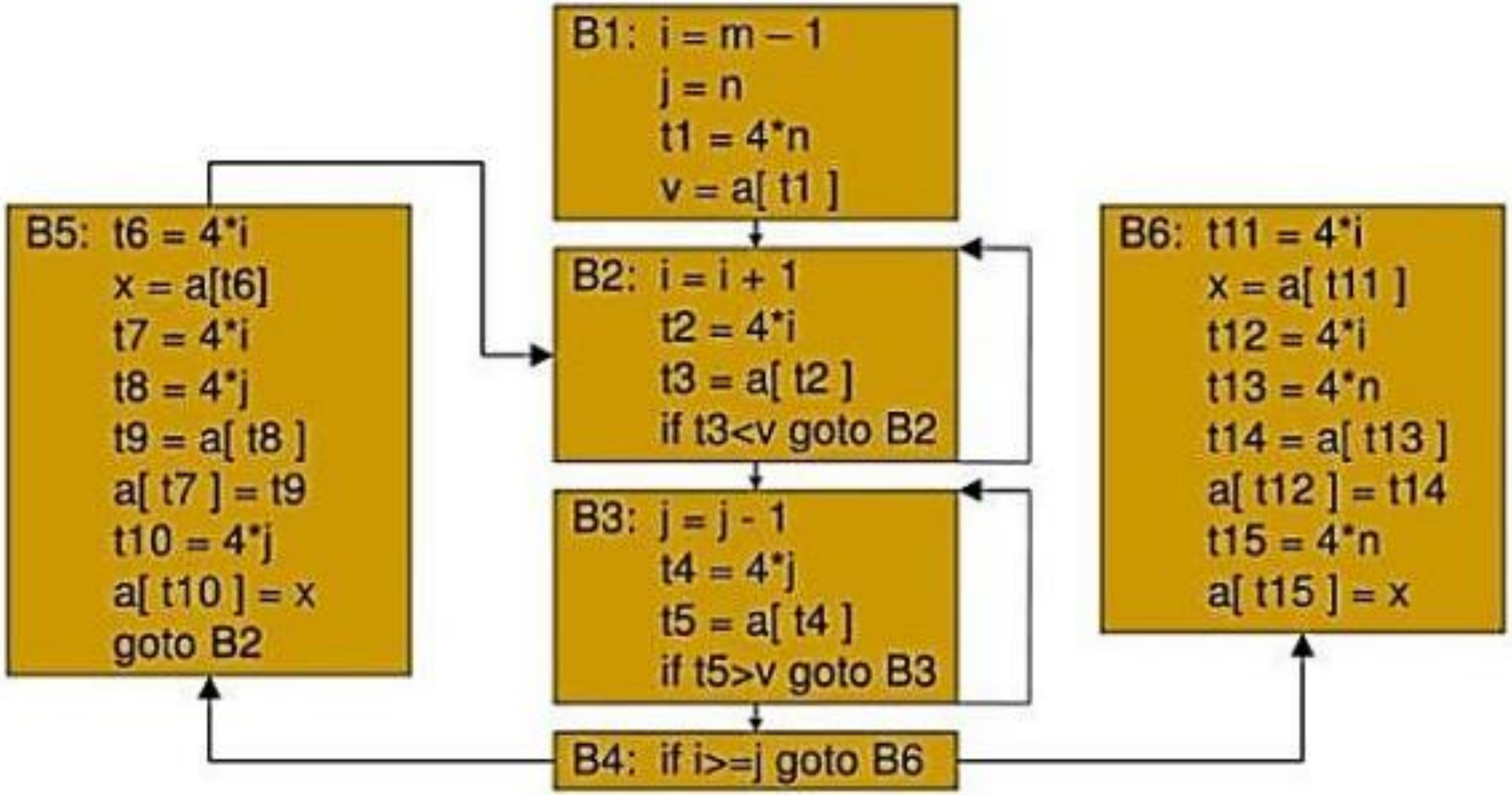


1	i = m - 1
2	j = n
3	t_1 =4 * n
4	v = a[t_1]
5	i = i + 1
6	t_2 = 4 * i
7	t_3 = a[t_2]
8	if t_3 < v goto (5)
9	j = j – 1
10	t_4 = 4 * j
11	t_5 = a[t_4]
12	if t_5 > v goto (9)
13	if i >= j goto (23)
14	t_6 = 4 * i
15	x = a[t_6]



16	t_7 = 4 * I
17	t_8 = 4 * j
18	t_9 = a[t_8]
19	a[t_7] = t_9
20	t_{10} = 4 * j
21	a[t_{10}] = x
22	goto (5)
23	t_{11} = 4 * I
24	x = a[t_{11}]
25	t_{12} = 4 * i
26	t_{13} = 4 * n
27	t_{14} = a[t_{13}]
28	a[t_{12}] = t_{14}
29	t_{15} = 4 * n
30	a[t_{15}] = x

Basic Blocks and Flow Graph for quick sort



```
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
```

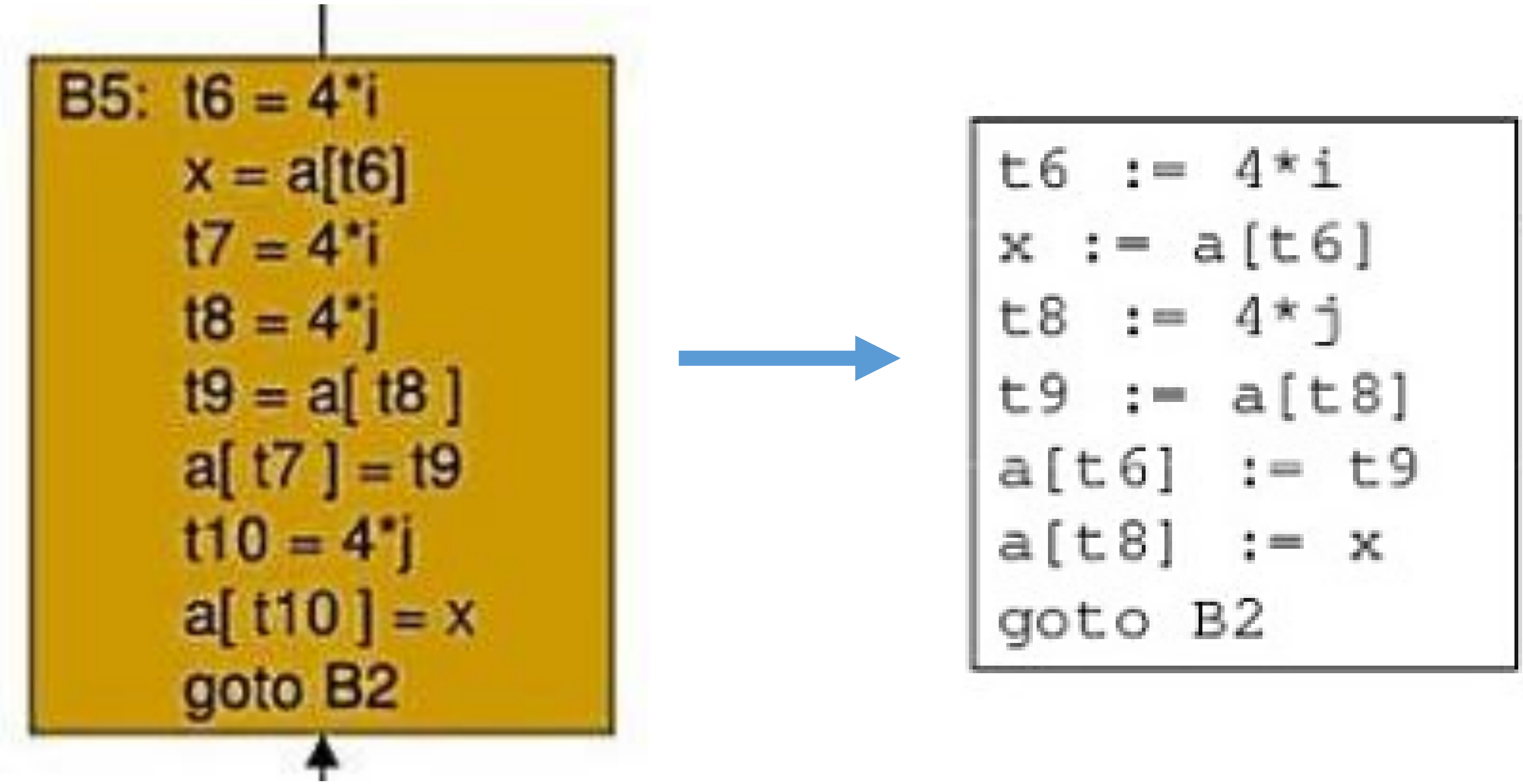
```
B3: j = j - 1
    t4 = 4*j
    t5 = a[ t4 ]
    if t5>v goto B3
```

```
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
```

Local common sub expression elimination



```
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
```

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

Global common sub expression elimination

After local common subexpressions are eliminated, B5 still evaluates 4 * i and 4 * j, as shown in local common subexpression elimination.

```
t8 = 4 * j
t9 = a[t8]
a[t8] =x
```

can be replaced by:

```
t9 = a[t4]
a[t4] = x
```

using t4 computed in block B3

- In Flow graph, as control passes from the evaluation of 4 * j in B3 to B5, there is no change to j and no change to t4, so t4 can be used if 4 * j is needed.
- Another common subexpression comes to light in B5 after t4 replaces t8. The new expression a[t4] corresponds to the value of a[j] at the source level.
- Not only does j retain its value as control leaves B3 and then enters B5, but a[j], a value computed into a temporary t5, does too, because there are no assignments to elements of the array a in the interim.
- The statement:

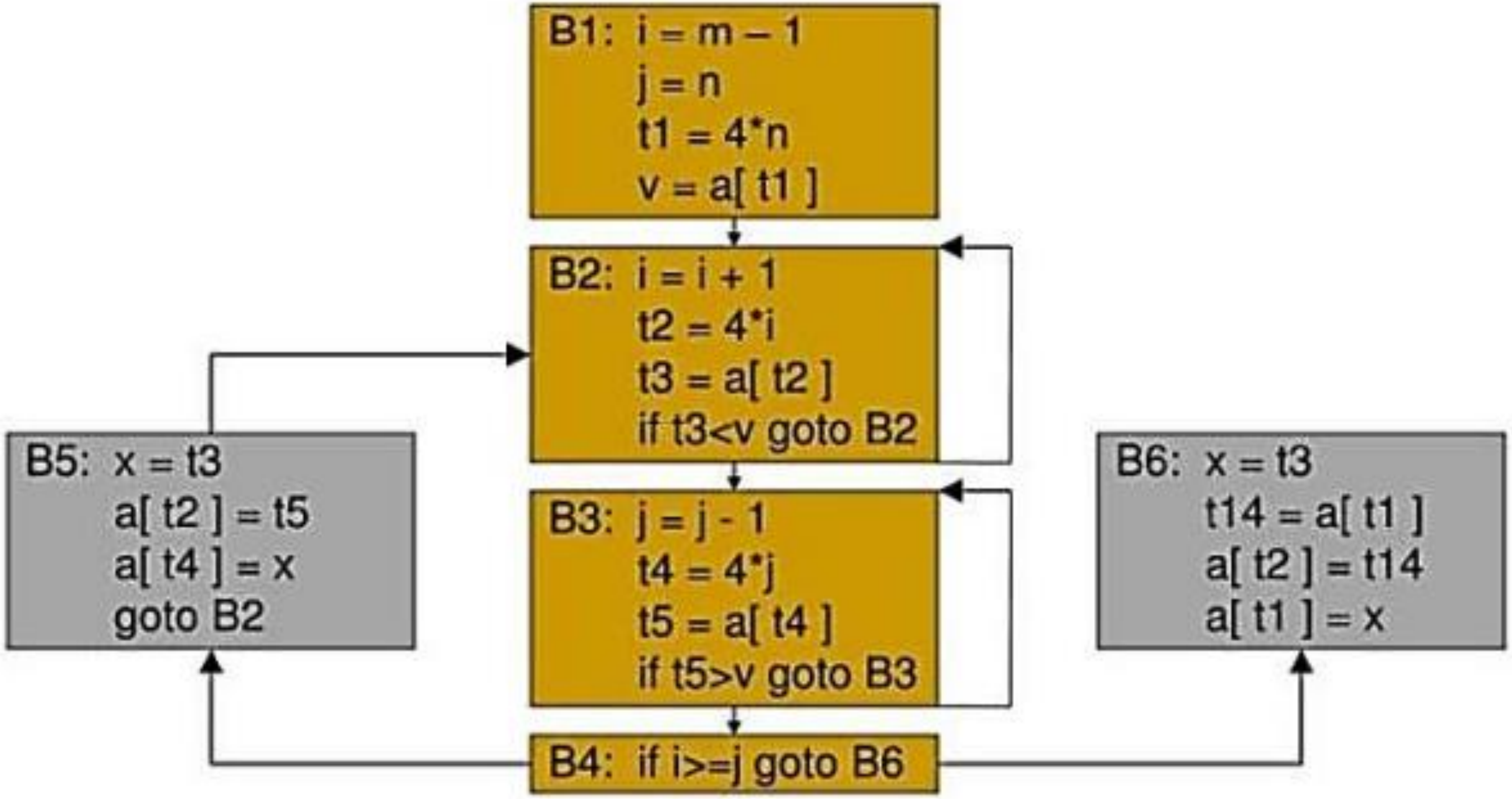
  t9 = a[t4]

  a[t6] = t9 in B5 therefore can be replace by a[t6] = t5

the same as the value assigned to t3 in block B2.

- Block B5 is the result of eliminating common sub expressions corresponding to the values of the source level expressions a[i] and a[j] from.
- A similar series of transformations has been done to B6 in Flow graph. The expression a[t1] in blocks B1 and B6 is not considered a common sub expression, although t1 can be used in both places.
- After control leaves B1 and before it reaches B6, it can go through B5, where there are assignments to a. Hence, a[t1] may not have the same value on reaching B6 as it did on leaving B1, and it is not safe to treat a[t1] as a common sub expression

B5 and B6 after common subexpression elimination



```
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: x = t3
    a[ t2 ] = t5
    a[ t4 ] = x
    goto B2
```

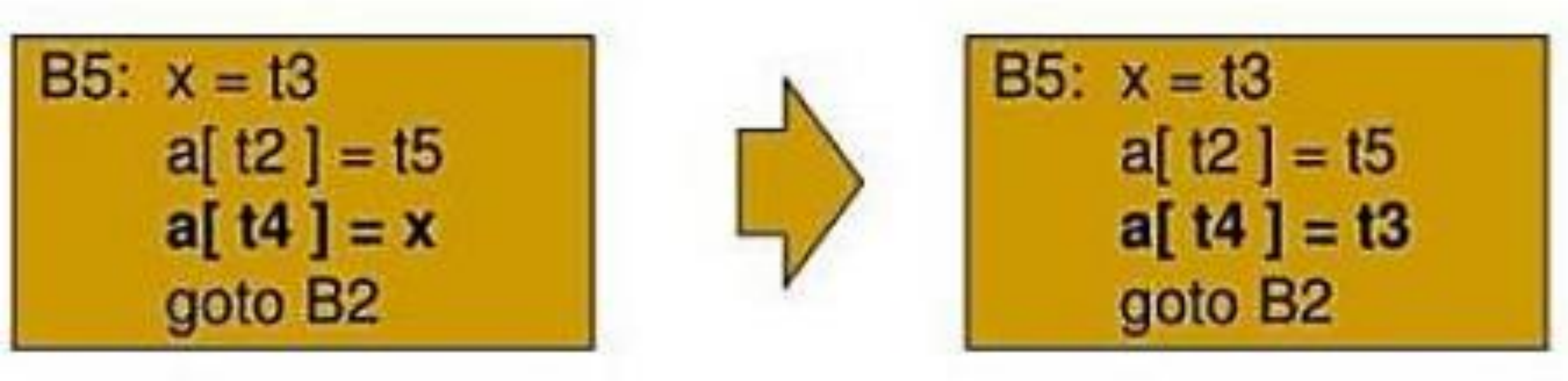
```
B6: x = t3
    t14 = a[ t1 ]
    a[ t2 ] = t14
    a[ t1 ] = x
```

Copy Propagation

- Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g.
- Copy propagation means use of one variable instead of another. Copy statements introduced during common subexpression elimination.

Copy Propagation

- The assignment x : = t3 in block B5 of Flow graph is a copy.



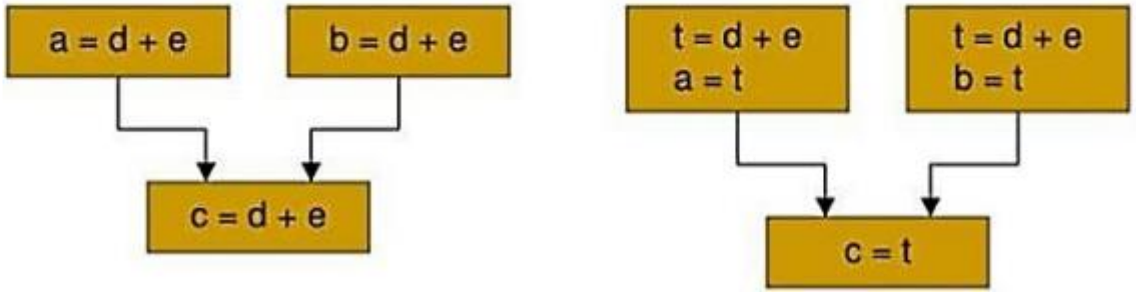
```
B5:  x = t3
     a[ t2 ] = t5
     a[ t4 ] = x
     goto B2
```

```
B5:  x = t3
     a[ t2 ] = t5
     a[ t4 ] = t3
     goto B2
```

This change may not appear to be an improvement, but it gives the opportunity to eliminate the assignment to x.

Example

- When the common subexpression in c := d+e is dominated, the algorithm uses a new variable t to hold the value of d+e.
- Since control may reach c := d+e either after the assignment to a or after the assignment to b, it would be incorrect to replace c := d+e by either c := a or by c := b.

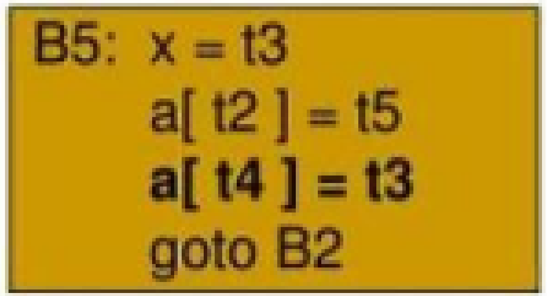


- One advantage of copy propagation is that it often turns the copy statement into dead code.

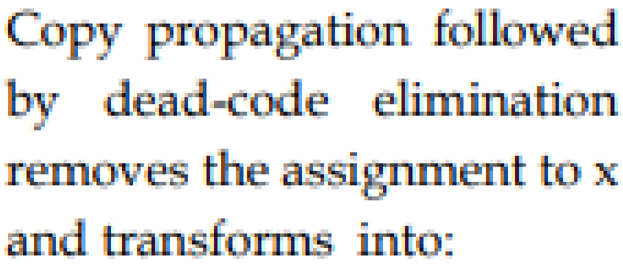
Dead-Code Elimination

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
- A related idea is dead or useless code, statements that compute values that never get used.
- While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

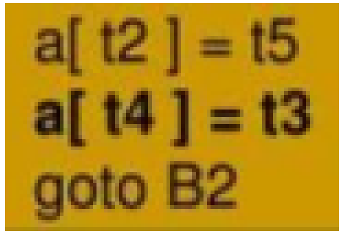
Example



```
B5:  x = t3
     a[ t2 ] = t5
     a[ t4 ] = t3
     goto B2
```



Copy propagation followed by dead-code elimination removes the assignment to x and transforms into:



```
a[ t2 ] = t5
a[ t4 ] = t3
goto B2
```

Example

```
i=0;
if(i==1)
{
        a=b+5;
}
```

Here, ‘if’ statement is dead code because this condition will never get satisfied.

Constant Folding

- If all operands are constants in an expression, then it can be evaluated at compile time itself. The result of the operation can replace the original evaluation in the program.
- This will improve the run time performance and reducing code size by avoiding evaluation at compile- time.

Example

a=3.14157/2 can be replaced by a=1.570

thereby eliminating a division operation.

Loop Optimization

The running time of a program may be improved if the number of instruction in an inner loop is decreased, even if we increase the amount of code outside the loop.

Mainly 3 techniques are there :-

- Code Motion
- Induction Variable
- Reduction In Strength

Code Motion

- An important modification that decreases the amount of code in a loop is code motion.
- Execution time of a program can be reduced by moving code from a part of a program which is executed very frequently to another part of the program which is executed fewer times
- Ex: Loop optimization – loop invariant code motion
- A fragment of code that resides in the loop and computes the same value of each iteration is called loop invariant code.

Example

```
for i = 1 to 100 begin
{
        z := i;
        x := 25 * a ;
        y := x + z ;
        end;
}
```

After Optimization

```
x := 25 * a ;
for i = 1 to 100 begin
{
        z := i;
        y := x + z ;
        end;
}
```

Example

- Evaluation of limit-2 is a loop-invariant computation in the following while-statement:

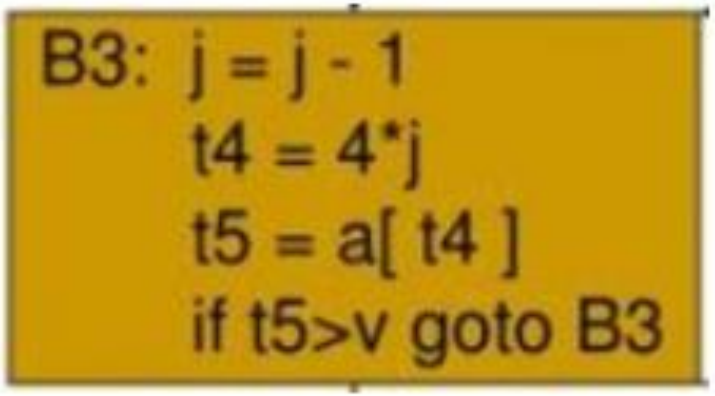
```
while (i <= limit - 2) /* statement does not change limit*/
```

- Code motion will result in the equivalent of

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

Induction Variables

- Loops are usually processed inside out. For example the loop around B3:



```
B3:  j = j - 1
     t4 = 4*j
     t5 = a[ t4 ]
     if t5>v goto B3
```

- Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables.

Strength Reduction

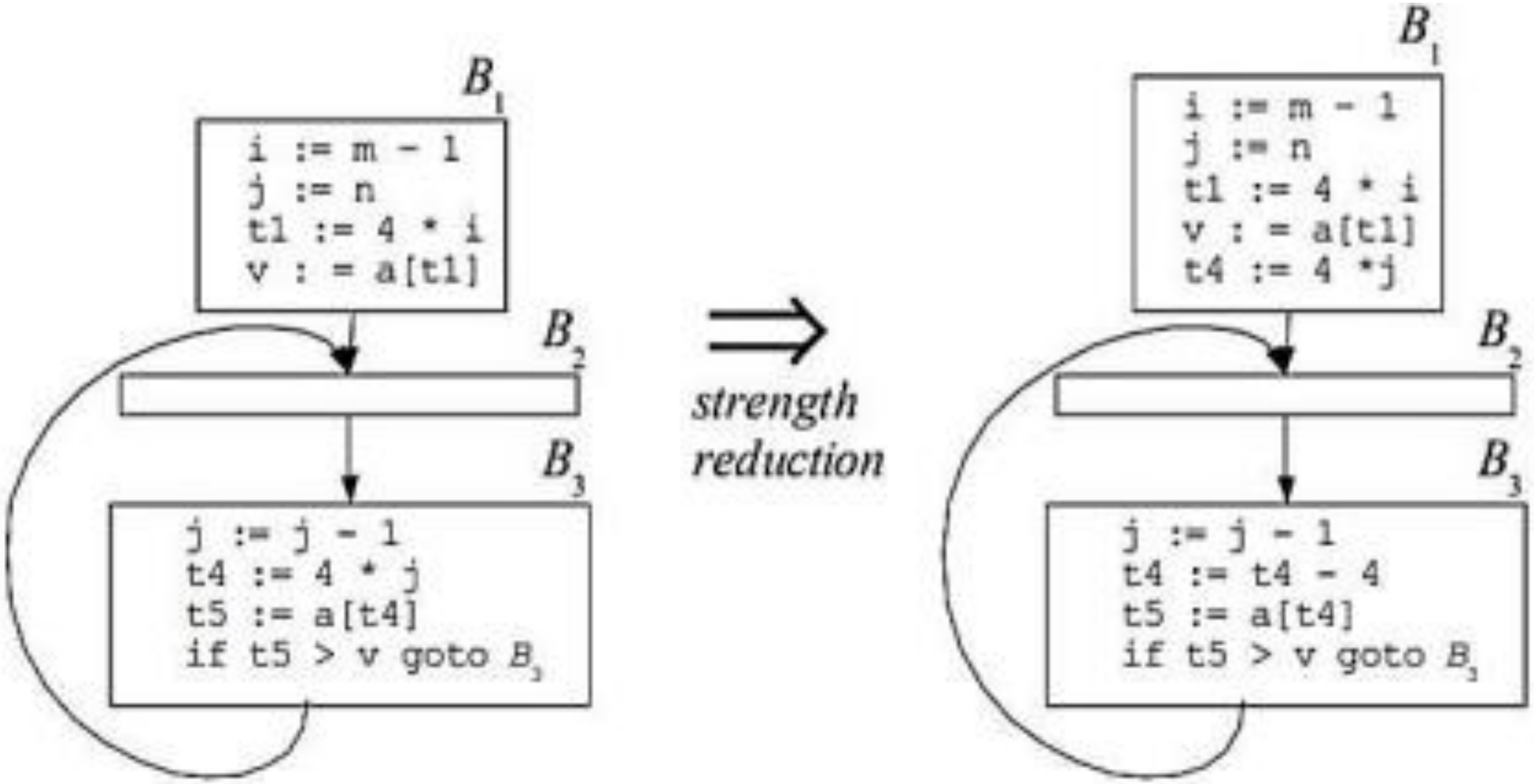
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination.
- For the inner loop around B3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.
- However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example

- As the relationship t4:=4*j surely holds after such an assignment to t4 and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold.
- We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time.
- Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, addition to block B1.

After Optimization

- This transformation involves replacement of expensive operation by cheaper one.



```
i := m - 1
j := n
t1 := 4 * i
v : = a[t1]
```

```
j := j - 1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
```

```
i := m - 1
j := n
t1 := 4 * i
v : = a[t1]
t4 := 4 *j
```

```
j := j - 1
t4 := t4 - 4
t5 := a[t4]
if t5 > v goto B3
```

Procedure Inlining

- Procedure calls are relatively expensive.
- One way to remove procedure call is by procedure inlining.
- This involves replacing procedure call with the code of procedure body.

The Use of Algebraic Identities

$x + 0 = 0 + x = x$

$x - 0 = x$

$x * 1 = 1 * x = x$

$x / 1 = x$

Algebraic optimization includes reduction in strength.

$x ** 2 = x * x$

$2 * x = x + x$

$x / 2 = x * 0.5$

Code Generation

Code Generation

- The final phase in the compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

Code Generation

- Input to code generator
- Target program
- Memory management
- Instruction selection
- Register allocation
- Evaluation order
- Approaches to code generation

Input to the code generator

- The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the intermediate representation.
- There are several choices for the intermediate language including
    - postfix notation,
    - three address representation such as quadruple, and
    - graphical representations such as syntax trees and Dags.
- The assumption is that prior to code generation the front end scanned, parsed and translated the source program into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language, type checking has taken place, so type conversion operators have been inserted wherever necessary.
- The code generation phase can therefore proceed on the assumption that its input is free of errors.

Target programs

- The output of the code generator is the target program. This output may take on a variety of forms- absolute machine language, relocatable machine language or assembly language.
- Producing an absolute machine language as output has the advantage that it can be placed in a fixed location in memory and intermediate executed.
- Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.
- Producing an assembly language program as output makes the process of code generation somewhat easier. It generate symbolic instructions and use the macro facilities of the assembler to help generate code.

- The instruction set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high quality machine code. The most common target machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer) and stack based.
- The RISC machine has many registers, three-address instructions, simple addressing modes and a relatively simple instruction set architecture.
- In contrast, a CISC machine has few registers, two-address instructions, a variety of addressing modes, several register classes, variable length instructions and instructions with side effects.

- In stack based machine, operations are done by pushing operands onto the stack and then performing the operations on the operands at the top of the stack. To achieve high performance, the top of the stack is typically kept in registers.
- Stack based architectures were revived with the introduction of Java Virtual Machine (JVM). The JVM is a software interpreter for java bytecodes, an intermediate language produced by Java compiler.
- The interpreter provides software compatibility across multiple platforms. To overcome the high-performance penalty of interpretation, which can be on the order of a factor of 10, just-in-time java compiler.

Memory Management

- Mapping of variable names to address is done co-operatively by the front end and code generator.
- Name and width are obtained from symbol table. Width is the amount of storage needed for that variable.
- Each three-address code is translated to addresses and instructions during code generation.
- A relative addressing is done for each instruction. All the labels should be addressed properly.

Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by a factor such as,
  - the level of the IR
  - the nature of the instruction-set architecture
  - the desired quality of the generated code.

- If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates.
- Such statement-by-statement code generation often produces poor code that needs further optimization.
- If IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequence.
- The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. Uniformity and completeness of the instruction set are important factors.

- If the target program does not support each data type in a uniform manner, then each exception to the general rule requires special handing.
- For example in some machines floating point operations are done using separate registers. Instruction speed and machine idioms are other important factors.

Example

```
x = y + z
        MOV y, R0
        ADD z, R0
        MOV R0, x
```

Example

```
a = b + c
d = a + e
        MOV b, R0
        ADD c, R0
        MOV R0, a
        MOV a, R0------------------- can be avoided.
        ADD e, R0
        MOV R0, d
```

- The quality of the generated code is usually determined by its speed and size. On most machines, a given IR program can be implemented by many different code sequence, with significant cost difference between the different implementations.
- For example if the target machine has an increment instruction INC, then the three-address statement a = a+1 may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads a into a register, adds one to the register, and then store the result back into a.

```
MOV a, R0
ADD #1, R0
MOV R0, a
```

Register Allocation

- A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but usually not have enough of then to hold all values.
- The use of registers is often subdivided into two sub problems:
- **Register allocation**, to select the set of variables that will reside in registers at each point in the program.
- **Register assignment**, to pick the specific register that a variable will reside in.
- Finding on optimal assignment of registers to variables is difficult, even with single register machines and it is an NP-complete problem.
- This problem becomes more complicated, if the target machine has certain conventions on register use.
- As in 8085, one of the operand of some operations should be placed in register A.

Choice Of Evaluation Order

- The order of evaluation can affect the efficiency of target code. Some order requires fewer registers and instructions than others.
- This can be solved up to an extend by code optimization in which the order of instruction may change.

Approaches To Code Generation

- The target code generated should be correct.
- Correctness depends on the number of special cases the code generator might face.
- Other design goals of code generator are, it should be easily implemented, tested and maintained.

TARGET MACHINE

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.

Target computer is a byte-addressable machine with four bytes to a word and n general purpose registers, R0, R1, R2…. Rn-1.

It has two address instructions of the form:

op source, destination

in which op is an op-code and source and destination are data fields.

- It has the following op-codes:

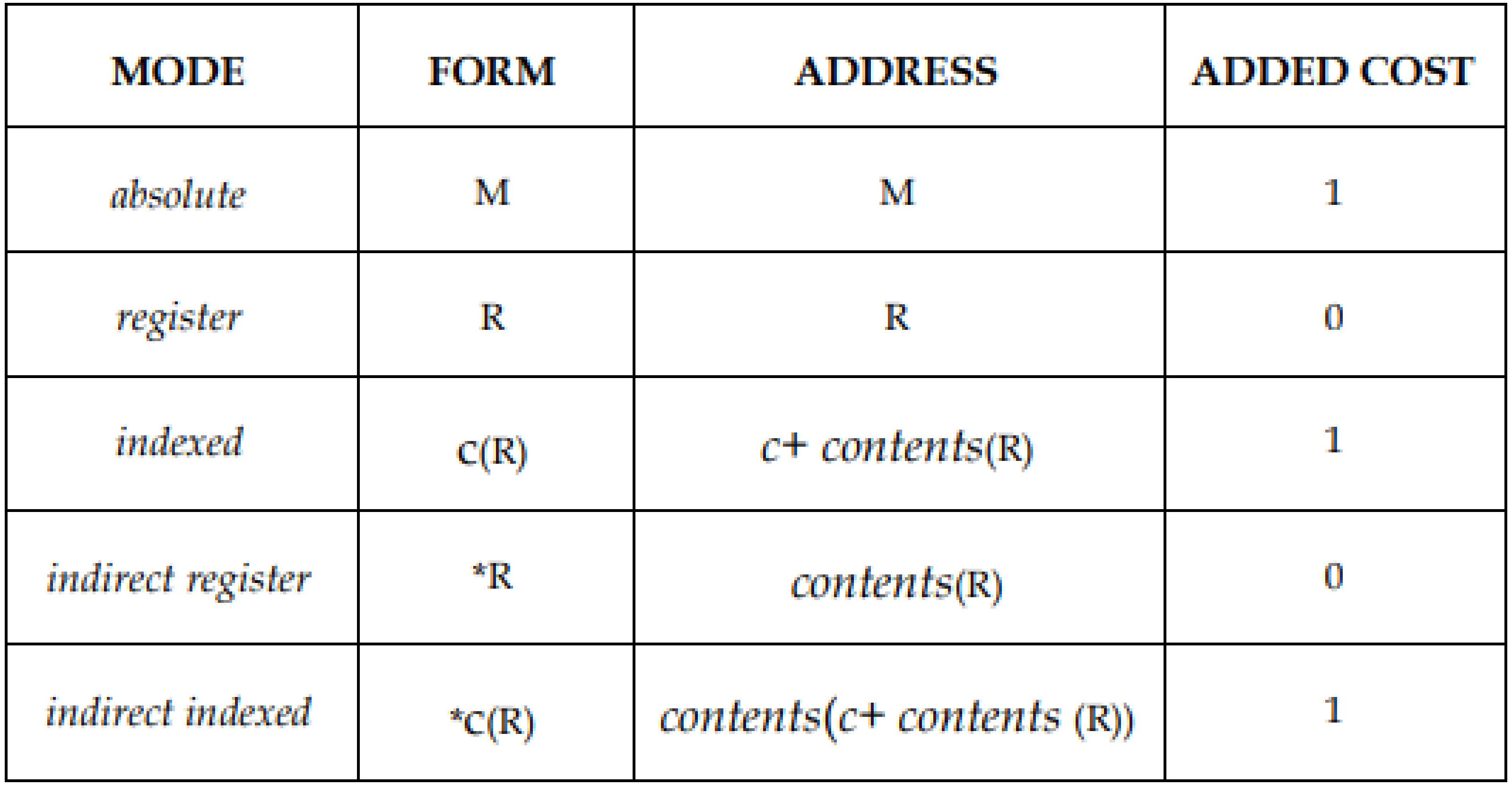
  MOV (move source to destination)

  ADD (add source to destination)

  SUB (subtract source from destination)

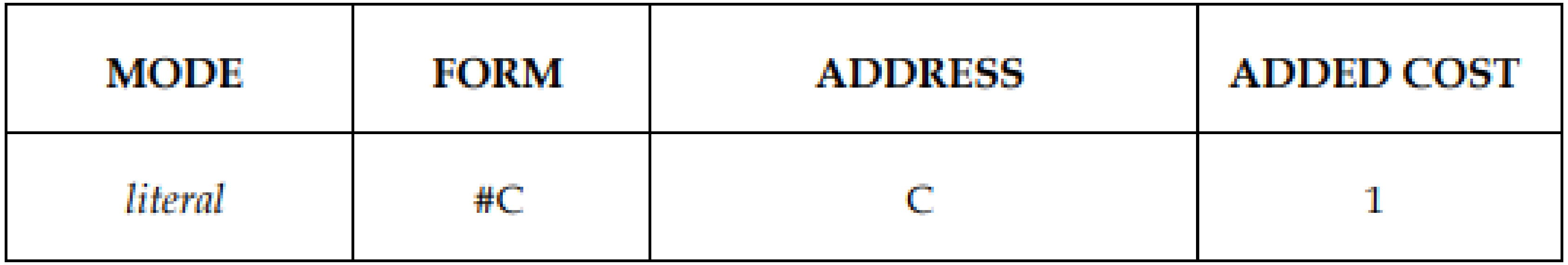
- The source and destination fields are not long enough to hold memory addresses, so certain bit patterns in these fields specify that words following an instruction contain operands and/or addresses.
- The source and destination of an instruction are specified by combining registers and memory locations with address mode. contents(a) denotes the contents of the register or memory address represented by a.

The address modes together with their assembly-language forms and associated costs are as follows:



MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	c(R)	*c+ contents*(R)	1
indirect register	*R	*contents*(R)	0
indirect indexed	*c(R)	*contents*(*c+ contents* (R))	1

- MOV R0, M – stores the contents of register R0 into memory location M.
- MOV 4(R0), M – stores the value contents(4 + contents(R0))
- MOV *4(R0), M – stores the value contents(contents(4 + contents(R0)))



MODE	**FORM**	**ADDRESS**	**ADDED COST**
literal	#C	C	1

- MOV #1, R0 – load constant 1 into register R0

Instruction Cost

- Cost of an instruction is 1 + the costs associated with the source and destination address modes, indicated by add cost in the table.
- This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero,
- while those with a memory location or literal in them have cost one, because such operands have to be stored with the instruction.

- The target is to minimize the length of instructions. Minimizing the instruction length will tend to minimize the time taken to perform the instruction as well.
- The instruction MOV R0, R1 copies the contents of register R0 into register R1. This instruction has cost one, since it occupies only one word of memory.
- The (store) instruction MOV R5 , M copies the contents of register R5 into memory location M. This instruction has cost two, since the address of memory location M is in the word following the instruction.
- The instruct ion ADD # 1 , R3 adds the constant 1 to the contents of register 3, and has cost two, since the constant 1 must appear in the next word following the instruction.

Example

- How the following three-address statement can be implemented by many different instruction sequences ?

a : = b + c

Example

```
1.      MOV b, R0
        ADD c, R0
        MOV R0, a
```

Cost: 6

```
2.      MOV b, a
        ADD c, a
```

Cost = 6

Assuming R0, R1 , and R2 contain the addresses of a, b, and c respectively, then:

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

Cost = 2

Assuming R1 and R2 contain the values of b and c, respectively, and that the value of b is not needed after the assignment, we can use:

```
4.      ADD R2, R1
        MOV R1, a                    cost = 3
```

Generating Code For Assignment Statements

d : = (a-b) + (a-c) + (a-c)

Three Address Code:

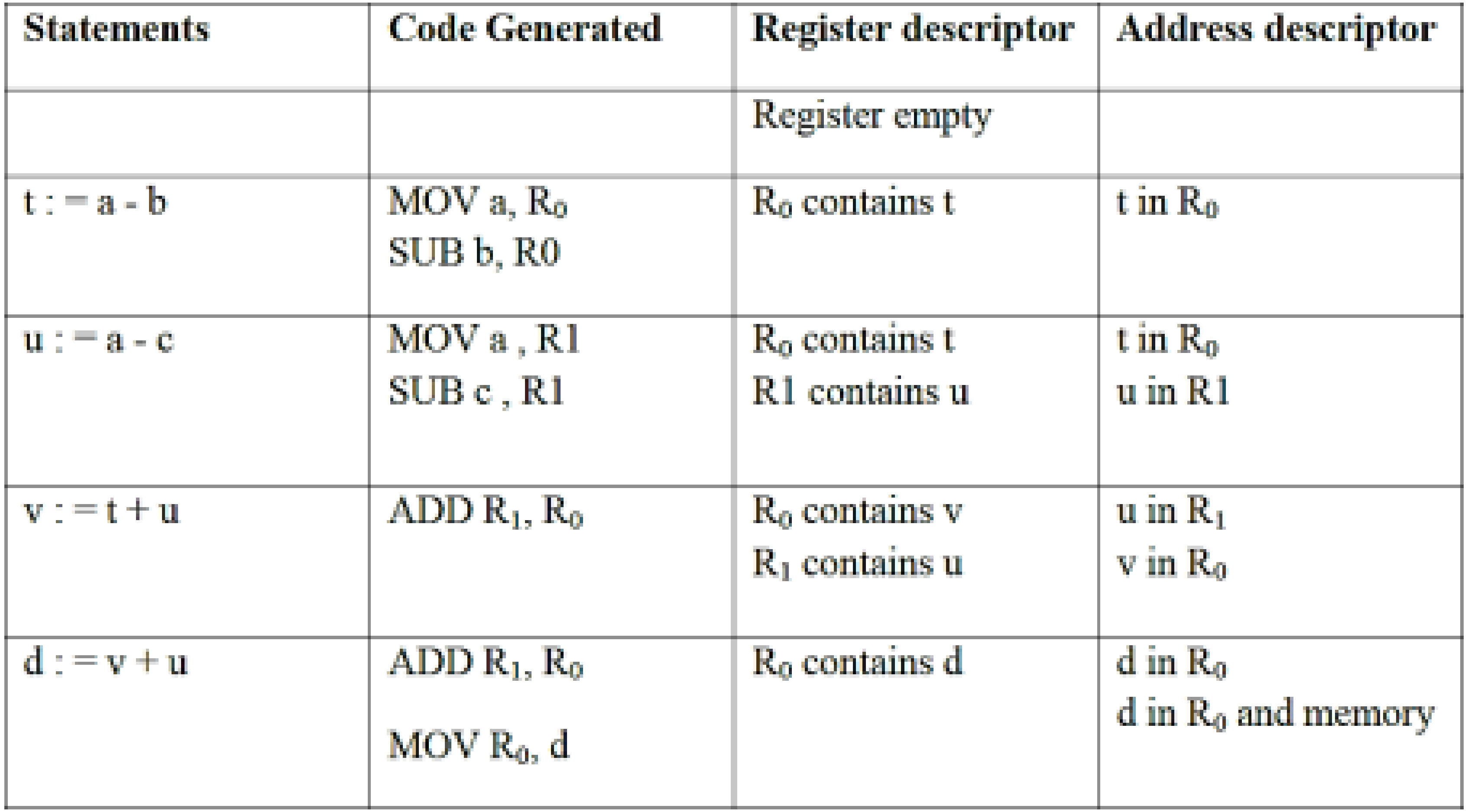
t := a – b

u := a – c

v := t + u

d := v + u

Target Code



Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Example: I = J + K

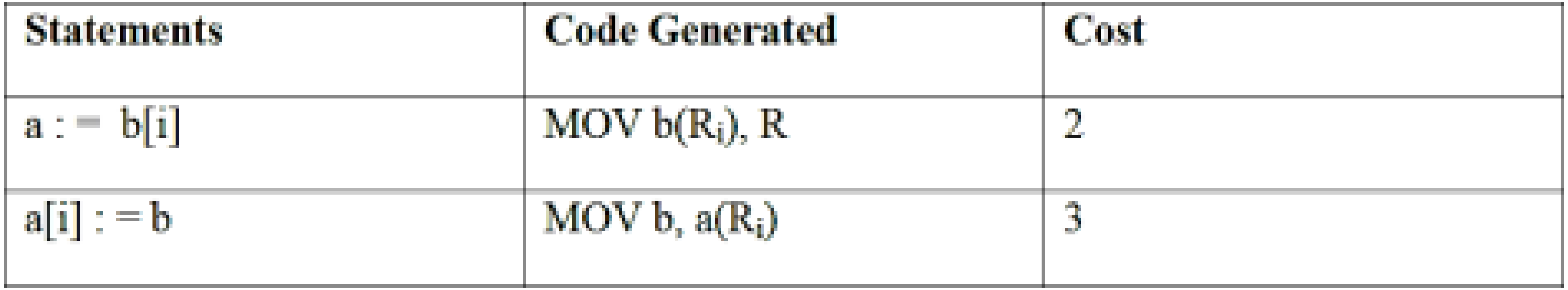
Intermediate code is then generated:

- R1 = J
- R2 = K
- R1 = R1 + R2
- I = R1

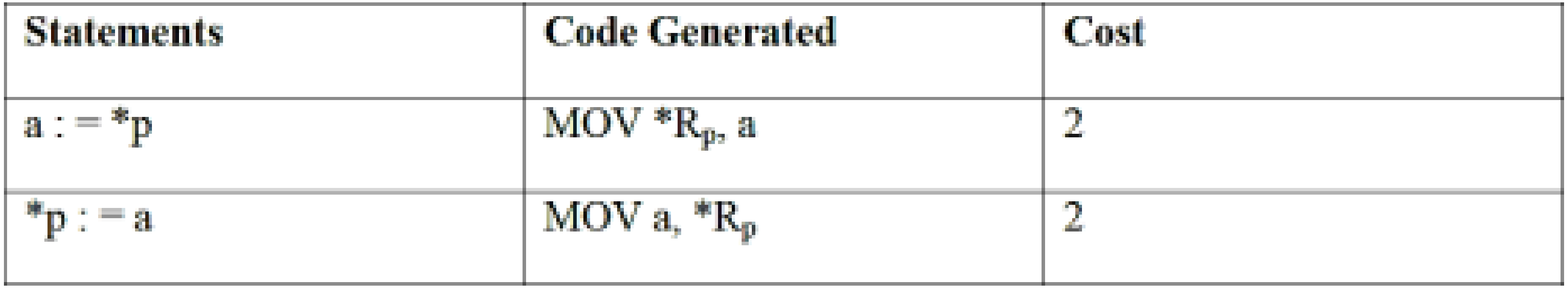
Then, the target code:

- LDR R1, J //loads value of j into r1
- LDR R2, K //loads value of k into r2
- ADD R1, R1, R2 //adds r1 and r2, stores it into r1
- STR I, R1 //stores the value of R1 to I

Example



Statements	Code Generated	Cost
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3



Statements	Code Generated	Cost
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

PEEPHOLE OPTIMIZATION

- Peephole optimization is a simple and effective technique for locally improving target code.
- This technique is applied to improve the performance of the target program by examining the short sequence of target instructions (called the peephole) and
- replace these instructions replacing by shorter or faster sequence whenever possible.
- Peephole is a small, moving window on the target program.

Peephole Optimization

- Redundant-instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Unreachable Code

Redundant instruction elimination

- The redundant loads and stores can be eliminated in this type of transformations.

Example:

```
        MOV R0, x
        MOV x, R0
```

- We can eliminate the second instruction since x is in already R0. But if MOV x, R0 is a label statement then we cannot remove it.

Unreachable Code

- Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

EXAMPLE

```
if debug == 1 goto L1
goto L2
L1: Print debugging information
L2:
```

- In this code segment, the debugging condition can be changed to eliminate multiple jump statements.

```
if debug != 1 goto L2
Print debugging information
L2:
```

Control Flow Optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...
MOV R1, R2
GOTO L1
...
L1 : GOTO L2
L2 : INC R1
```

In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2:

```
...
MOV R1, R2
GOTO L2
...
L2 : INC R1
```

Algebraic simplification

- There are occasions where algebraic expressions can be made simple. For example,

  the expression a = a + 0

  can be replaced by a itself and

  the expression a = a + 1

  can be replaced by INC a.

Reduction in strength

- There are operations that consume more time and space. Their ‘strength’ can be reduced by replacing them with other operations that consume less time and space, but produce the same result.
- For example, x * 2 can be replaced by x << 1, which involves only one left shift.

Machine idioms

- So The target instructions have equivalent machine instructions for performing some have operations.
- Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.
- Example:

  Some machines have auto-increment or auto-decrement addressing modes. These modes can use in a code for a statement like i=i+1. (INC/DEC machine instruction)

Instruction Scheduling Example

Operation:

(a+b)+(e+(c-d))

t1 = a+b

t2 = c-d

t3 = e+t2

t4 = t1+t3

Instruction Order

```
MOV a, R0
ADD b, R0
MOV c, R1
SUB d, R1
MOV R0, t1
MOV e, R0
ADD R0, R1
MOV t1, R0
ADD R1, R0
MOV R0, t4
```

After Rescheduling

Operation:

(a+b)+(e+(c-d))

t2 = c-d

t3 = e+t2

t1 = a+b

t4 = t1+t3

Instruction Order

```
MOV c, R0
SUB d, R0
MOV e, R1
ADD R0, R1
MOV a, R0
ADD b, R0
ADD R1, R0
MOV R0, t4
```

Example2

a+b-(c+d)*e

t1=a+b

t2=c+d

t3=e*t2

t4=t1-t3

```
MOV a, R0
ADD b, R0
MOV R0, t1
MOV c, R1
ADD d, R1
MOV e, R0
MUL R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

After Rescheduling

a+b-(c+d)*e

```
t2=c+d
t3=e*t2
t1=a+b
t4=t1-t3
```

```
MOV c, R0
ADD d, R0
MOV e, R1
MUL R0, R1
MOV a, R0
Add b, R0
SUB R1, R0
MOV R0, t4
```

Example 3

t= a-b

u= a-c

v= t+u

a= d

d= v+u

Example

```
x = a[i]
y = b[i]
z = x * y
```

```
LD R1, i
MUL R1, R1, #4
LD R2, a(R1)
LD R1, b(R1)
 MUL R1, R2, R1
 ST z, R1
```

Example

```
if x < y goto L1
    z = 0
    goto L2
L1: z = 1
```

```
    LD R1, x
    LD R2, y
    SUB R1, R1, R2
    BLTZ R1, L1
    LD R1, #0
    ST z, R1
    BR L2
L1: LD R1, #1
    ST z, R1
```

Example

```
s = 0
    i = 0
L1: if i > n goto L2
    s = s + i
    i = i + 1
    goto L1
L2:
```

```
    LD R1, #0
    ST s, R1
    ST i, R1
L1: LD R1, i
    LD R2, n
    SUB R2, R1, R2
    BGTZ R2, L2
    LD R2, s
    ADD R2, R2, R1
    ST s, R2
    ADD R1, R1, #1
    ST i, R1
    BR L1
L2:
```

```
    LD R2, #0
    LD R1, R2
    LD R3, n
L1: SUB R4, R1, R3
    BGTZ R4, L2
    ADD R2, R2, R1
    ADD R1, R1, #1
    BR L1
L2:
```

Example: Calculate cost of the operation

```
LD R0, y
LD R1, z
ADD R0, R0, R1
ST x, R0
```

Example: Calculate cost of the operation

```
a:=a+1

MOV a,R0
ADD #1,R0
MOV R0,a

ADD #1,a

INC a
```