MODULE 6

Code Optimization: Principal sources of optimization, Optimization of Basic blocks

Code generation: Issues in the design of a code generator. The target machine, A simple code generator.

6.1 CODE OPTIMIZATION

- The code generated by the compiler can be made faster or take less space or both. Some transformations can be applied on this code called optimization or optimization transformations.

- Compilers that can apply optimizing transformations are called optimizing compilers.

- Code optimization is an optional phase and it must not change the meaning of the program.

- 2 points concerning the scope of optimization:

    1. CO aims at improving a program, rather than improving the algorithm used in the program. Thus replacement of an algorithm by a more efficient algorithm is beyond the scope of CO.
    2. Efficient code generation for a specific target machine (eg: by fully exploiting its instruction set) is also beyond the scope of CO.

- Compiler was found to consume 40% extra compilation time due to optimization. The optimized program occupied 25% less storage and executed 3 times faster than unoptimized program.

Need For Optimization Phase In Compiler

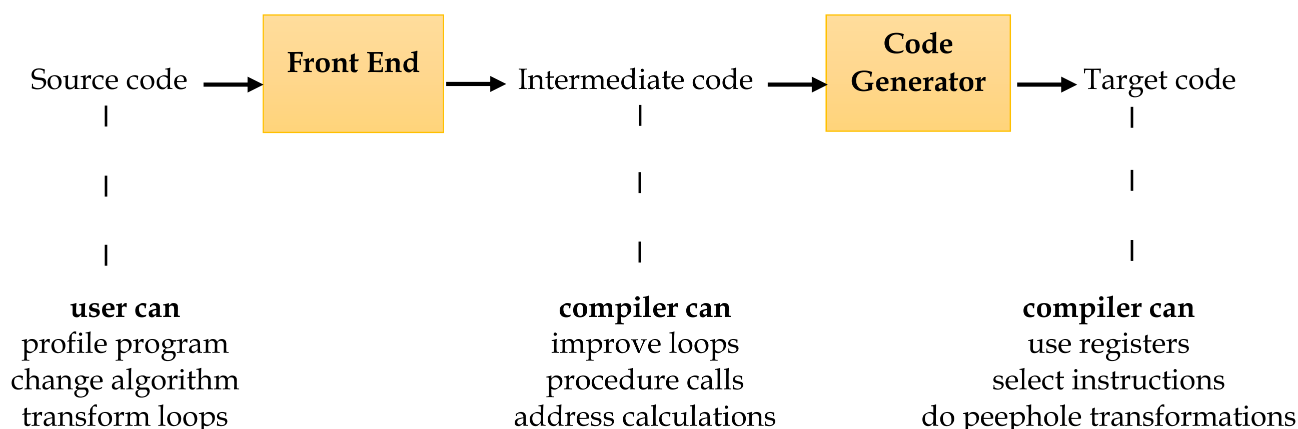
1. Code produced by a compiler may not be perfect in terms of execution speed and memory space.
2. Optimization by hand takes much more effort and time.
3. Machine level details like instructions and addresses are not known to the programmer.
4. Advanced architecture features like instruction pipeline requires optimized code.
5. Structure reusability and maintainability of the code are improved.

Criteria For Code Optimization

1. It should preserve the meaning of the program ie, it should not change the output or produce error for a given input. This approach is called safe approach.
2. Eventually it should improve the efficiency of the program by a reusable amount. Sometimes it may increase the size of the code or may slow the program slightly but it should improve the efficiency.
3. It must be worth with the effort, ie, the effort put on optimization must be worthy when compared with the improvement.

Optimization can be applied in 3 places.

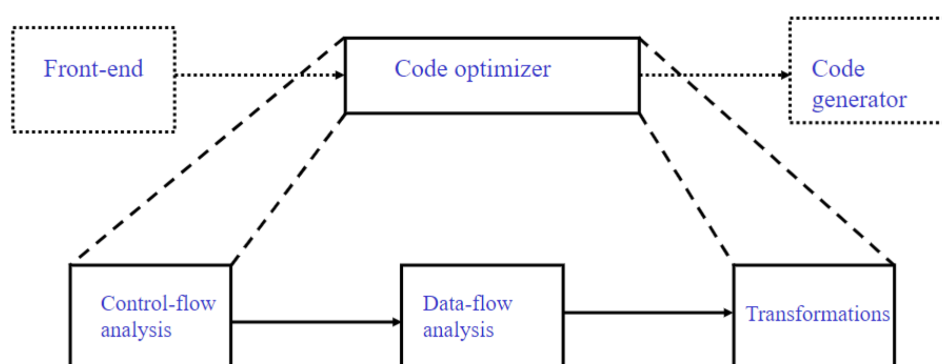
- Source code
- Intermediate code
- Target code



There are two types of optimization

1. Machine dependent optimization – run only in particular machine.
2. Machine independent optimization – used for any machine.

Organization Of Code Optimizer



Optimization can be done in 2 phases

1. **Local optimization**
   Transformations are applied over a small segment of the program called basic block, in which statements are executed in sequential order. Speed-up factor for local optimization is 1.4.

2. **Global optimization**
   Transformations are applied over a large segment of the program like loop, procedures, functions etc. Local optimization must be done before applying global optimization. Speed-up factor is 2.7.

Basic Block

- Basic block is a sequence of consecutive 3 address statements which may be entered only at the beginning and when entered statements are executed in sequence without halting or branching.

- To identify basic block, we have to find **leader** statements. Rules for leader statements are

Input: A sequence of three-address statements
Output: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements if basic blocks
   a) The first statement is the leader
   b) Any statement that is the target of a goto is a leader
   c) Any statement that immediately follows a goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

- From a leader statement to all statements up to but, not including the next leader is a **basic block**.

Flow Graphs

- Basic block is a sequence of consecutive 3 address statements which may be entered only at the beginning and when entered statements are executed in sequence without halting or branching.

- To identify basic block, we have to find **leader** statements. Rules for leader statements are

- It is the pictorial representation of control flow analysis in a program. It shows the relationship among basic blocks.

- Nodes are basic blocks and edges are control flow. It is a directed graph **G = (N,E n_0).**

Where, N – set of basic blocks

E – set of control flows

n_0 - starting node

- If there is a directed edge from B1 to B2, the control transfers from the last statement of B1 to the first statement of B2. B1 is called predecessor of B2 and B2 is successor of B1.

EXAMPLE

Consider the code for quick sort

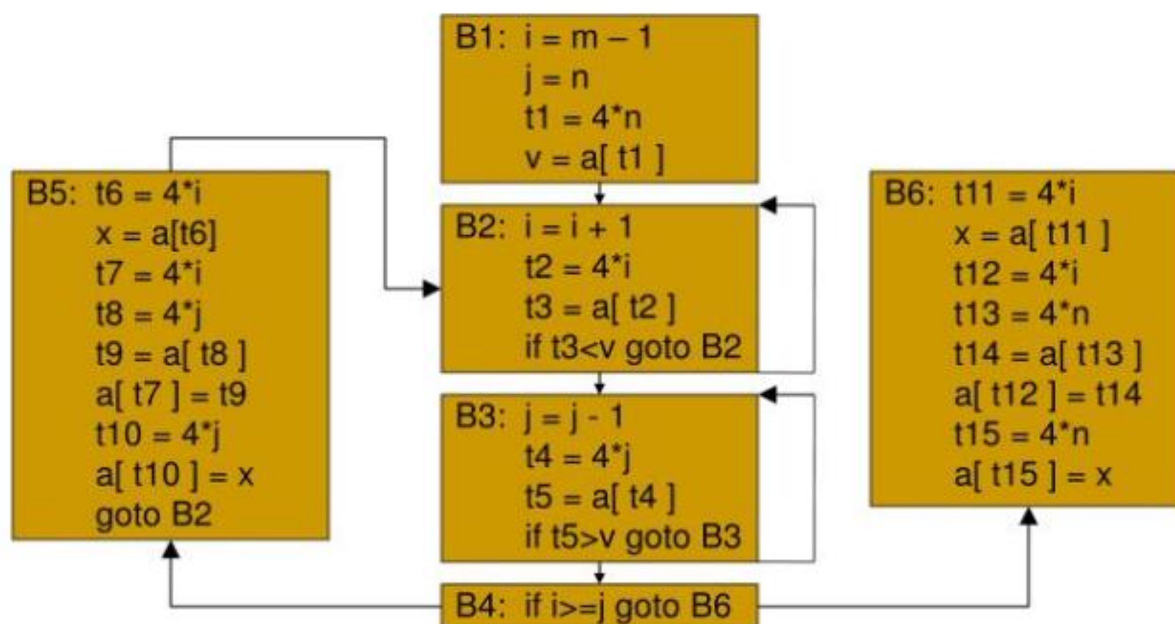
```
void quicksort(m, n)
int m, n;
{
 int I, j;
 if (n <= m ) return;
 /* fragment begins here */
 i = m-1; j = n; v = a[n];
 while(1)  {
    do i = i+1; while( a[i] < v );
        do j = j-1; while( a[j] > v );
        if( i >= j ) break;
        x = a[i]; a[i] = a[j];    a[j] = x;
    }
 x = a[i];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

Three address code for quick sort fragment

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

Flow graph for quick sort



6.1.1 PRINCIPAL SOURCES OF OPTIMIZATION

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

1. **Function preserving transformations**
   a. Common subexpression elimination
   b. Copy propagation
   c. Dead code elimination
   d. Constant folding
2. **Loop optimization**
   a. Code motion
   b. Induction variable elimination
   c. Reduction in strength

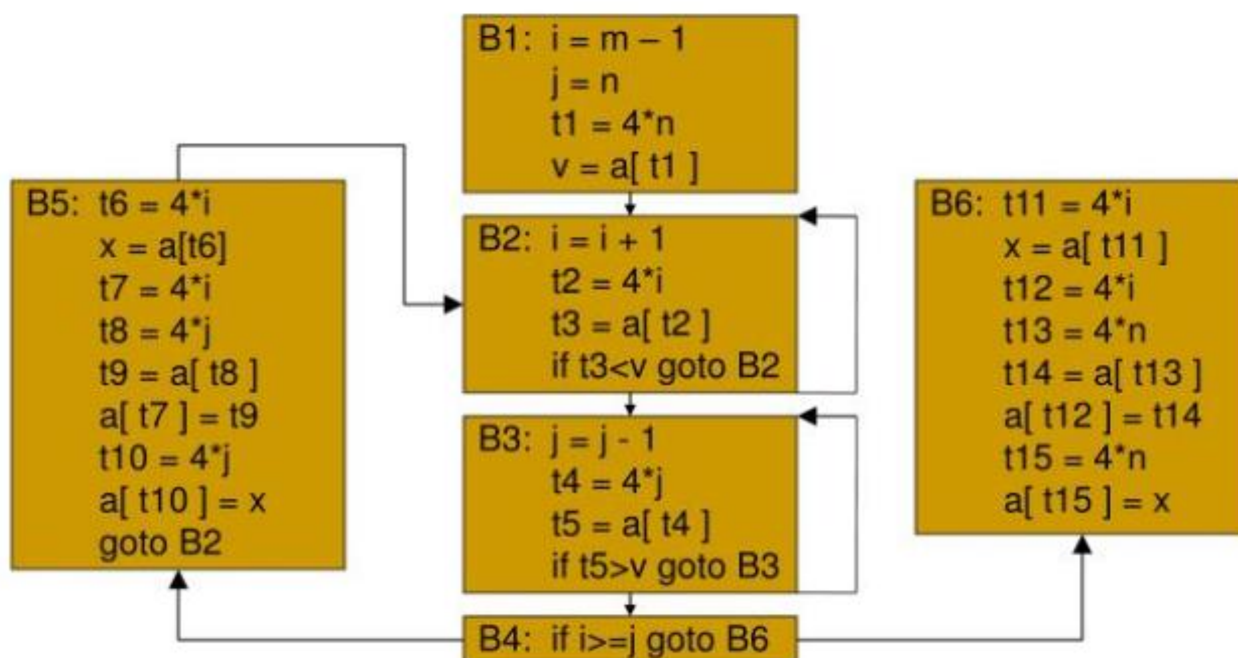
Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- Some function preserving transformations examples are given below

Common Sub Expression Elimination (CSE)

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.

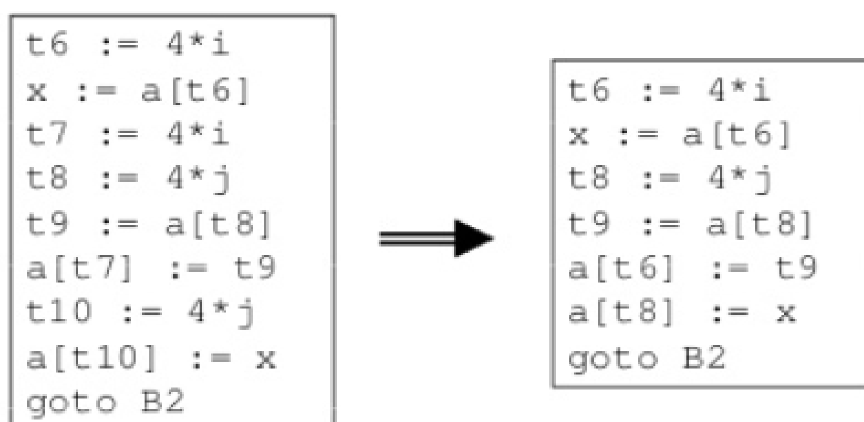
- We can avoid recomputing the expression if we can use the previously computed value. Two types are: -
  - Local common sub expression elimination
  - Global common sub expression elimination
- Consider the flow graph of quick sort fragment



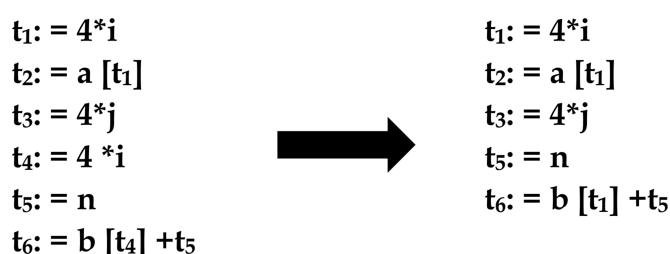
Local common sub expression elimination

EXAMPLE 1

B_5



EXAMPLE 2



t_1: = 4*i		t_1: = 4*i
t_2: = a [t_1]		t_2: = a [t_1]
t_3: = 4*j	→	t_3: = 4*j
t_4: = 4 *i		t_5: = n
t_5: = n		t_6: = b [t_1] +t_5
t_6: = b [t_4] +t_5		

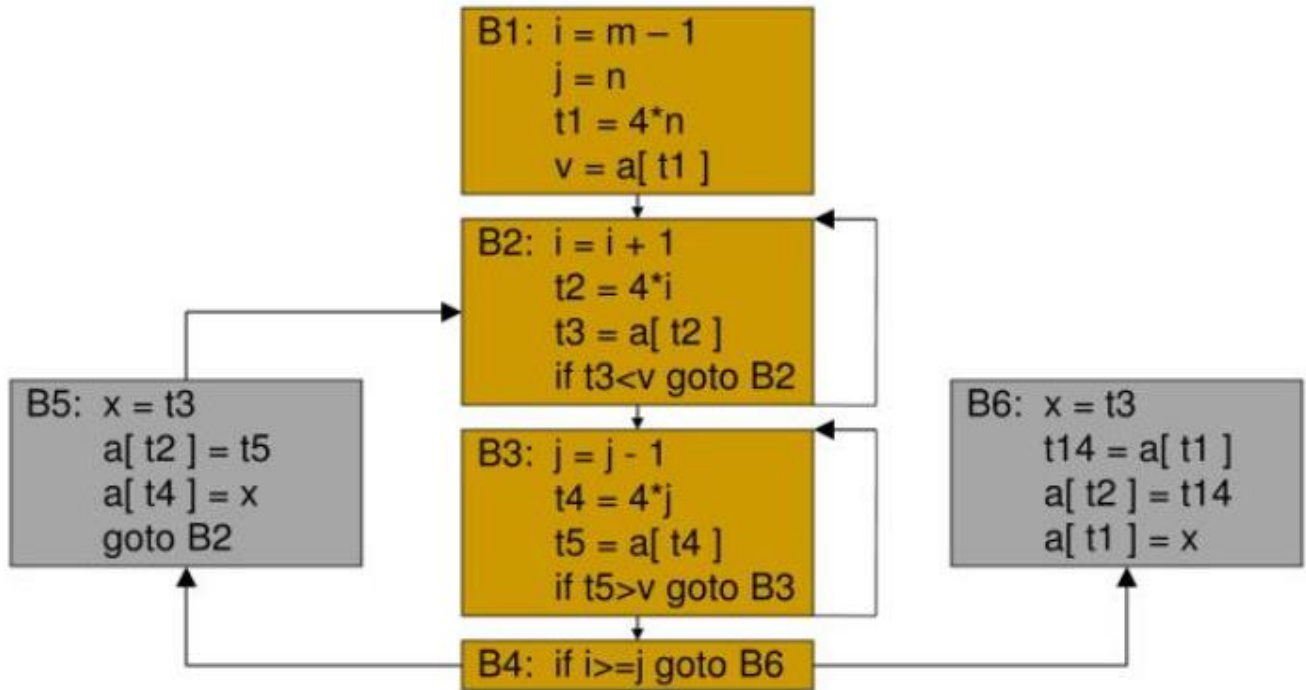
The common sub expression **t_4: =4 * i** is eliminated as its computation is already in **t_1** and the value of **i** is not been changed from definition to use.

Global common sub expression elimination

- After local common subexpressions are eliminated, **B_5** stiil evaluates **4 * i** and 4+ j, as shown in **EXAMPLE 1** of local common subexpression elimination.

t_8 = 4 * j
t_9 = a[t_8] in **B_5** can be replaced by **t_9 = a[t_4]**
a[t_8] =x **a[t_4] = x** using **t_4** computed in block **B_3**

- In Flow graph given above, observe that as control passes from the evaluation of **4 * j** in **B_3** to **B_5**, there is no change to **j** and no change to **t_4**, so **t_4** can be used if **4 * j** is needed.
- Another common subexpression comes to light in **B_5** after **t_4** replaces **t_8**. The new expression **a[t_4]** corresponds to the value of **a[j]** at the source level.
- Not only does **j** retain its value as control leaves **B_3** and then enters **B_5**, but **a[j]**, a value computed into a temporary **t_5**, does too, because there are no assignments to elements of the array a in the interim.
- The statements , **t_9 = a[t_4]**
a[t_6] = t_9 in **B_5** therefore can be replace by **a[t_6] = t_5** 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 **B_6** in Flow graph. The expression **a[t_1]** in blocks **B_1** and **B_6** is not considered a common sub expression, although **t_1** can be used in both places.
- After control leaves **B_1** and before it reaches **B_6**, it can go through **B_5**, where there are assignments to **a.** Hence, **a[t_1]** may not have the same value on reaching **B_6** as it did on leaving **B_1**, and it is not safe to treat **a[t_1]** as a common sub expression.



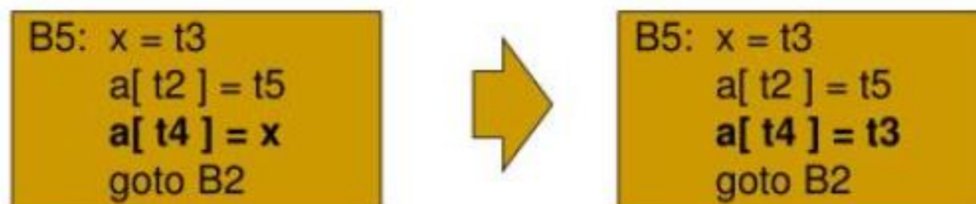
B_5 and B_6 after common subexpression elimination.

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.

EXAMPLE 1

The assignment **x : = t_3** in block **B_5** of Flow graph is a copy.

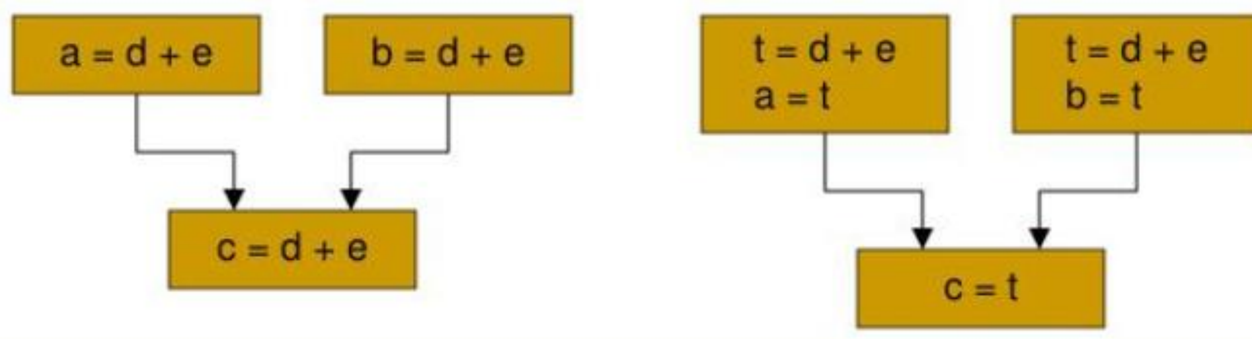


This change may not appear to be an improvement, but it gives us the opportunity to eliminate the assignment to x.

EXAMPLE 2

When the common subexpression in **c := d+e** is dominated in fig given below, 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**.



ADVANTAGE

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 1

Consider B5 of flow graph.

```
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 2

```
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.

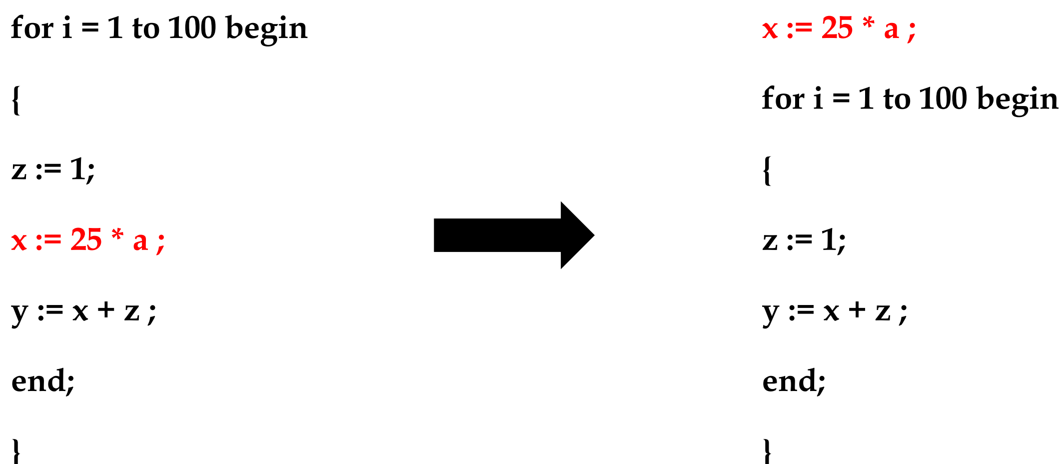
Loop Optimization

- 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 1



Here **x := 25 * a ;** is a loop variant. Hence in the optimised program it is computed only once before entering the for loop. **y := x + z ;** is not loop invariant. Hence it cannot be subjected to frequency reduction.

EXAMPLE 2

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 consider 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.

Reduction In Strength

- 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.

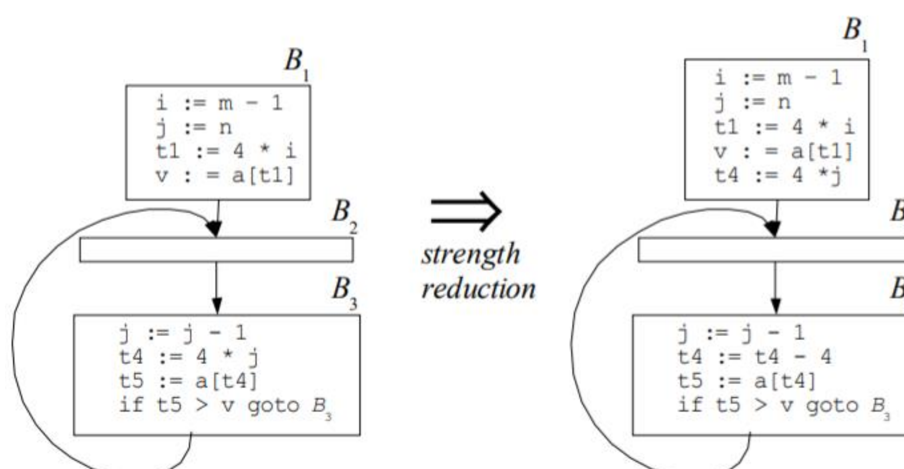
- However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

  EXAMPLE

  As the relationship t4:=4*j surely holds after such an assignment to t4 in Figure. 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, shown by the dashed addition to block B1 in Figure



- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

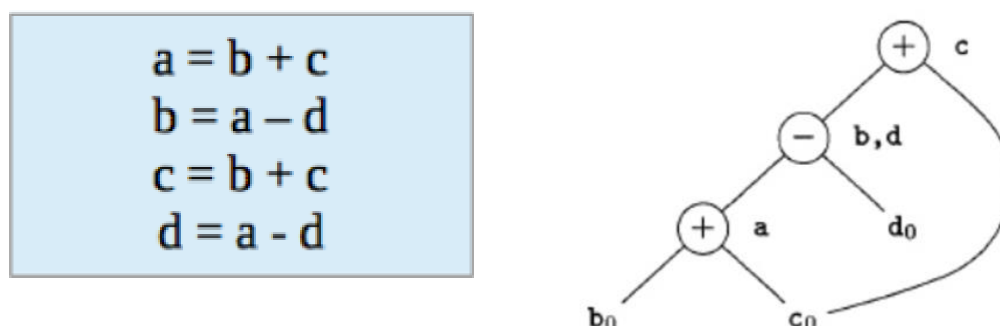
6.1.2 OPTIMIZATION OF BASIC BLOCKS

Many of the structure preserving transformations can be implemented by constructing a dag for a block. There is a node ***n*** associated with each statement ***s*** within the block. The children of ***n*** are those nodes corresponding to statement that are the last definitions prior to ***s*** of the operands used by ***s***.

Directed Acyclic Graph

- In compiler design, a DAG is an abstract syntax tree with a unique node for each value. DAG is an useful data structure for implementing transformation on basic block. DAG is constructed from three address code.

- Common subexpression can be detected by noticing, as a new node m is about to added, whether there is an existing node n with the same children, in the same order, and with the same operator. If so, n computes the same value as m and may be used in its place.



- When we construct the node for the third statement **c = b + c**, we know that the use of **b** in **b + c** refers to the node labeled -, because that is the most recent definition of **b**.

Application of DAG

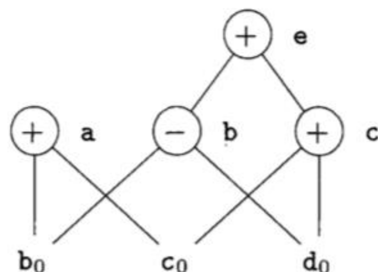
- Determine the common subexpression.
- Determine which names are used in the block and compute outside the block.
- Determine which statement of the block could have their computed value outside the block.
- Simplify the list of quadruples by eliminating common subexpression and not performing the assignment of the form x = y and unless it is a must.

Rules For The Construction Of A DAG

1. In a DAG Leaf node represents identifiers, names, constants. Interior node represents operators.
2. While constructing DAG, there is a check made to find if there is an existing node with same children. A new node is created only when such a node does not exist. This action allows us to detect common subexpression and eliminate the same.
3. Assignment of the form x = y must not be performed until unless it is a must.

EXAMPLE 1

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

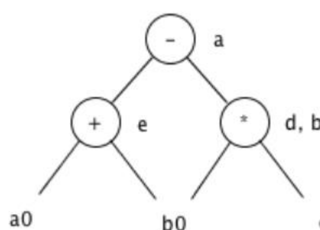


The two occurrences of the sub-expressions **b + c** computes the same value.

Value computed by **a** and **e** are the same.

EXAMPLE 2

d = b * c
e = a + b
b = b * c
a = e – d

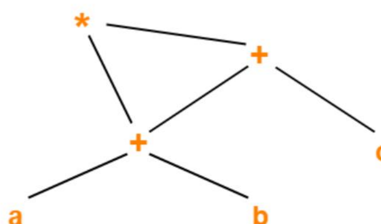


EXAMPLE 3

(a + b) * (a + b + c)

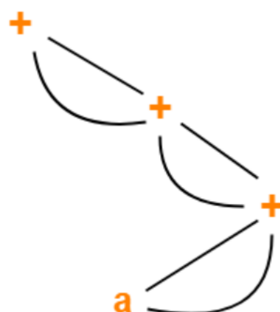
Three address code will be

t1 = a + b
t2 = t1 + c
t3 = t1 * t2



EXAMPLE 4

(((a + a) + (a + a)) + ((a + a) + (a + a)))



The Use of Algebraic Identities

- It represents another important class of optimizations on basic blocks.

  x + 0 = 0 + x = x
  x – 0 = x
  x * 1 = 1 * x = x
  x / 1 = x

- Another class of algebraic optimization includes reduction in strength.

  x ** 2 = x * x
  2 * x = x + x
  x / 2 = x * 0.5

- associative laws may also be applied to expose common subexpression.

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

- With the intermediate code might be

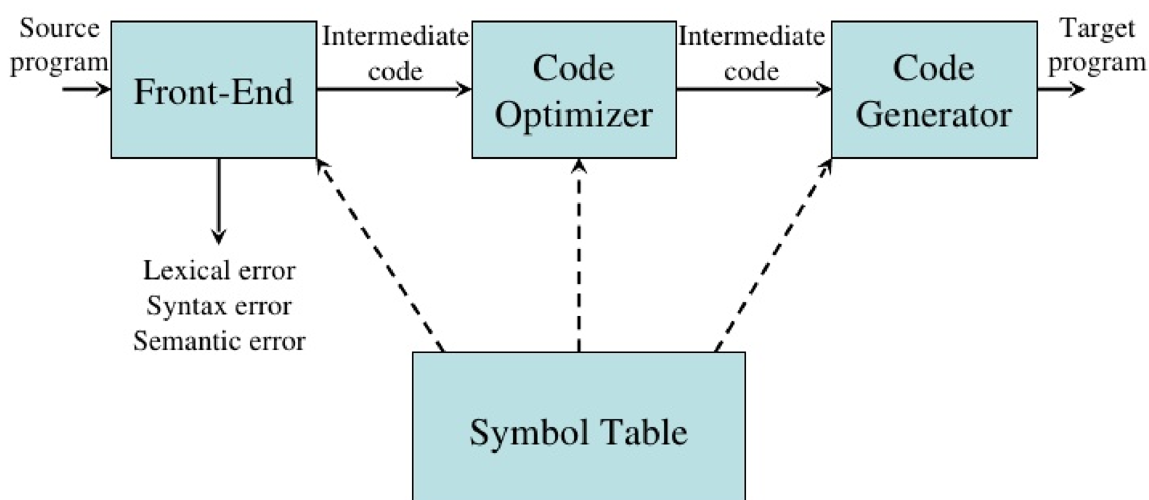
  a = b + c
  t = c + d
  e = t + b

- If t is not needed outside this block, the sequence can be

  a = b + c
  e = a + d

6.2 CODE GENERATION

- The final phase in our 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.



6.2.1 ISSUES IN THE DESIGN OF A CODE GENERATOR

- The following issues arise during the code generation phase:

    1. Input to code generator
    2. Target program
    3. Memory management
    4. Instruction selection
    5. Register allocation
    6. Evaluation order
    7. 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, virtual machine representations such as stack machine code, and graphical representations such as syntax trees and dags.
- We assume 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. We can 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. Al the labels should be addressed properly. Backward jump is easier to manage than the forward jump.

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.eg: in some machines floating point operations are done using separate registers. **Instruction speed and machine idioms** are other important factors.

- If we do not care about the efficiency of the target program, instruction selection is straightforward. For each common three-address statement, a general code can be designed.

  Eg: x = y + z

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

  Eg:

  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.

- Eg: 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 we usually not have enough of then to hold all values.
- The use of registers is often subdivided into two sub problems:
  - Register allocation, during which we select the set of variables that will reside in registers at each point in the program.
  - Register assignment, during which we 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.
- Eg: 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.
- Picking the best order is an NP-complete problem. 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.

6.2.2 TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- Our 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 one plus the costs associated with the source and destination address modes, indicated by add cost in the above 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.
- We should clearly minimize the length of instructions. Minimizing the instruction length will tend to minimize the time taken to perform the instruction as well.

1. 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.
2. 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.
3. The instruct ion ADD # 1 , R3 adds the constant I to the contents of register 3, and has cost two, since the constant I must appear in the next word following the instruction.
4. The instruction SUB 4 (R0) , * 12 (R) stores the value

***contents* (*contents* (12+ *contents* (R1))) - *contents* (*contents* (4 +R0))**

into the destination *12 (R1) .The cost of this instruction is three, since the constants 4 and 12 are stored in the next two words following the instruction.

Here are some examples

1. MOV b, R0
   ADD c, R0
   MOV R0, a
   cost = 6

2. MOV b, a
   ADD c, a
   cost = 6

Assuming R0, R 1 , and R2 contain the addresses of a, b, and c. respectively, we can use:

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**

6.2.3 SIMPLE CODE GENERATOR

- The code generation strategy is the generation of target code for a sequence of three-address statement. We assume that computed result is in registers as long as possible, storing them only a) if their register is needed for another computation or b) just before a procedure call, jump or labelled statement.
- For a three-address statement a = b + c, generate instruction ADD Rj, Ri with cost one, leaving the result a in register Ri.
- This sequence is possible only if register Ri contains b, Rj contains c and b is not live after the statement; that is, b is not used after the statement.
- If Ri contain b but c is in a memory location,

```
ADD   c, Ri          cost =2

        Or

MOV   c, Rj
ADD   Rj, Ri         cost =3
```

Register And Address Descriptors

The code generation algorithm uses descriptors to keep track of register contents and addresses for names.

1. **A Register Descriptor** keeps track of what is currently in each register. It is consulted whenever a new register is needed.
2. **An Address Descriptor** keeps track of the location where the current value of the name can be found at run time. The location might be a register, a stack location or a memory address. This information can be stores in the symbol table and is used to determine the accessing method for a name.

A code-generation algorithm

Code generation algorithm takes input as a sequence of three-address statements constituting a basic block. Statement of the form x = y op z performs the following actions.

1. **Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.**
2. **Consult the address descriptor for y to determine y′, the current location of y. Prefer the register for y′ if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y′ , L to place a copy of y in L.**
3. **Generate the instruction OP z′ , L where z′ is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.**
4. **If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z**

The Function *getreg* M

The function 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 Y is not live and has no next use after X := Y op Z then return register of Y for L. Update the address descriptor of y to indicate that y is no longer in L.
2. Failing (1) return an empty register for L if there is one.
3. Failing (2) if X has a next use in the block or op is an operator, such as indexing that requires a register, find an occupied register R. Store the value of R into a memory location (by MOV R, M) If it is not already in proper memory location M, update the address descriptor for M, and return R. if R hold the value of several variables, a MOV instruction must be generated for each variable that need to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future, or one whose value is also in memory. We leave the exact choice unspecified, since there is no one proven best way to make the selection.
4. If X is not used in the block. Or no suitable occupied register can be found, select the memory location of X as L.

Generating Code For Assignment Statements

The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

t := a – b
u := a – c
v := t + u
d := v + u with d live at the end.

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

Generating Code for other type of statements

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

KTU ASSIST

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */

x : = y +z
if x < 0 goto z
	MOV y, R_0
	ADD z, R_0
	MOV R_0,x
	CJ< z

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.

Characteristics Of Peephole Optimization

So The peephole optimization can be applied to the target code using the following characteristic.

1. Redundant instruction elimination

- Especially 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.

2. 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

```
void add_ten(int x)
{
return x + 10;
printf(“value of x is %d”, x);
}
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

3. The flow of control 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, as shown below:

...

MOV R1, R2

GOTO L2

...

L2 : INC R1

4. 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 simply be replaced by INC a.

5. 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. Though the output of a * a and a2 is same, a2 is much more efficient to implement

6. 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.

****