UNIT VI : Code Generation

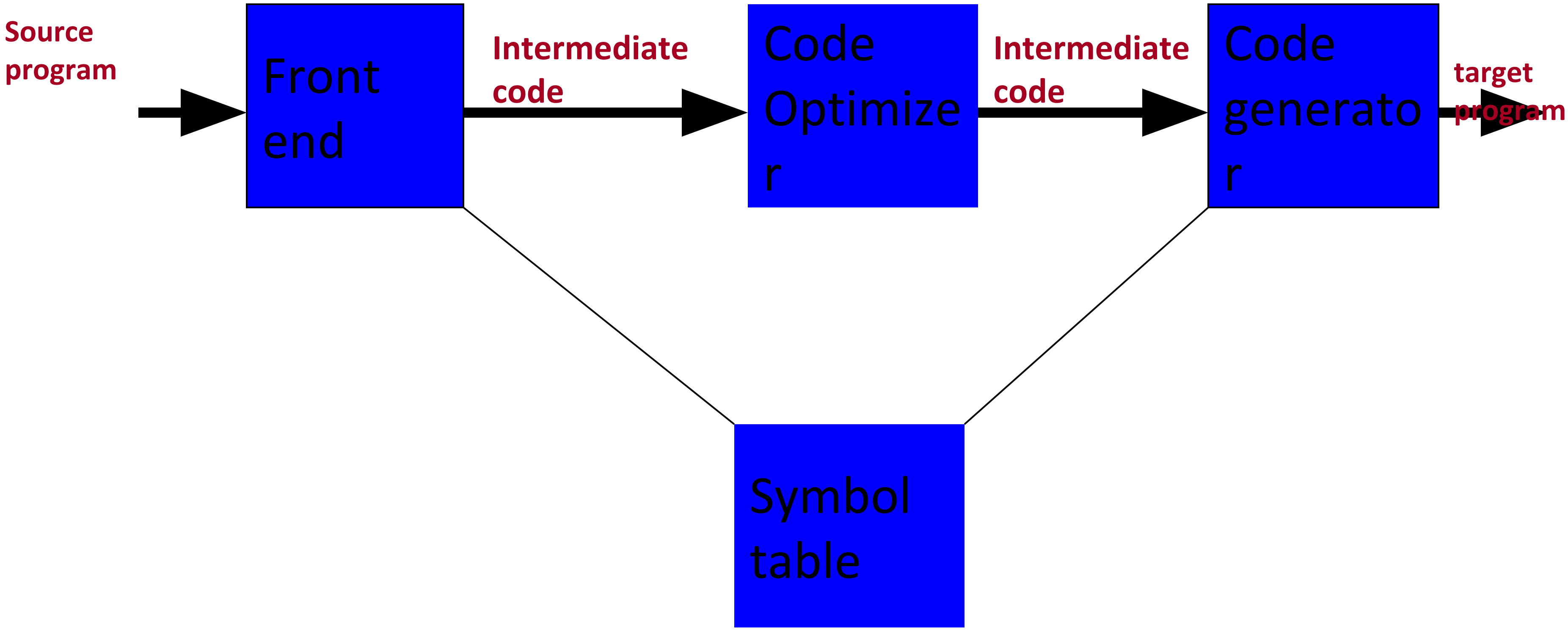
Syllabus UNIT-VI: Code Generation

- Code generation Issues
- Basic blocks and flow graphs
- A Simple Code Generator
- Code Optimization: Machine Independent, Peephole optimizations, Common Sub-expression elimination, Removing of loop invariants, Induction variables and Reduction in strengths, Use of machine idioms, Dynamic Programming Code Generation
- Machine dependent Issues: Assignment and use of registers

Discussion on Issues in **Code Generation** Phase of Compiler

Underlying machine architecture plays an important role

Introduction



Position of code generator

Code generation

- Produces the target language in a specific architecture.
- The target program is normally a relocatable object file containing the machine codes.
- Ex:

(assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
MOVE    id2,R1
MULT    id3,R1
ADD     #1,R1
MOVE    R1,id1
```

Code Optimization

- Code produced by standard algorithms can often be made to **run faster**, take **less space** or **both**
- These improvements are achieved through transformations called **optimizations**
- Compilers that apply these transformations are called optimizing compilers

Code Optimization Issues

- After Optimization
  - Meaning must be preserved (correctness)
  - Speedup must occur on average
  - Work done must be worth the effort
  - No change in output
  - Optimization should not introduce any error

Code Optimization Types

- Machine Independent Optimization
  - (1) Constant Folding / Compile time evaluation
  - (2) Code Motion, Removing of loop invariants
  - (3) Dead code elimination
  - (4) Common Subexpression elimination / Variable Propagation
  - Reduction in strengths (Induction Variable and Strength Reduction)
- Machine dependent Issues
  - Assignment and Use of registers
  - Managing bounded machine resources like registers, functional units, caches

Q) Where is code optimization carried out?

Code optimization is carried out on the **intermediate code** because program analysis is more accurate on intermediate code than on machine code.

Issues during the **Code Generation** phase

- Compilers may encounter issues during the code generation phase
- Addressing these issues may requires a combination of sophisticated algorithms, heuristics, and platform-specific knowledge

…contd..Issues during the code generation phase

- **Suboptimal code generation**
  - Leads to slower execution or inefficient memory usage
- **Register allocation**
  - In architectures with a limited number of registers, efficient allocation can be challenging
  - Poor register allocation can lead to excessive spills to memory which impacts performance
- **Instruction selection crucial for performance**
  - Suboptimal instruction selection can result in slower code execution or even incorrect behavior

…contd..Issues during the code generation phase

- **Handling of complex data types efficiently**
  - Such as structs, arrays, and pointers
  - Code generation for management of memory layout and accesses for these data types is essential
- **Support for language features and advanced features can be challenging**
  - Features like exceptions, closures, and polymorphism,
- **Missed optimization opportunities such as redundant calculations or unnecessary memory accesses**

…contd..Issues during the code generation phase

- **Constraints and limitations of Target platform**
  - Such as - available instruction set extensions, memory layout, and alignment requirements
- **Exception handling and error recovery** requires careful consideration during code generation
  - Generated code should properly handle exceptions and errors, and ensure correct program behavior in exceptional circumstances,
- **Need to generate code displaying correct interoperability with runtime environments**
  - Such as handling of dynamic memory allocation, garbage collection, and system calls

…contd..Issues during the code generation phase

- **Debugging support**
  - Generating code that facilitates effective debugging can be challenging.
  - Compilers must emit debug information that accurately maps generated code to the original source code, aiding developers in identifying and fixing issues

Basic blocks and flow graphs

Program Analysis for Loop Optimization

- For this – it is important to know control flow relationships between different pieces of code using
  - i) Basic Blocks (group of 3-address codes)
  - ii) Control Flow Graph (Control-flow relationships between basic blocks)

Basic Block

- Sequence of consecutive statements
- Flow of control enters only at the **beginning**
- Flow of control leaves only at the **end**
- No halt in middle
- No branching except at the end

Basic Blocks

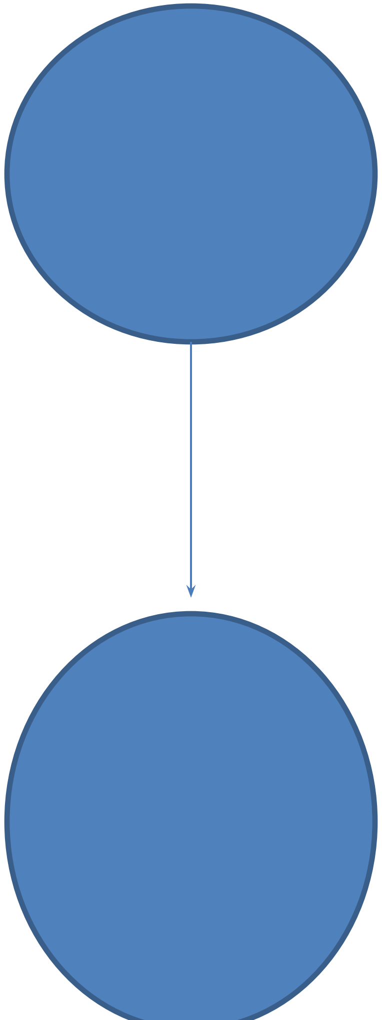
This is a basic block

$$
\begin{aligned}
t_1 &= a*a \\
t_2 &= a*b \\
t_3 &= 2*t_2 \\
t_4 &= t_1 + t_3 \\
t_5 &= b*b \\
t_6 &= t_4 + t_5
\end{aligned}
$$

Three address statement $x = y + z$ is said to define x and to use y and z.

A **name** in a basic block is said to be **live** at a given point, if its value is used after that point in the program, perhaps in another basic block

Basic Blocks – B1, B2



(1) prod = 0
(2) I = 1

(3) t_1=4*I

----
(11) I = t_7
(12) If I <= 20 goto (3)

B_1

B_2

Transformation on Basic Block

- A basic block computes a set of expressions.
- Transformations are useful for improving the quality of code.
- Two important classes of local optimizations that can be applied to a basic blocks
  - Structure Preserving Transformations
  - Algebraic Transformations

Structure Preserving Transformations

- **Common sub-expression elimination**
- **Dead – Code Elimination**

  Say, x is dead, that is never subsequently used, at the point where the statement x = y + z appears in a block.

  We can safely remove x
- **Renaming Temporary Variables**
  - **say, t = b+c where t is a temporary var.**
  - **If we change u = b+c, then change all instances of t to u.**
- **Interchange of Statements**
  - $t_1 = b + c$
  - $t_2 = x + y$
  - **We can interchange iff neither x nor y is t_1 and neither b nor c is t_2**

Structure Preserving Transformations

- **<u>Common sub-expression elimination</u>**

same

```
a  = b + c
b = a – d
c = b + c
d = a - d
```

```
a  = b + c
b = a – d
c = b + c
d = b
```

b is redefined

Algebraic Transformations

- Eliminate following

  X = X + 0 eliminate

  X = X * 1 eliminate

- Replace expensive expressions by cheaper one
  – X = y**2 (why expensive? Answer: Normally implemented by function call)
    - by X = y * y

Algebraic Transformations: Replace expensive expressions by cheaper one

- Replace X + 0 or 0+X by X
- Replace X * 1 or 1* X by X
- Replace X/1 by X
- Replace X = y * y*y*y by X = y**4
- X*2 = X+X
- X/2 = X* 0.5
- 2*11 = 22

Flow Graphs

- A graph representation of three address statements
- Nodes in the flow graph represent **computations**
- Edges represent the **flow of control**

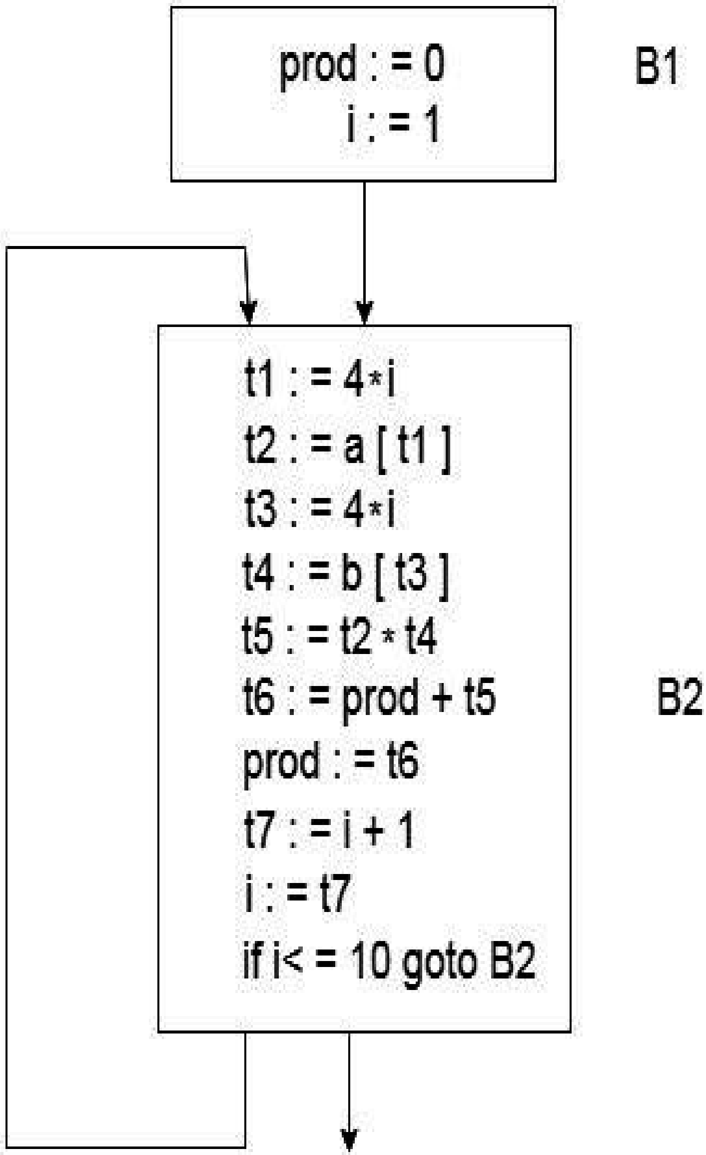
Flow graph

- We can add flow of control information to the set of basic blocks making up a program by constructing **directed graph called flow graph**.
  - There is a directed edge from block B_1 to block B_2 if B2 can immediately follow B1 in execution, i.e. if
    - **There is conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 or**
    - **B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump**

Partitioning into Basic Blocks

- Determine set of Leaders (i.e. first statements in basic block)
  - First statement is leader
  - Target statement of unconditional / conditional goto is leader
  - Statement that immediately follows conditional goto is leader

- For each leader construct a basic block
- Any statement not in any block may be removed



Control Flow graph

- it is a directed graph
- It contains the flow of control information for the set of basic block.
- It is used to depict that how the program control is being parsed among the blocks.
- It is useful in the loop optimization

Loop:

- A loop is a collection of nodes in a flow graph such that
  - All nodes in the collection are strongly connected, that is, from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and
  - There is always one path from a node outside the loop to the node inside the loop i.e it has unique entry

Loops in Flow Graphs

- A loop is a collection of nodes in a flow graph such that
  - All nodes in the collection are strongly connected, i.e. from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and
  - The collection of nodes has a unique entry, that is, a node in the loop such that, the only way to reach a node from a node out side the loop is to first go through the entry.

Exercise

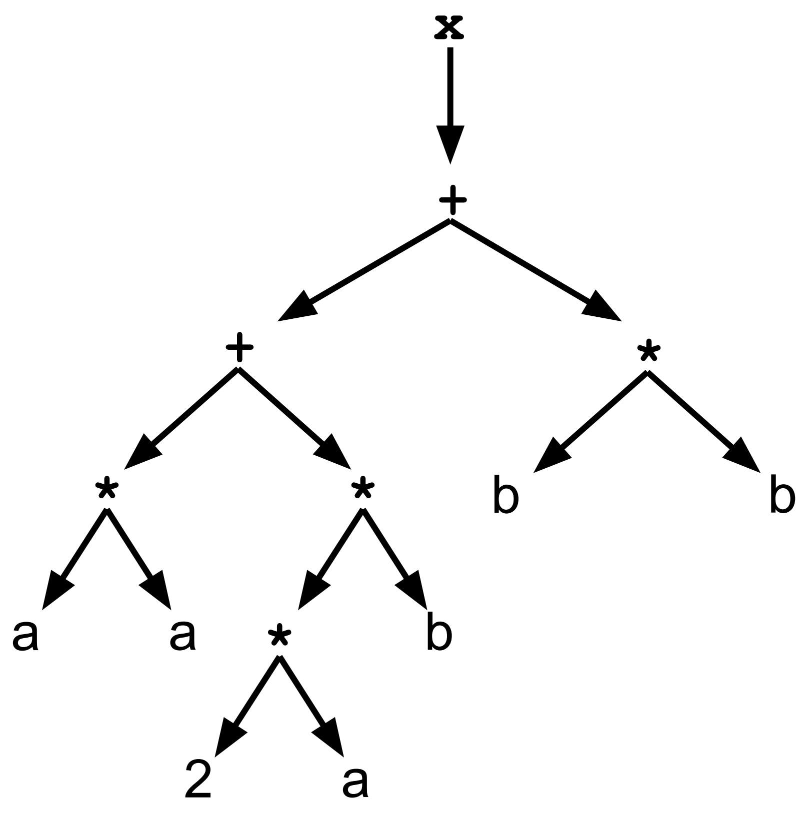
- given the code fragment

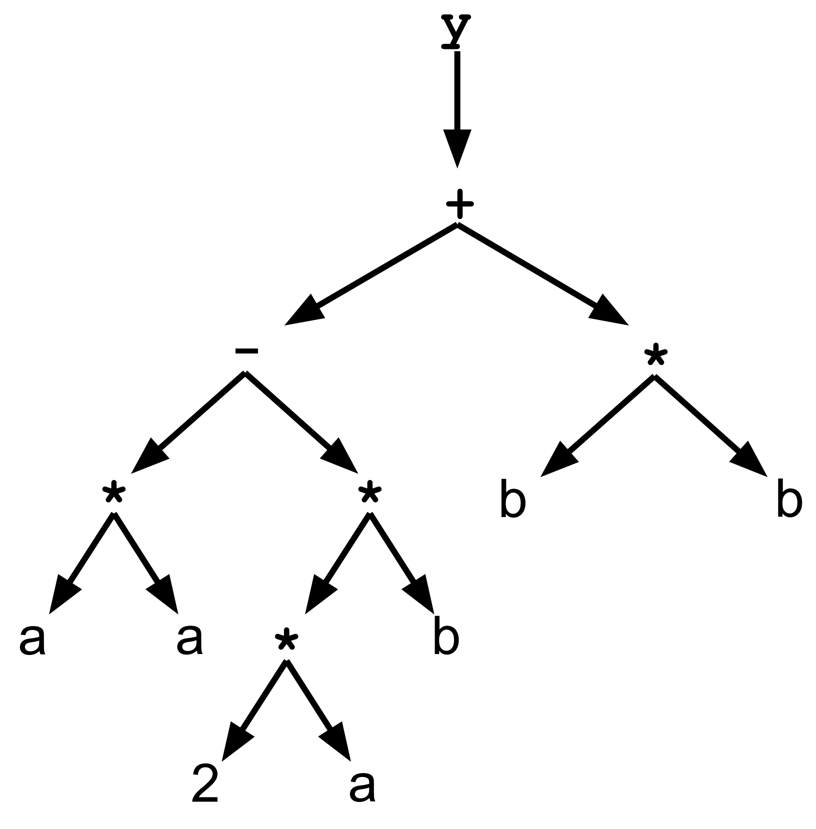
```
x := a*a + 2*a*b + b*b;
y := a*a – 2*a*b + b*b;
```

draw the **dependency graph** before and after common subexpression elimination.

Answers

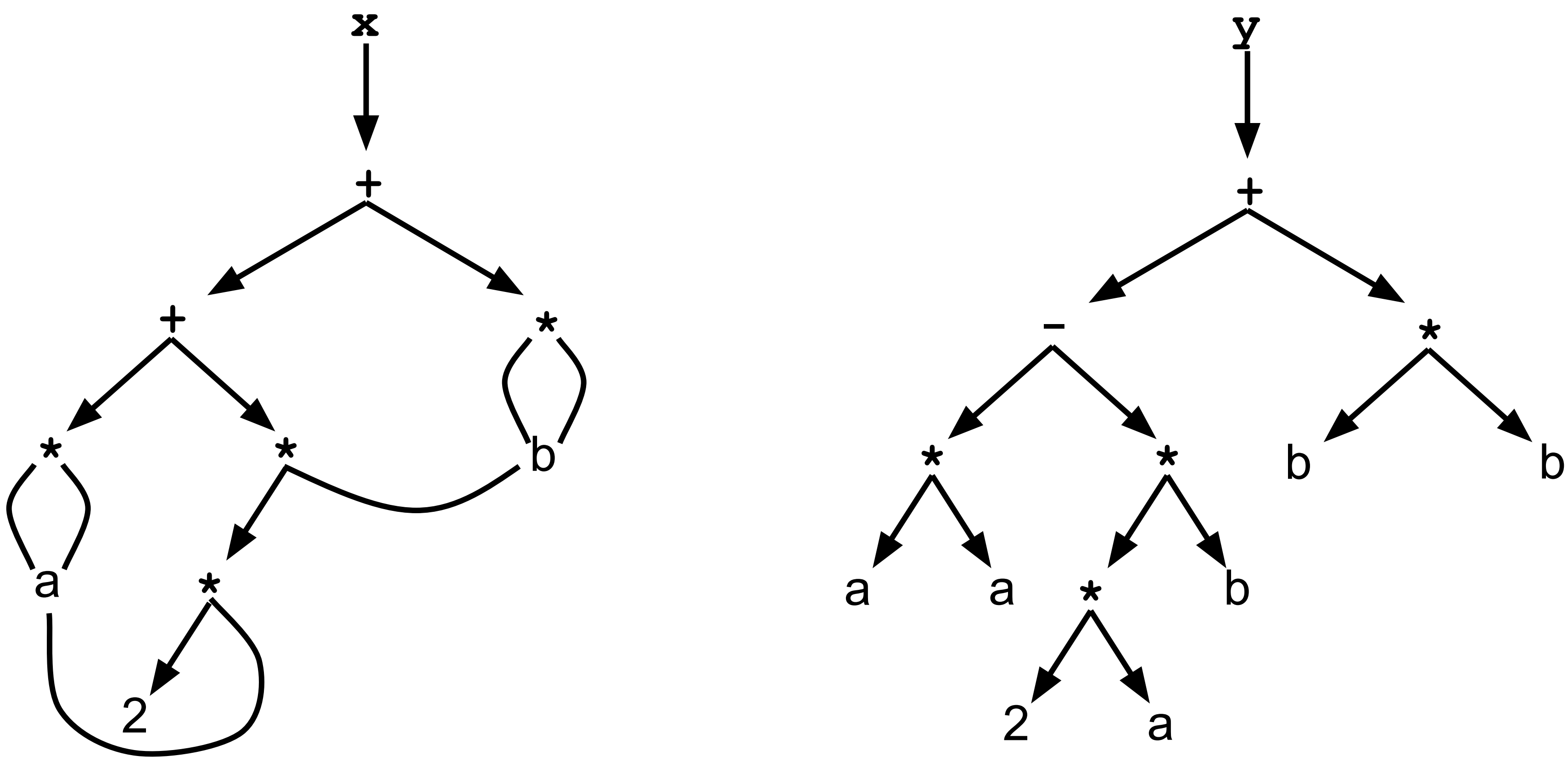
dependency graph before CSE





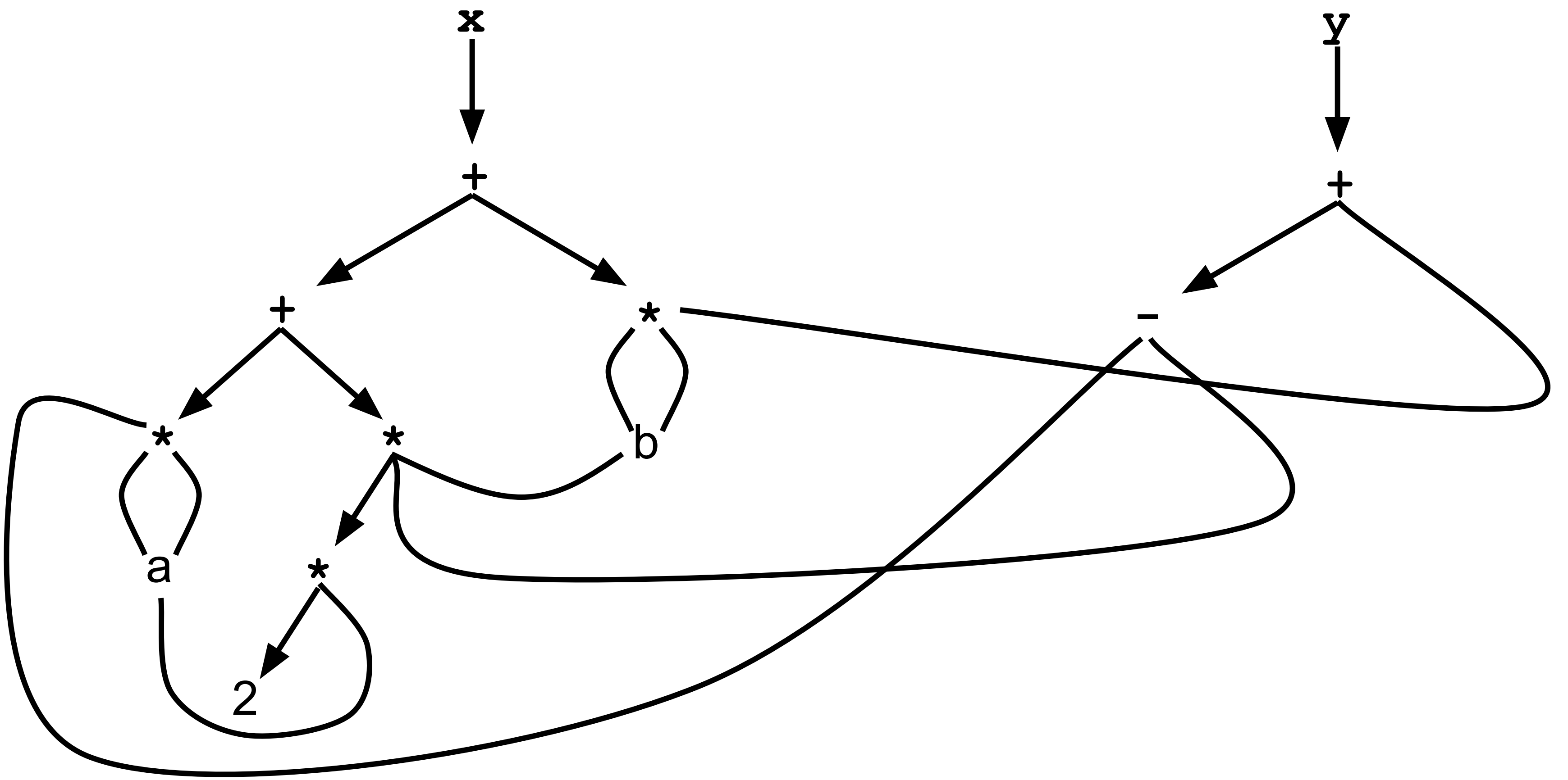
Answers

dependency graph after CSE



Answers

dependency graph after CSE



DAG (Directed Acyclic Graph)

- DAG contains no cycle
- Is **used to optimize the basic block**
- DAG is used to eliminate the common sub-expression.
- It is used to implement transformations on basic blocks.

Algorithm for construction of DAG

- Leaf node represent the identifier, name or constant
- Interior nodes of the graph is labeled by an **operator symbol.**

- Construct the DAG for the given expression

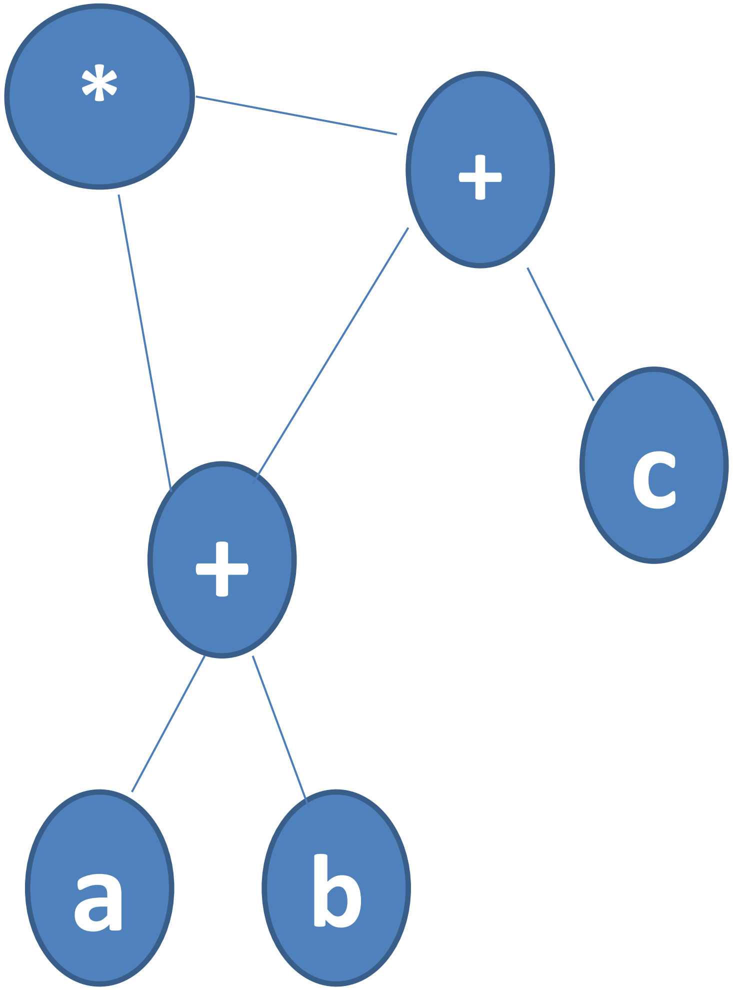
(a + b) * (a+ b+c)

Solution: 3 address code

T1= a+b

T2= T1+c

T3= T1* T2



Labelling algorithm

- **Traversal and Labeling**
  - During traversal of the parse tree/AST generated in the previous phases, the labeling algorithm assigns labels to each node, usually representing basic blocks or points of execution in the control flow graph.

Labelling Algorithm

- **Identify Branch Targets**: The compiler analyzes the control flow of the program to identify points where control may transfer, such as loop headers, conditional branches, or function entry points.
- **Assign Labels**: Once branch targets are identified, the compiler assigns unique labels to each target point. Labels are typically alphanumeric strings or numerical identifiers.
- **Insert Labels into Generated Code**: As the compiler generates machine code, it inserts labels at appropriate locations in the code where control flow may transfer. These labels serve as markers for branch instructions to jump to.
- **Resolve Forward References**: If a branch instruction references a label that hasn't been defined yet (forward reference), the compiler remembers the instruction's location and updates it later when the label is encountered.
- **Output Code with Labels**: Finally, the compiler outputs the generated machine code with labels inserted at appropriate locations, ensuring that the code is properly annotated for control flow.

Sample

if (x < y) { z = x + y; } else { z = x - y; }

AST labelled as A,B,C

```
 A: if (x < y)
     / \
    /   \
   B     C
```

Generating intermediate code for each basic block represented by the labeled nodes

A:
if x < y goto B
goto C

B:
z = x + y
goto D

C:
z = x - y

D:

Sample Code : annotated with comments to demonstrate how labels are inserted

```c
#include <stdio.h>

int factorial(int n) {
   int result = 1;
   int i;

   // Label for loop entry
   LOOP_START:

   for (i = 1; i <= n; i++) {
      // Multiply result by i
      result *= i;
   }

   // Label for loop exit
   LOOP_END:

   return result;
}

int main() {
   int num = 5;
   int result = factorial(num);

   printf("Factorial of %d is %d\n", num, result);

   return 0;
}
```

In this example:

The LOOP_START label marks the entry point of the loop.

The LOOP_END label marks the exit point of the loop.

When the compiler generates machine code for this program, it inserts these labels at appropriate locations. For example, in assembly code:

```
factorial:
   mov eax, 1          ; Initialize result
   mov ecx, 1          ; Initialize loop counter

LOOP_START:
   cmp ecx, [ebp+8]    ; Compare loop counter to n
   jnle LOOP_END       ; Jump to LOOP_END if ecx > n
   imul eax, ecx       ; Multiply result by ecx
   inc ecx             ; Increment loop counter
   jmp LOOP_START      ; Jump to LOOP_START for next iteration

LOOP_END:
   ret

main:
   push 5              ; Push argument num
   call factorial      ; Call factorial function
   ; Print result...
```

- Here, LOOP_START and LOOP_END labels are inserted into the assembly code at appropriate locations to mark the loop entry and exit points, respectively.
- This demonstrates how the labelling algorithm works in the code generation phase of a compiler, enabling proper annotation of control flow in the generated machine code.

A Simple Code Generator

Code Generation

- Requirements imposed on a code generator
  - Preserving the semantic meaning of the source program and being of high quality
  - Making effective use of the available resources of the target machine
  - The code generator itself must run efficiently.
- A code generator has three primary tasks:
  - Instruction selection, register allocation, and instruction ordering

Issues in the Design of a Code Generator

- Details depend on
  - Target language
  - Operating System
- But following issues are inherent in all code generation problems
  - Input to the Code Generator
  - Memory management
  - Instruction Selection
  - Register allocation and
  - Evaluation order
  - Approaches to code generation

A Code-Generation Algorithm

The code-generation algorithm takes as input a sequence of Three-address statements constituting a basic block. For each three-address statement of the form x : = y *op z* perform the Following actions :

1. Invoke a function *getreg t*o determine the location L where the result of the computation *y op z* should be stored. L will usually be a register, but it could also be a memory location.
2. Consult the address descriptor for y to determine *y'*, (one of) the current location(s) 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.* Again, 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 L is a register, update its descriptor to indicate that it contains the value of x. and remove x from all other register descriptors.
4. If the current values of y and/or z have no next uses, 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 no longer will contain y and/or z, respectively.

A Code-Generation Algorithm (Cont'd)

- Consider d := (a-b) + (a-c) + (a-c)
- Three address code sequence is

  t := a-b

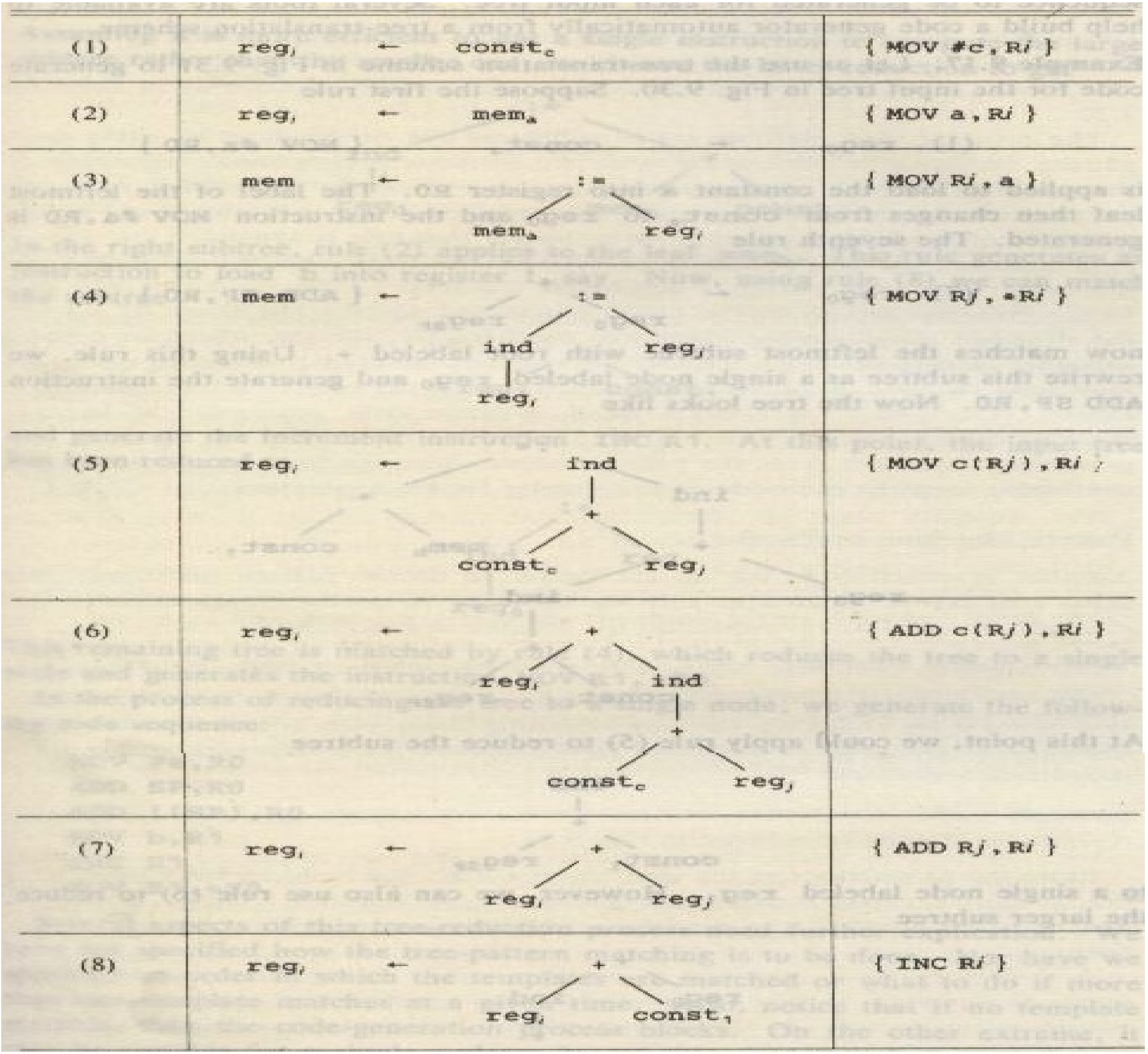
  u := a-c

  v := t+u

  d :=v+u

- Code sequenc

STATEMENTS	CODE GENERATED	REGISTER DESCRIPTOR	ADDRESS DESCRIPTOR
		registers empty	
t := a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u := a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d := v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory



Tree-rewriting for some target-machine instruction

Syntax-directed translation scheme constructed

(1)	$reg_i \rightarrow const_c$		{ MOV #c, Ri }
(2)	$reg_i \rightarrow mem_a$		{ MOV a, Ri }
(3)	$mem \rightarrow := mem_a\ reg_i$		{ MOV Ri, a }
(4)	$mem \rightarrow := ind\ reg_i\ reg_j$		{ MOV Rj, *Ri }
(5)	$reg_i \rightarrow ind + const_c\ reg_j$		{ MOV c(Rj), Ri }
(6)	$reg_i \rightarrow + reg_i\ ind + const_c\ reg_j$		{ ADD c(Rj), Ri }
(7)	$reg_i \rightarrow + reg_i\ reg_j$		{ ADD Rj, Ri }
(8)	$reg_i \rightarrow + reg_i\ const_1$		{ INC Ri }

Common Techniques for Code Optimization Phase of Compiler

Code Optimization Phase

- Crucial in the compilation process
- Compiler transforms the intermediate code produced by the front-end into more efficient code while preserving the semantics of the program.
- Optimization aims to improve various aspects of the generated code, such as **execution speed**, **memory usage**, and **power consumption**.

Code Optimization Types

- Machine Independent Optimization
  (1) Constant Folding / Compile time evaluation
  (2) Code Motion, Removing of loop invariants
  (3) Dead code elimination
  (4) Common Subexpression elimination / Variable Propagation
  –
  – Reduction in strengths (Induction Variable and Strength Reduction)
- Machine dependent Issues
  – Assignment and Use of registers
  – Managing bounded machine resources like registers, functional units, caches

About Optimization Techniques

- Optimization techniques, to be discussed, are often applied in multiple passes, with each pass focusing on specific optimization goals or transformations.
- The effectiveness of optimization depends on factors such as the target architecture, programming language features, and characteristics of the program being compiled

Q) Where is code optimization carried out?

Code optimization is carried out on the **intermediate code** because program analysis is more accurate on intermediate code than on machine code.

Common Subexpression Elimination (CSE)

- Identify and eliminate redundant computations by reusing previously computed expressions.

- For instance, replacing x * 2 with a temporary variable t if x has already been computed

Loop Optimization

- **Loop Unrolling**: Duplicate loop bodies to reduce loop overhead and exploit instruction-level parallelism
- **Loop Fusion**: Combine multiple loops into a single loop to reduce loop overhead and improve cache locality
- **Loop-Invariant Code Motion (LICM)**: Move loop-invariant computations outside the loop to reduce redundant computations

Inlining

- Replace function calls with the body of the called function to reduce the overhead of function call and return.

Control Flow Optimization

- **Branch Prediction**: Rearrange conditional branches to minimize branch mispredictions.
- **Control Flow Graph (CFG) Simplification**: Simplify the control flow graph by eliminating unreachable code, merging basic blocks, or removing redundant branches.

Machine Independent Optimizations

(1) **Compile time** evaluation/ Constant Folding

- x = 5.7

  y = x/3.6

Evaluate x/3.6 as 5.7/3.6 at compile time

- X=55

  Y=2*X

Evaluate 2*X at compile time

Constant Folding

- Evaluate constant expressions at **compile-time** rather than runtime.

- For example, replacing 3 + 4 with 7 during compilation.

(2) Code Motion

- Bring Loop **invariant** statements out of the loop.

Ex Loop optimization:

```
{ int a;
for (i = 0; i < 1000; i++ )
{ /* ... */
a = 10;
 /* ... */
} }
```

if 'a' is local and not used in the loop, then it can be optimized as follows

```
{ int a;
 a = 10;
 for (i = 0; i < 1000; i++ )
{
/* ... */
 } }
```

(3) Dead code elimination

```
int a= 10;
if(a>5)
    {.....}
else
{....}
```

```
int a= 0;
if(a)
{.....}
```

- Compiler knows the value of 'a' at compile time, therefore it also knows that
  - the **if condition is always true**, it can eliminate the else part in the code

Dead Code Elimination

- Remove code that does not contribute to the final output, such as
  - unreachable code or
  - variables that are never used

…contd.

```
X=0;
if(X)
{
.....
}
```

Code in RED, never gets reached, and so can be eliminated

(4) Variable Propagation and common subexpression elimination

- **Common subexpression** - is an expression appearing repeatedly in the program

Example 1

Before optimization

```
int a, b, c;
int x,y;
{
int a, b, c;
int x, y;
/* ... */
x = a + b;
y = a + b + c;
 /* ... */
}
```

Example 1

After optimization

```
{
int a, b, c;
 int x, y;
/* ... */
x = a + b;
 y = x + c; // a + b is replaced by x
/* ... */
 }
```

Example 2??

a = b +c

b= a-d

c=b+c

d= a-d

Example 2??

a = b+c

b= a-d

c=b+c

d=a-d=b

b changed

d uses previous value of d

(5) Induction Variable and Strength Reduction

- Strength reduction is used to replace the **high strength** operator by the low strength
- An induction variable is used in loop

Before Optimization

```
i = 1;
while(i<10)
{
    y = i * 4;
}
```

After Optimization

```
i = 1
t = 4
{
  while( t<40)
  y = t;
  t = t + 4;
}
```

Replacing high strength operator by low strength

X = y**2 (why expensive? Answer: Normally implemented by function call)
Replaced by X = y * y

X=X*2 can be replaced by
X= X+X

X=X/2 can be replaced by
X=X*0.5

Loop Optimization

- Code motion: It brings loop invariant statements out of the loop

- **Induction-variable elimination**

- **Strength reduction**: It is used to replace the expensive operation like multiplication a by the cheaper once like addition.
- **Loop unrolling**: reduce number of jumps, tests
- Loop jamming : merge bodies of two loops

O

A
A
A

Loop Optimization

- Code motion: It brings loop invariant statements out of the loop

- Induction-variable elimination

- Strength reduction: It is used to replace the expensive operation like multiplication a by the cheaper once like addition.
- **Loop unrolling**: reduce number of jumps, tests
- **Loop jamming** : merge bodies of two loops

Peephole optimization

- Peephole optimization is a type of Code Optimization performed on a **small part of the target code (locally improve target code)**
- And replacing these instruction by shorter and faster code
- The small set of instruction or code to which optimization is performed is called **peephole or window**
- It is machine **dependent** optimization

Machine Dependent Optimization

- Machine-dependent optimization occurs after the target code has been generated and transformed to fit the target machine architecture.
- It makes use of CPU registers and may make use of absolute memory references rather than relative memory references.

Peephole optimization

The objective of peephole optimization is:

- To improve performance
- To reduce memory footprint
- To reduce code size

Peephole optimization techniques

- **Redundant load and store elimination**
- **Flow of Control Optimization**
- **Strength Reduction**
- **Use of Machine Idioms**
- **Null sequence**
- **Combine operations**

Peephole optimization Techniques

- **Redundant load and store elimination**

```
Initial code
y = x + 5;
i = y;
z = i;
w = z * 3;
```

```
Optimized code
y = x + 5;
i = y;
w = y * 3;
```

Mov R1, Y
Mov Y , R1

Eliminate Unreachable instructions

- **Null sequences:**
  Unusable operations are deleted

Algebraic Simplification

- Eliminate A=A+0;   A=A*1;   A=A/1;   etc

Peephole optimization Techniques

- **Flow of control optimization**

if cond true goto 100

....

100 goto 112

.....

112

...

Optimized to

if cond true goto 112

.....

112

....

Peephole optimization Techniques

Strength Reduction

- Y=x*2   optimized to y=x+x
- Multiplication – div replaced with addition subtraction

Peephole optimization Techniques

Use of Machine Idioms

- When the target instructions have equivalent machine instructions for performing operations, we can replace target instructions with their equivalent machine instructions, to improve efficiency
- **Combine operations:**
  Many operations are replaced by a single equivalent operation, as per target instruction availability (due to advanced resources Functional Units)

Machine Dependent Optimization

Optimization for Specific Architectures

- Apply architecture-specific optimizations, such as exploiting vectorization, instruction pipelining, or cache hierarchies

Global Optimization

- Perform optimizations that analyze the entire program rather than individual functions or basic blocks.
- Examples include interprocedural analysis and optimization

Data Flow Analysis

- **Register Allocation**: Assign variables to processor registers to minimize memory accesses and improve performance.
- **Data Dependency Analysis**: Analyze dependencies between instructions to identify opportunities for parallel execution or instruction reordering.