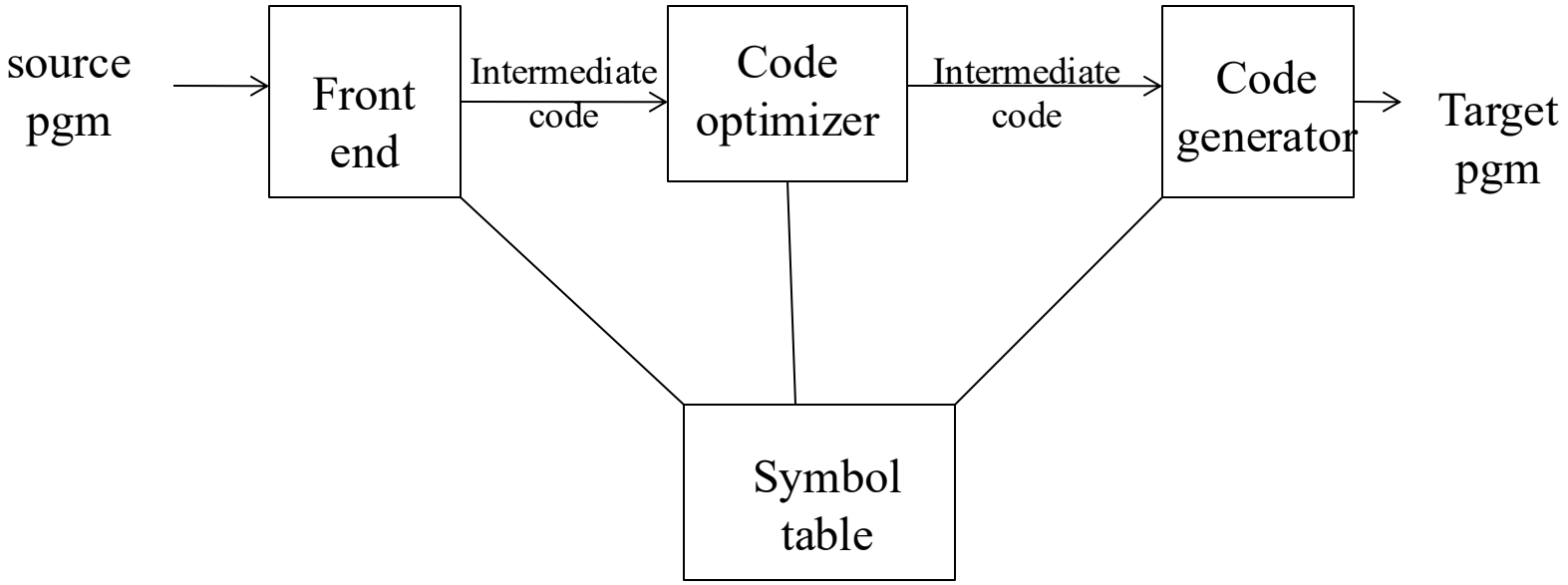
UNIT V

- **Code generation:** Issues in the Design of a Code Generator. The Target Machine, Basic Blocks and Flow Graphs, a simple Code Generator, Peephole optimization.
- **Machine independent optimization:** Data Flow Analysis, Constant Propagation, Live Variable Analysis, Loops. Error recovery in various phases.
- **Advanced topics:** Review of Compiler Structure, Advanced elementary topics, Structure of optimizing compilers.

Introduction

- 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.
- The requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.

Position of code generator



Issues in the Design of a Code Generator

- depends on
  - Target language
  - Operating System
- But following issues are inherent in all code generation problems
  - Input to the code generator
  - Target programs
  - Memory management
  - Instruction Selection
  - Register allocation and
  - Evaluation order

Input to the Code Generator

- We assume, front end has
  - Scanned, parsed and translate the source program into a reasonably detailed intermediate representations
  - Type checking, type conversion and obvious semantic errors have already been detected
  - Symbol table is able to provide run-time address of the data objects
  - Intermediate representations may be
    - Linear representations such as Postfix notations
    - Three address representations such as quadruples
    - Stack machine code
    - virtual machine representations such as Syntax tree, DAG

Target Programs

- The output of the code generator is the target program.
- Target program may be
  - Absolute machine language
    - It can be placed in a fixed location of memory and immediately executed
  - Re-locatable machine language
    - Subprograms to be compiled separately
    - A set of re-locatable object modules can be linked together and loaded for execution by a linker
  - Assembly language
    - Easier

Factors influencing Optimization

- The target machine: machine dependent factors can be parameterized to compiler for fine tuning
- Architecture of Target CPU
  - Number of CPU registers
  - RISC vs CISC
  - Pipeline Architecture
  - Number of functional units

- Machine Architecture
  - Cache Size and type
  - Cache/Memory transfer rate

Memory Management

Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator. We assume that a name in a three-address statement refers to a symbol table entry for the name.

Instruction Selection

- The nature of the instruction set of the target machine determines the difficulty of the instruction selection.
- Uniformity and completeness of the instruction set are important
- Instruction speeds and machine idioms are other important factors. Say, x = y + z

```
Mov  y, R0
Add  z, R0
Mov  R0, x
```

This kind of statement by statement code generation often produces poor code

Instruction Selection contd.

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

would be translated into

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

Here the fourth statement is redundant, and so is the third if ‘a’ is not subsequently used.

Instruction Selection contd.

- The quality of the generated code is determined by its speed and size.
- Cost difference between the different implementation may be significant.
  - Say a = a + 1

    Mov a, R0

    Add #1, R0

    Mov R0, a
  - If the target machine has increment instruction (INC), we can write INC a rather than by a more obvious sequence that loads a into a register, add one to the register, and then stores the result back into a.

- Instruction speeds are needed to design good code sequence but unfortunately, accurate timing information is often difficult to obtain. Deciding which machine code sequence is best for a given three address construct may also require knowledge about the context in which that construct appears.

Register Allocation

- Instructions involving register operands are usually shorter and faster than those involving operands in memory.
- Efficient utilization of register is particularly important in code generation.
- The use of register is subdivided into two sub problems
  - During register allocation, we select the set of variables that will reside in register at a point in the program.
  - During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

- Finding an optimal assignment of registers to variables is difficult, even with single register values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.
- Certain machines require **register pairs** (an even and next odd numbered register) for some operands and results.

```
t := a + b          t := a + b
t := t * c          t := t + c
t := t / d          t := t / d
```

(a) **(b)**

Two three address code sequences

```
L   R1, a           L   R0, a
A   R1, b           A   R0, b
M   R0, c           A   R0, c
D   R0, d           SRDA  R0, 32
ST  R1, t           D     R0, d
                    ST    R1, t
```

(a) **(b)**

Optimal machine code sequence

Choice of Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

Picking a best order is another difficult, NP-complete problem. We can avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

Approaches to Code Generation

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that code generator must face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

Target Machine

Two-address instruction form - op source, destination

The address modes together with their assembly-language forms and associated

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

Instruction Costs

- Cost of an instruction = 1 + costs of source and destination addressing modes
- This cost corresponds to the length (in words) of the instruction
- Minimize instruction length also tend to minimize the instruction execution time

Instruction Costs

Instruction	Cost
MOV R0 R1	1
MOV R5 M	2
MOV a b	3

Instruction Costs

- Example
- a :=b+c
- This statement can be implemented by different instruction sequences

	INSTRUCTION		COST
1.	MOV b, R0	(2)	
	ADD c, R0	(2)	
	MOV R0, a	(2)	6
2.	MOV b, a	(3)	
	ADD c, a	(3)	6
3.	MOV *R1,*R2	(1)	
	ADD *R2,*R0	(1)	2
4.	ADD R2,R1	(1)	
	MOV R1,a	(2)	3

Basic Blocks and Flow Graphs

- A graph representation of three address statements, called flow graph.
- Nodes in the flow graph represent computations called basic block
- Edges represent the flow of control

Basic Block

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibly of the branching except at the end.

Basic Blocks and Flow Graphs

Basic Block

- **Partition into basic blocks**
  - **Method**
    - **We first determine the leader**
      - **The first statement is a leader**
      - **Any statement that is the target of a conditional or unconditional goto is a leader**
      - **Any statement that immediately follows a goto or unconditional goto statement is a leader**
    - **For each leader, its basic block consists of the leader and all the statements up to but not including the next leader or the end of the program.**

Flow Graph

- The flow of control information can be added to the set of basic blocks making up a program by constructing a directed graph called a flow graph.
- The nodes of the flow graph are the basic blocks. One node is distinguished as initial; it is the block whose leader is the first statement.
- There is a directed edge from block B1 to block B2 if B2 can immediately follow B1 in some execution sequence, if
  1. there is a conditional or unconditional jump from the last statement of B1 to the first statement of B2, or
  2. B2 immediately follows B1 in the order of the program and B1 does not end in an conditional jump.

We say that B1 is a predecessor of B2 and B2 is a successor of B1.

Sample: Quicksort

```
// assume an external input-output array: int a[]
void quicksort( int m, int n ) {
   int i, j, v, x; // temps
   if ( n <= m ) return;
```

```
//fragment begins
   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;
   } //end while
```

```
   x = a[i]; a[i] =a [n]; a[n] = x;
   quicksort( m, j );
   quicksort( i+1, n );
} //end quicksort
```

Three address code for fragment

```
 (1) i := m-1
 (2) j := n
 (3) t1 := 4*n
 (4) v := a[t1]
L1:
 (5) i := i+1
 (6) t2 := 4*i
 (7) t3 := a[t2]
 (8) if t3<v goto L1
L2:
 (9) j := j-1
 (10) t4 := 4*j
 (11) t5 := a[t4]
 (12) if t5>v goto L2
 (13) if i>=j goto L3
 (14) t6 := 4*i
 (15) x := a[t6]
```

```
(16) t7 := 4*i
(17) t8 := 4*j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4*j
(21) a[t10] := x
 (22) goto L1
     L3:
(23) t11 := 4*i
(24) x := a[t11]
(25) t12 := 4*i
(26) t13 := 4*j
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4*j
(30) a[t15] := x
```

Three address code for fragment (with leader)

```
 (1) i := m-1
 (2) j := n
 (3) t1 := 4*n
 (4) v := a[t1]
L1:
 (5) i := i+1
 (6) t2 := 4*i
 (7) t3 := a[t2]
 (8) if t3<v goto L1
L2:
 (9) j := j-1
 (10) t4 := 4*j
 (11) t5 := a[t4]
 (12) if t5>v goto L2
 (13) if i>=j goto L3
 (14) t6 := 4*i
 (15) x := a[t6]
 (16) t7 := 4*i
 (17) t8 := 4*j
 (18) t9 := a[t8]
 (19) a[t7] := t9
 (20) t10 := 4*j
 (21) a[t10] := x
 (22) goto L1
      L3:
(23) t11 := 4*i
(24) x := a[t11]
(25) t12 := 4*i
(26) t13 := 4*j
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4*j
(30) a[t15] := x
```

Quicksort Control Flow Graph

B1: 1-4

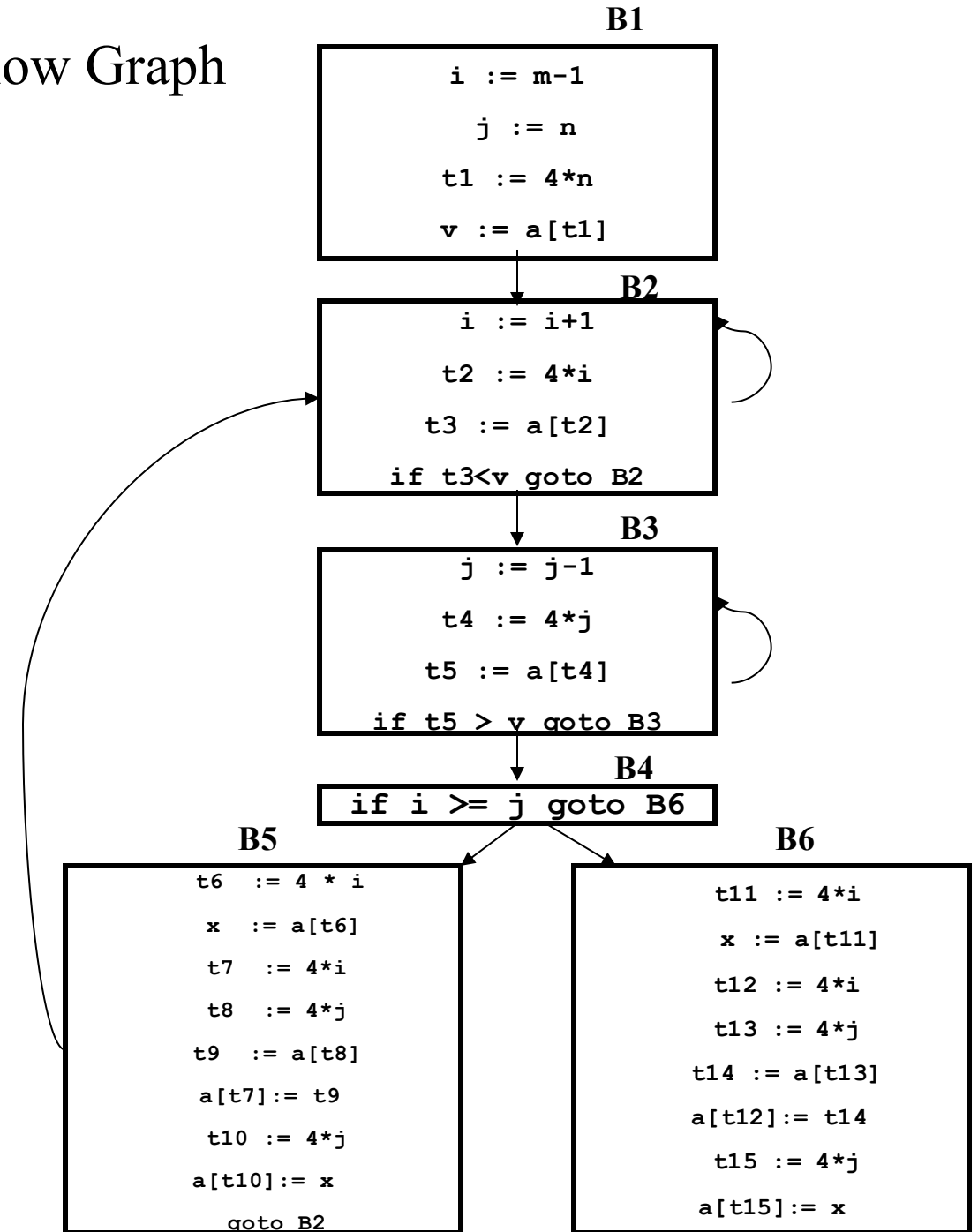
B2: 5-8

B3: 9-12

B4: 13

B5: 14-22

B6: 23-30



Simple code generator

Register and Address descriptors:

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

- A register descriptor keeps track of what is currently in each register
- An address descriptor keeps track of the location where the current value of the name can be found at run time

A code generation algorithm

For each 3-address statement of the form x:=y op z we perform 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 y1 .Prefer the register for y1 if the valuue of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y1 , L
3. Generate the instruction op z1 , L where z1 is a current location of Z

A code generation algorithm contd.

The function getreg :

The function getreg returns the location L to hold the value of x for the assignment x:=y op z

1.If y is in register, and y is not live after execution of x:=y op z then return the register of y for L.

2. Failing (1) , return an empty register for L

3. Failing (2) and x is not used in the block, return memory location of x as L

Code sequences for an assignment statement

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

Three address code:

t := a –b

u := a – c

v := t + u

d := v + u

Consider d live at end.

Code sequence

Statement	Code generated	Register descriptor	Address descriptor
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

Code sequences for indexed assignments

Statement	i IN REGISTER Ri		i IN MEMORY Mi		i IN STACK	
	CODE	COST	CODE	COST	CODE	COST
a:= b[i]	MOV b(Ri), R	2	MOV Mi , R MOV b(R), R	4	MOV Si(A), R MOV b(R), R	4
a[i] := b	MOV b, a(Ri)	3	MOV Mi , R MOV b , a(R)	5	MOV Si(A), R MOV b, a(R)	5

Code sequences for pointer assignments

Statement	p IN REGISTER Rp		p IN MEMORY Mp		p IN STACK	
	CODE	COST	CODE	COST	CODE	COST
a:= *p	MOV *Rp , a	2	MOV Mp , R MOV *R , R	3	MOV Sp(A), R MOV *R , R	3
*p := a	MOV a , *Rp	2	MOV Mp , R MOV a , *R	4	MOV a , R MOV R , Sp(A)	4

Peephole Optimization

1. **A Simple but effective technique for locally improving the target code is peephole optimization,**
2. **a method for trying to improve the performance of the target program**
3. **by examining a short sequence of target instructions and replacing these instructions by a shorter or faster sequence whenever possible.**

Characteristics of peephole optimization

1. **Redundant instruction elimination**
2. **Flow of control information**
3. **Algebraic Simplification**
4. **Use of machine Idioms**

Peephole Optimization

Redundant Loads and Stores:

1) MOV R0 , a

2) MOV a , R0 **eliminate**

Unreachable code:

An unlabeled instruction immediately following an unconditional jump may be removed

goto L1

x := x + 1 ← No need

L1:

Peephole Optimization

- **Flow of control optimizations**

  goto L1 becomes goto L2

  …

  L1: goto L2 ← No needed if no other L1 branch

- **Algebraic Simplification**

  x := x + 0 ← No needed

  or

  x := x * 1

- **Use of Machine idioms**

  Use auto increment or auto decrement addressing modes to add or subtract one from an operand Use machine specific hardware instruction which may be less costly.

  i := i + 1

  ADD i, # 1 INC i

(Classifications of Optimization techniques)

- Peephole optimizations
- Local optimizations

  Optimization of Basic Blocks
- Global optimizations
  - Inter-procedural
  - Intra-procedural
- Loop optimizations

(Themes behind Optimization Techniques)

- Avoid redundancy: something already computed need not be computed again
- Smaller code: less work for CPU, cache, and memory!
- Less jumps: jumps interfere with code pre-fetch
- Code locality: codes executed close together in time is generated close together in memory – increase locality of reference
- Extract more information about code: More info – better code generation

(Redundancy elimination)

- Redundancy elimination is determining that two computations are equivalent and eliminating one.
- There are several types of redundancy elimination:
  - Value numbering
    - Associates symbolic values to computations and identifies expressions that have the same value
  - Common subexpression elimination
    - Identifies expressions that have operands with the same name
  - Constant/Copy propagation
    - Identifies variables that have constant/copy values and uses the constants/copies in place of the variables.
  - Partial redundancy elimination
    - Inserts computations in paths to convert partial redundancy to full redundancy.

(Compile-Time Evaluation)

- Expressions whose values can be pre-computed at the compilation time
- Two ways:
  - Constant folding
  - Constant propagation

Ex.Constant Folding :

X=32;
X=X+32;

These statements are replaced by

X=64;

Main Types of Code Optimization are

- High-level optimizations, intermediate level optimizations, and low-level optimizations .
- High-level optimization is a language dependent type of optimization that operates at a level in the close vicinity of the source code.
- High-level optimizations include in lining where a function call is replaced by the function body and partial evaluation which employs reordering of a loop, alignment of arrays, padding, layout, and elimination of tail recursion.

Intermediate level Optimization

- Most of the code optimizations performed fall under **intermediate code optimization** which is language independent. This includes:

1. The elimination of common sub-expressions
2. Constant propagations.
3. Jump threading
4. Loop invariant code motion
5. Dead code elimination
6. Strength reduction

Intermediate level Optimization

- Most of the code optimizations performed fall under **intermediate code optimization** which is language independent. This includes:

1. **The elimination of common sub-expressions** – This type of compiler optimization probes for the instances of identical expressions by evaluating to the same value and researches whether it is valuable to replace them with a single variable which holds the computed value.
2. **Constant propagations** – Here, expressions which can be evaluated at compile time are identified and replaced with their values.
3. **Jump threading** – This involves an optimization of jump directly into a second one. The second condition is eliminated if it is an inverse or a subset of the first which can be done effortlessly in a single pass through the program. Acyclic chained jumps are followed till the compiler arrives at a fixed point.

- 4. **Loop invariant code motion** – This is also known as hoisting or scalar promotion. A loop invariant contains expressions that can be taken outside the body of a loop without any impact on the semantics of the program. The above-mentioned movement is performed automatically by loop invariant code motion.

5**. Dead code elimination** – Here, as the name indicates, the codes that do not affect the program results are eliminated. It has a lot of benefits including reduction of program size and running time. It also simplifies the program structure. Dead code elimination is also known as DCE, dead code removal, dead code stripping, or dead code strip.

6. Strength reduction – This compiler optimization replaces expensive operations with equivalent and more efficient ones, but less expensive. For example, replace a multiplication within a loop with an addition.

Low-level Optimization

- Low-level Optimization is highly specific to the type of architecture. This includes the following:

1. Register allocation
2. Instruction Scheduling
3. Floating-point units utilization
4. Branch prediction
5. Peephole and profile-based

- Low-level Optimization is highly specific to the type of architecture. This includes the following:

1. **Register allocation** – Here, a big number of target program variables are assigned to a small number of CPU registers. This can happen over a local register allocation or a global register allocation or an inter-procedural register allocation.

2. **Instruction Scheduling** – This is used to improve an instruction level parallelism that in turn improves the performance of machines with instruction pipelines. It will not change the meaning of the code but rearranges the order of instructions to avoid pipeline stalls. Semantically ambiguous operations are also avoided.

3. **Floating-point units utilization** – Floating point units are designed specifically to carry out operations of floating point numbers like addition, subtraction, etc. The features of these units are utilized in low-level optimizations which are highly specific to the type of architecture.

4. Branch prediction – Branch prediction techniques help to guess in which way a branch functions even though it is not known definitively which will be of great help for the betterment of results.

5. Peephole and profile-based optimization – Peephole optimization technique is carried out over small code sections at a time to transform them by replacing with shorter or faster sets of instructions. This set is called as a peephole. Profile-based optimization is performed on a compiler which has difficulty in the prediction of likely outcome of branches, sizes of arrays, or most frequently executed loops. They provide the missing information, enabling the compilers to decide when needed.

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

Overview of Machine Dependent Optimization

- Register Allocation
- Instruction Scheduling (Addressing modes)
- Peephole optimization

1. Redundant instruction elimination
2. Unreachable code
3. Flow of control information
4. Algebraic simplification
5. Strength reduction
6. Use of machine Idioms

Machine Independent Code optimizations

- **Function preserving**
- **Loop Optimizations**

- **Function Preserving**
  - Common Subexpression elimination
  - Folding
  - Dead code elimination
  - Copy Propagation

- **Loop Optimizations**
  - Frequency Reduction
    - ✓ Code Motion
    - ✓ Loop Unrolling
    - ✓ Loop Jamming
  - Algebraic expression simplification
  - Strength Reduction
  - Redundancy Elimination

Conditional statements

Conditional statements are implemented using condition codes

Ex : if x<y goto z is implemented as

```
CMP   x , y
CJ<   z
```

Register Allocation and Assignment

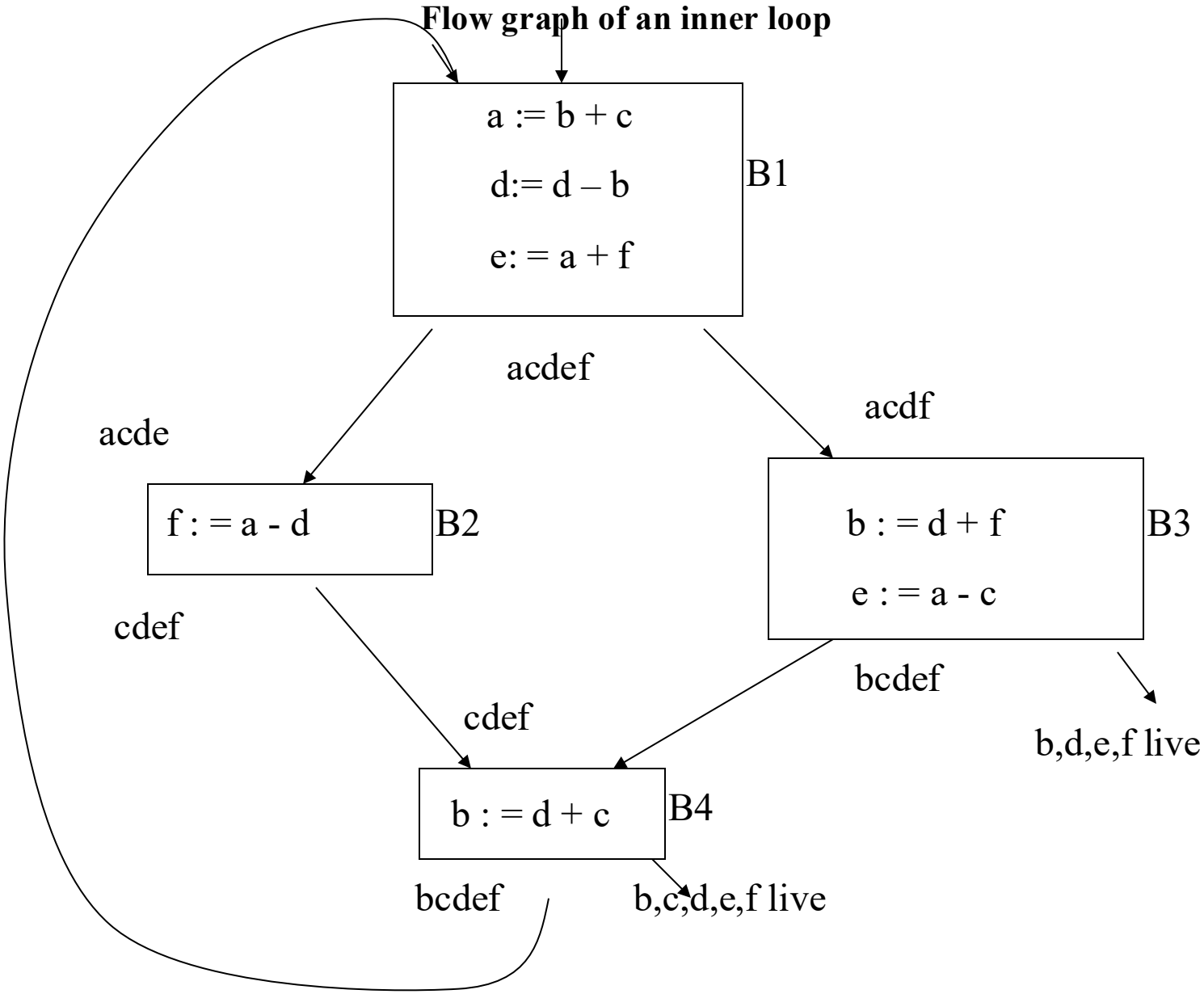
An approximate formula for the benefit to be realized from allocating a register to x within loop L is:

$$\sum_{\text{Blocks B in L}} (use (x , B) + 2 * live (x , B))$$

Where use (x , B) is the number of times x is used in B prior to any definition of x

live (x , B) is 1 if x is live on exit from B and assigned a value in B, and live (x, B) is 0 otherwise

Example- Register Allocation and Assignment



use $(a, B_1) + 2 * live(a, B_1) = 0 + (2 * 1) = 2$

use $(a, B_2) + 2 * live(a, B_2) = 1 + (2 * 0) = 1$

use $(a, B_3) + 2 * live(a, B_3) = 1 + (2 * 0) = 1$

use $(a, B_4) + 2 * live(a, B_4) = 0 + (2 * 0) = 0$

$= 4$

use $(b, B_1) + 2 * live(b, B_1) = 2 + (2 * 0) = 2$

use $(b, B_2) + 2 * live(b, B_2) = 0 + (2 * 0) = 0$

use $(b, B_3) + 2 * live(b, B_3) = 0 + (2 * 1) = 2$

use $(b, B_4) + 2 * live(b, B_4) = 0 + (2 * 1) = 2$

$= 6$

a = 4
b = 6
c = 3
d = 6
e = 4
f = 4

R_0	R_1	R_2
b	d	a e f

Main idea

- Want to replace temporary variables with some fixed set of registers
- First: need to know which variables are live after each instruction
  - Two simultaneously live variables cannot be allocated to the same register