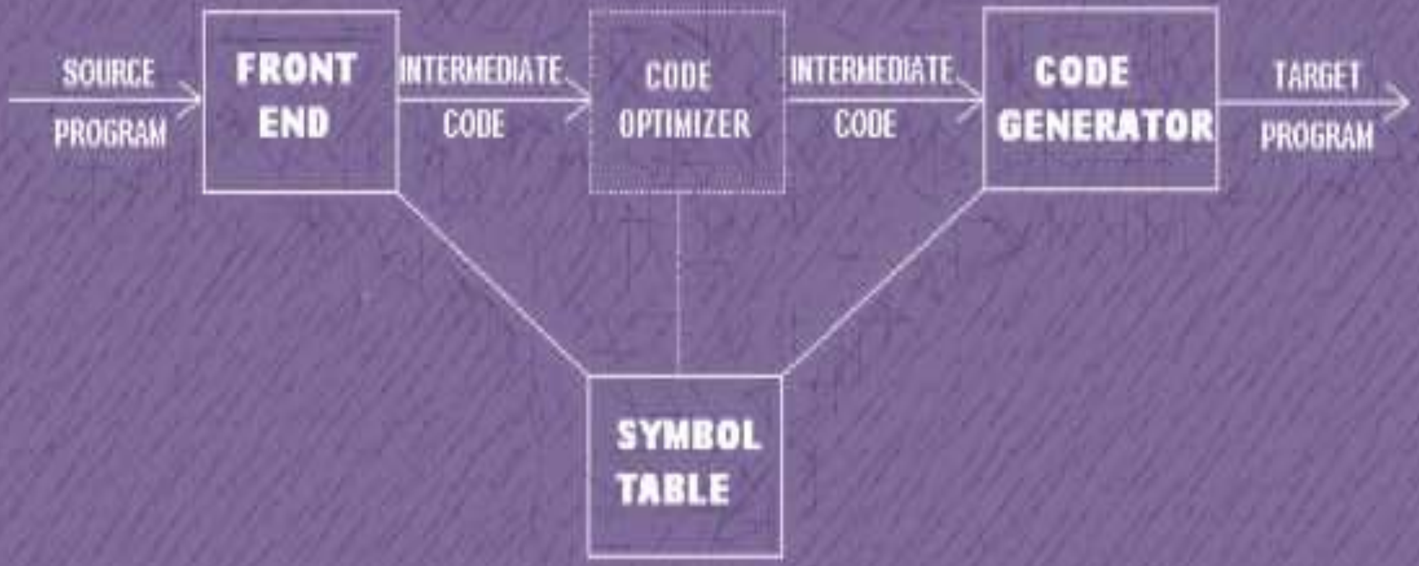
CODE GENERATION

Introduction

- The final phase of our compiler model is code generator.
- It takes input from the intermediate representation with supplementary information in symbol table of the source program and produces as output an equivalent target program.
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering

- Instruction selection
    - choose appropriate target-machine instructions to implement the IR statements
- Register allocation and assignment
    - decide what values to keep in which registers
- Instruction ordering
    - decide in what order to schedule the execution of instructions

POSITION OF CODE GENERATOR



Issues in the design of code generator

- Input to the code generator
- Target program
- Memory management
- Instruction selection
- Register allocation
- Choice of evaluation order
- Approaches to code generation

Issues in the design of code generator

- Input to the code generator
  - IR + Symbol table
  - IR has several choices
    - Postfix notation
    - Syntax tree
    - Three address code
  - We assume front end produces low-level IR, i.e. values of names in it can be directly manipulated by the machine instructions.
  - Syntactic and semantic errors have been already detected

Issues in the design of code generator

- The target program
  - The output of code generator is target program.
  - Output may take variety of forms
    - Absolute machine language(executable code)
    - Relocatable machine language(object files for linker)
    - Assembly language(facilitates debugging)
  - Absolute machine language has advantage that it can be placed in a fixed location in memory and immediately executed.
  - Relocatable machine language program allows subprograms to be compiled separately.
  - Producing assembly language program as o/p makes the process of code generation somewhat easier.

Issues in the design of code generator

- Memory management
  - Mapping names in the source program to addresses of data objects in run time memory is done by front end & code generator.
  - If a machine code is being generated, labels in three address statements have to be converted to addresses of instructions. This process is analogous to “backpatching” technique.

Issues in the design of code generator

- Instruction selection
  - Uniformity and completeness of the instruction set are imp factors
  - If we do not care about the efficiency of the target program, instruction selection is straightforward.
  - The quality of the generated code is determined by its speed and size.
  - For ex,

x=y+z

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

a=b+c
d=a+e

```
LD      R0, b
ADD     R0, R0, c
ST      a, R0
LD      R0, a
ADD     R0, R0, e
ST      d, R0
```

Issues in the design of code generator

- Register allocation
  - Instructions involving register operands are usually shorter and faster than those involving operands in memory.
  - Two subproblems
    - Register allocation: select the set of variables that will reside in registers at each point in the program
    - Register assignment: select specific register that a variable reside in
  - Complications imposed by the hardware architecture
    - Example: register pairs for multiplication and division

- The multiplication instruction is of the form

  M x, y

where x, is the multiplicand, is the even register of an even/odd register pair.

- The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

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

```
t=a+b
t=t+c
T=t/d
```

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

```
L       R0, a
A       R0, b
A       R0, c
SRDA    R0, 32
(shift right double arithmetic)
D       R0, d
ST      R1, t
```

INC a

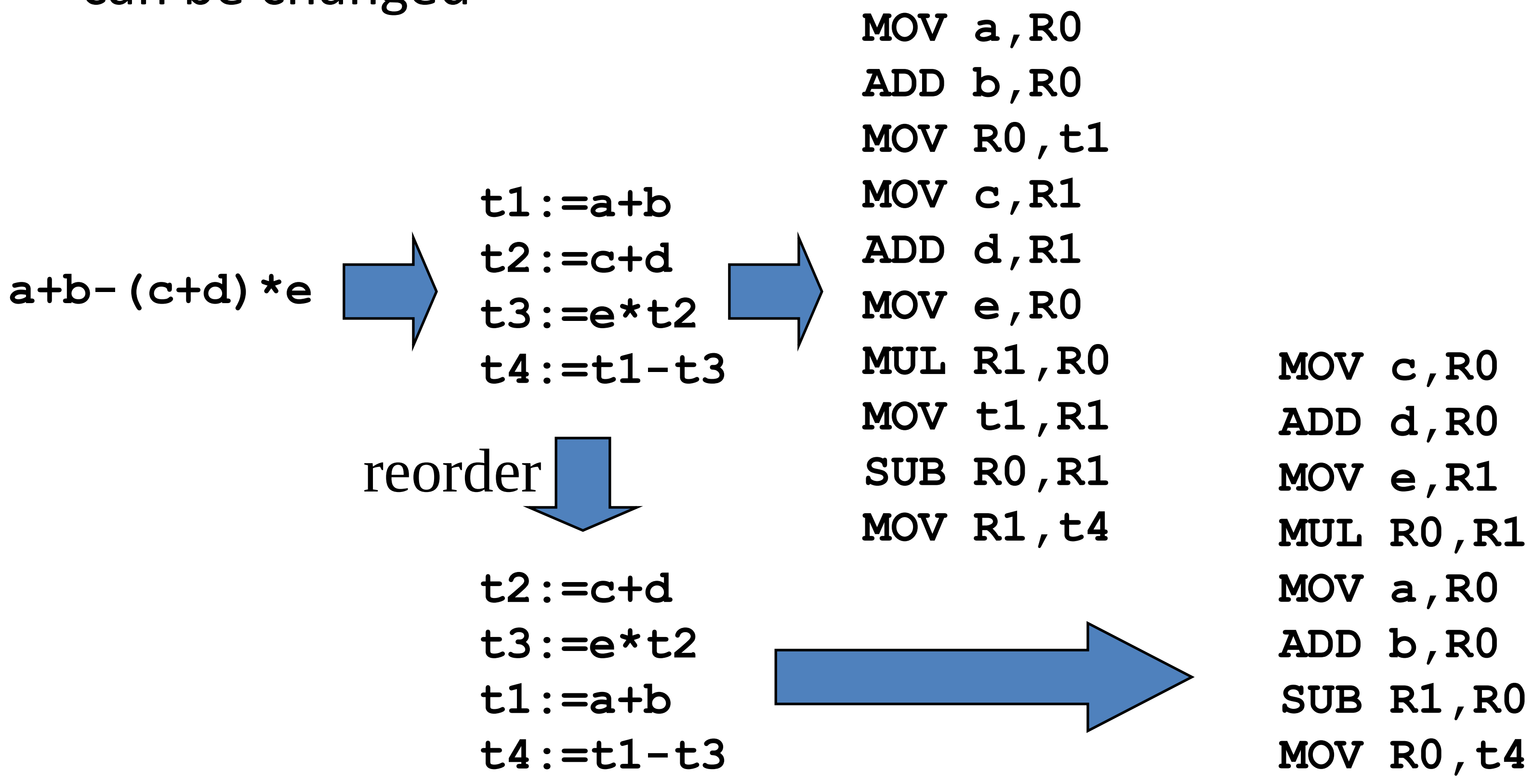
```
MOV   a, R0
ADD   #1,R0
MOV   R0, a
```

Issues in the design of code generator

- Approaches to code generator
  - Criterion for a code generator is to produce correct code.
  - Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

Issues in the design of code generator

- Choice of evaluation order
  - The order in which computations are performed can affect the efficiency of the target code.
  - When instructions are independent, their evaluation order can be changed



```
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
```

```
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
```

Target machine

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set

- Our (hypothetical) machine:
  - Byte-addressable (word = 4 bytes)
  - Has *n* general purpose registers **`R0`**, **`R1`**, …, **`R`***n*-1
  - Two-address instructions of the form

    op source, *destination*
  - *Op* – op-code
  - *Source, destination* – data fields

The Target Machine: Op-codes

- Op-codes (op), for example
- MOV (move content of source to destination)
- ADD (add content of source to destination)
- SUB (subtract content of source from destination)

- There are also other ops

The Target Machine: Address modes

- **Addressing mode**: Different ways in which location of an operand can be specified in the instruction.

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
Literal	**`#`***c*	N/A	1

Instruction Costs

- Machine is a simple processor with fixed instruction costs
- In most of the machines and in most of the instructions the time taken to fetch an instruction from memory exceeds the time spent executing the instruction. So reducing the length of the instruction has an extra benefit.

Examples

Instruction	Operation	Cost
`MOV R0,R1`	Store *content*(**`R0`**) into register **`R1`**	1
`MOV R0,M`	Store *content*(**`R0`**) into memory location **`M`**	2
`MOV M,R0`	Store *content*(**`M`**) into register **`R0`**	2
`MOV 4(R0),M`	Store *contents*(4+*contents*(**`R0`**)) into **`M`**	3
`MOV *4(R0),M`	Store *contents*(*contents*(4+*contents*(**`R0`**))) into **`M`**	3
`MOV #1,R0`	Store 1 into **`R0`**	2
`ADD 4(R0),*12(R1)`	Add *contents*(4+*contents*(**`R0`**)) to value at location *contents*(12+*contents*(**`R1`**))	3

Instruction Selection

- Instruction selection is important to obtain efficient code
- Suppose we translate three-address code

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

```
a:=a+1
```

$\Rightarrow$

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

Cost = 6

Better $\Downarrow$

```
ADD #1,a
```

Cost = 3

Best $\Downarrow$

```
INC a
```

Cost = 2

Instruction Selection: Utilizing Addressing Modes

- Suppose we translate **`a:=b+c`** into

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

- Assuming addresses of **`a`**, **`b`**, and **`c`** are stored in **`R0`**, **`R1`**, and **`R2`**

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

- Assuming **`R1`** and **`R2`** contain values of **`b`** and **`c`**

```
ADD R2,R1
MOV R1,a
```

Run Time Storage Management

- The information needed during an execution of a procedure is kept in a block of storage called activation record.
- Whenever a procedure is called a stack frame is allocated on the stack
- The information stored in the stack are return address, local variables, temporaries, return values etc.
- 2 standard storage allocation strategies namely
  - Static allocation
  - Stack allocation

- In static allocation the position of an activation record in memory is fixed at compile time.
- In stack allocation a new activation record is pushed onto the stack for each execution of the procedure. The record is popped when the activation ends.

- An **activation record** for a procedure has fields to hold parameters, results, machine-status information, local data, temporaries and the like. Since run-time allocation and de-allocation of activation records occurs as part of the procedure call and return sequences, we focus on the following three-address statements:

1. call
2. return
3. halt
4. action, a placeholder for other statements

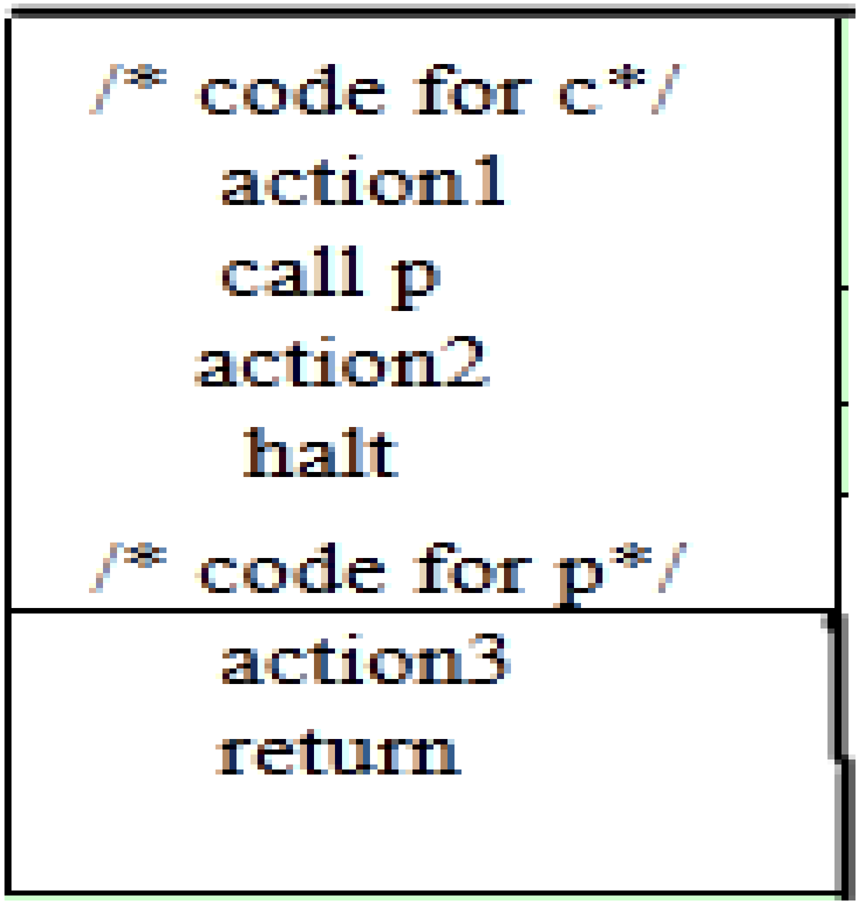
Static allocation

- A call statement, needs a move instruction to save the return address and GOTO transfers control to the target code for the called procedure.

```
MOV    #here +20, callee.static_area
GOTO   callee.code_area
```

  - MOV instruction saves return address.
  - GOTO transfers control to the target code for the called procedure.
  - Callee.static_area, calle.code_area are constants referring to the address of activation record and first instruction for the called procedure.
  - #here+20 in the mov instruction is return address.
  - Cost of 2 instructions is 5

Three address code



```
/* code for c*/
    action1
    call p
   action2
     halt
/* code for p*/
    action3
    return
```

Activation record for c(64 bytes)

0:	Return address
8:	arr
56:	I
60:	J

Activation record for p(88 bytes)

0:	Return address
4:	buf
84:	n

```
                                        /*code for c*/

100:   ACTION1
120:   MOV   #140,364                  /*save return address 140 */
132:    GOTO  200                      /* call p */
140:   ACTION2
160:    HALT
          ……
                                        /*code for p*/
200:    ACTION3
220:    GOTO  *364                      /*return to address saved in location 364*/
          ……
                                       /*300-363 hold activation record for c*/
300:                                   /*return address*/
304:                                   /*local data for c*/

          ……                           /*364-451 hold activation record for p*/
364:                                   /*return address*/
368:                                   /*local data for p*/
```

Target code

Stack allocation

- Static allocation can become stack allocation by using relative addresses for storage in activation records. The position of the record for an activation of a procedure is not known until run time.
- In stack allocation, this position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register
- When a procedure call occurs, the calling procedure increments SP and transfers control to the called procedure.
- After control returns to the caller, it decrements SP, thereby de-allocating the activation record of the called procedure.

- The code for the 1st procedure initializes the stack by setting SP to the start of the stack area in memory.

```
MOV   #stackstart, SP        /*initialize the stack*/
code for the first procedure
HALT                         /*terminate execution*/
```

- A procedure call sequence increments SP, saves the return address, and transfers control to the called procedure:

```
ADD   #caller.recordsize, SP
MOV   #here+16, SP              /* save return address*/
GOTO   callee.code_area
```

- The attribute *caller.recordsize* represents the size of an activation record, so the ADD instruction leaves SP pointing to the beginning of the next activation record.

- The source #*here*+16 in the MOV instruction is the address of the instruction following the GOTO; it is saved in the address pointed to by SP.

- The return sequence consists of two parts. The called procedure transfers control to the return address using

```
GOTO   *0(SP)          /*return to caller*/
```

  - The reason for using *0(SP) in the GOTO instruction is that we need two levels of indirection: 0(SP) is the address of the first word in the activation record and *0(SP) is the return address saved there.
  - The second part of the return sequence is in the caller, which decrements SP, thereby restoring SP to its previous value. That is, after the subtraction SP points to the beginning of the activation record of the caller:

    SUB #*caller.recordsize*, SP

The following program is a condensation of the three-address code for the Pascal program for reading and sorting integers. Procedure q is recursive, so more than one activation of q can be alive at the same time.

```
/*code for s*/
    action1
    call q
    action2
     halt
```

```
/*code for p*/
     action3
      return
```

```
/*code for q*/
      action4
       call p
       action5
       call q
       action6
       call q
       return
```

```
                          /*code for s*/
100:  MOV   #600, SP     /*initialize the stack*/
108:  ACTION1
128:  ADD  #ssize, SP     /*call sequence begins*/
136:  MOV  #152, *SP     /*push return address*/
144:  GOTO  300          /*call q*/
152:  SUB  #ssize, SP     /*restore SP*/
160:  ACTION2
180:  HALT
      ……….
```

```
                        /*code for p*/
200:  ACTION3
220:  GOTO  *0(SP)      /*return*/
      ..........
```

```
                              /*code for q*/
300:  ACTION4                 /*conditional jump to  456*/
320:  ADD  #qsize, SP
328:  MOV  #344, *SP           /*push return address*/
336:  GOTO  200               /*call p*/
344:  SUB  #qsize, SP
352:  ACTION5
372:  ADD  #qsize, SP
380:  MOV  #396, *SP           /*push return address*/
388:  GOTO  300                /*call q*/
396:  SUB  #qsize, SP
404:  ACTION6
424:  ADD  #qsize, SP
432:  MOV  #448, *SP            /*push return address*/
440:  GOTO  300                 /*call q*/
448:  SUB #qsize, SP
456:  GOTO  *0(SP)              /*return*/
          .........
 600:                         /*stack starts here*/
```

Register allocation and assignment

- Values in registers are easier and faster to access than memory
  - Reserve a few registers for stack pointers, base addresses etc
  - Efficiently utilize the rest of general-purpose registers
- Register allocation
  - At each program point, select a set of values to reside in registers
- Register assignment
  - Pick a specific register for each value, subject to hardware constraints
  - Register classes: not all registers are equal

- One approach to register allocation and assignment is to assign specific values in a object program to certain registers.
- Advantage of this method is that design of compiler will become easy.
- Disadvantage is that, registers will be used inefficiently, certain registers will go unused and unnecessary loads and stores are generated.

Global Register Allocation

- In code generation algo, registers were used to hold values of a single basic block.
- At the end of a block, store instructions were needed in order to store the live values in memory.
- *Live variables* : are the ones which are going to be used in subsequent blocks.

- So, LRA was slightly modified for the registers in order to hold the frequently used variables and keep these registers consistent across block boundaries(globally)
- Programs will spend most of their time in inner loops, a natural approach is that to keep a frequently used value in a fixed registers throughout the loop.
- But number of registers is fixed.

- Programs are written as if there are only two kinds of memory: main memory and disk
- Programmer is responsible for moving data from disk to memory (e.g., file I/O)
- Hardware is responsible for moving data between memory and caches
- Compiler is responsible for moving data between memory and registers

Usage counts

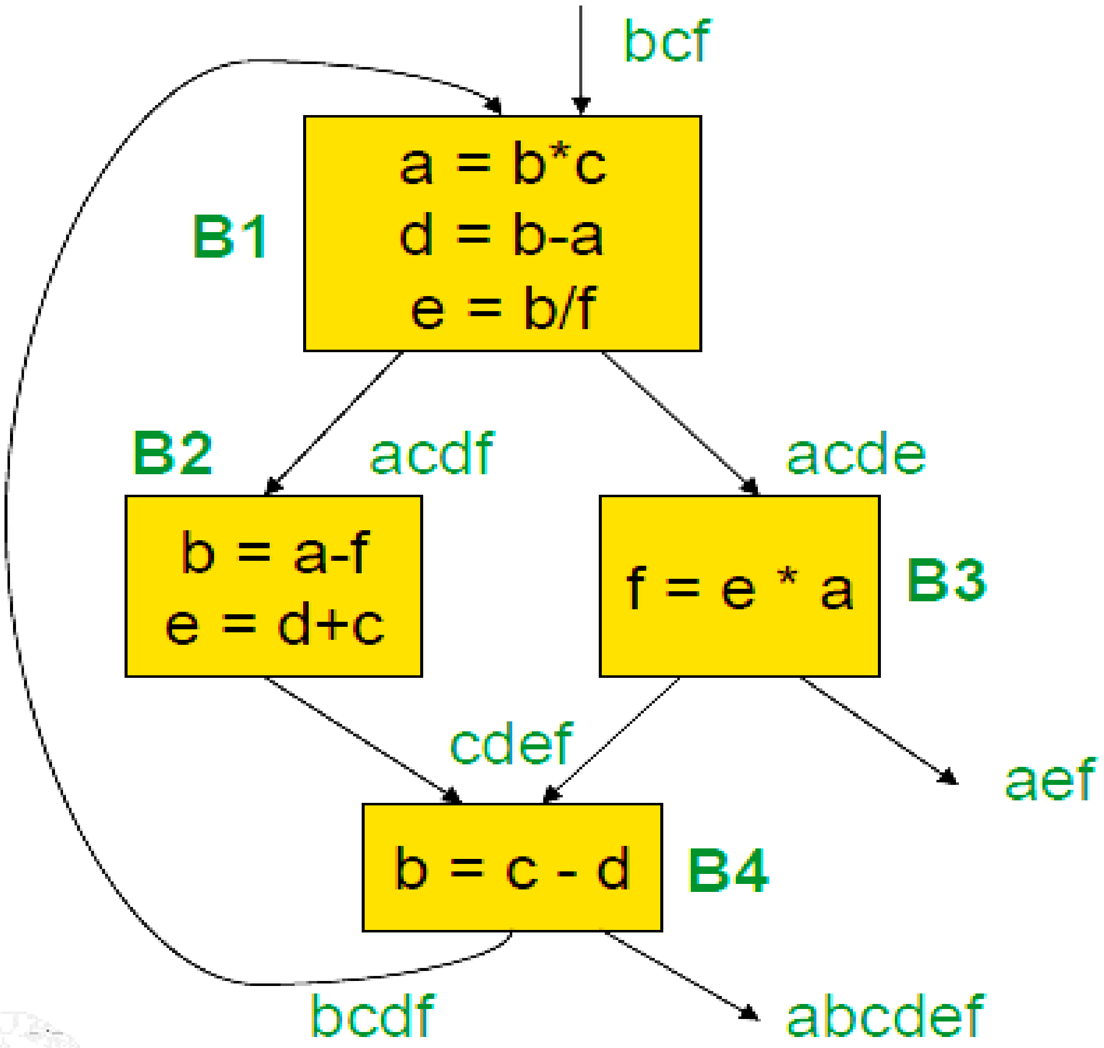
- If a variable say x, is in register then we can say that we have saved 1 cost.
- If a variable x, is defined somewhere outside the loop(a basic block), then for every usage of variable x in block we will save 1 cost.
- For every variable x computed in block, if it is live on exit from block, then count a saving of 2, since it is not necessary to store it at the end of the block.

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

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 is assigned a value in B., 0 otherwise.


bcf
a = b*c
d = b-a
e = b/f
B1
B2
acdf
acde
b = a-f
e = d+c
f = e * a
B3
cdef
aef
b = c - d
B4
bcdf
abcdef

B1 B2 B3 B4

a: (0+2)+(1+0)+(1+0)+(0+0) = 4

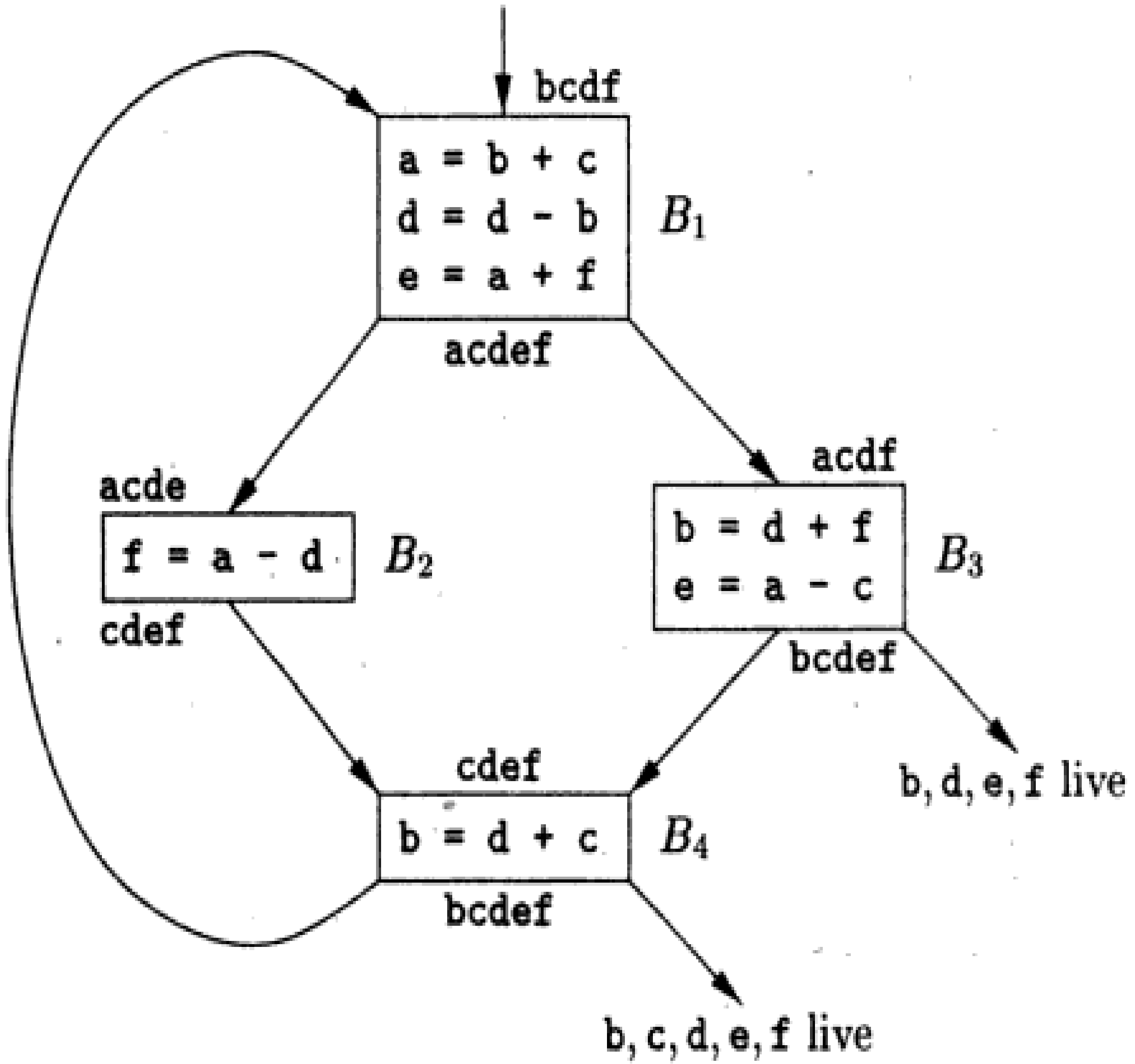
b: (3+0)+(0+0)+(0+0)+(0+2) = 5

c: (1+0)+(1+0)+(0+0)+(1+0) = 3

d: (0+2)+(1+0)+(0+0)+(1+0) = 4

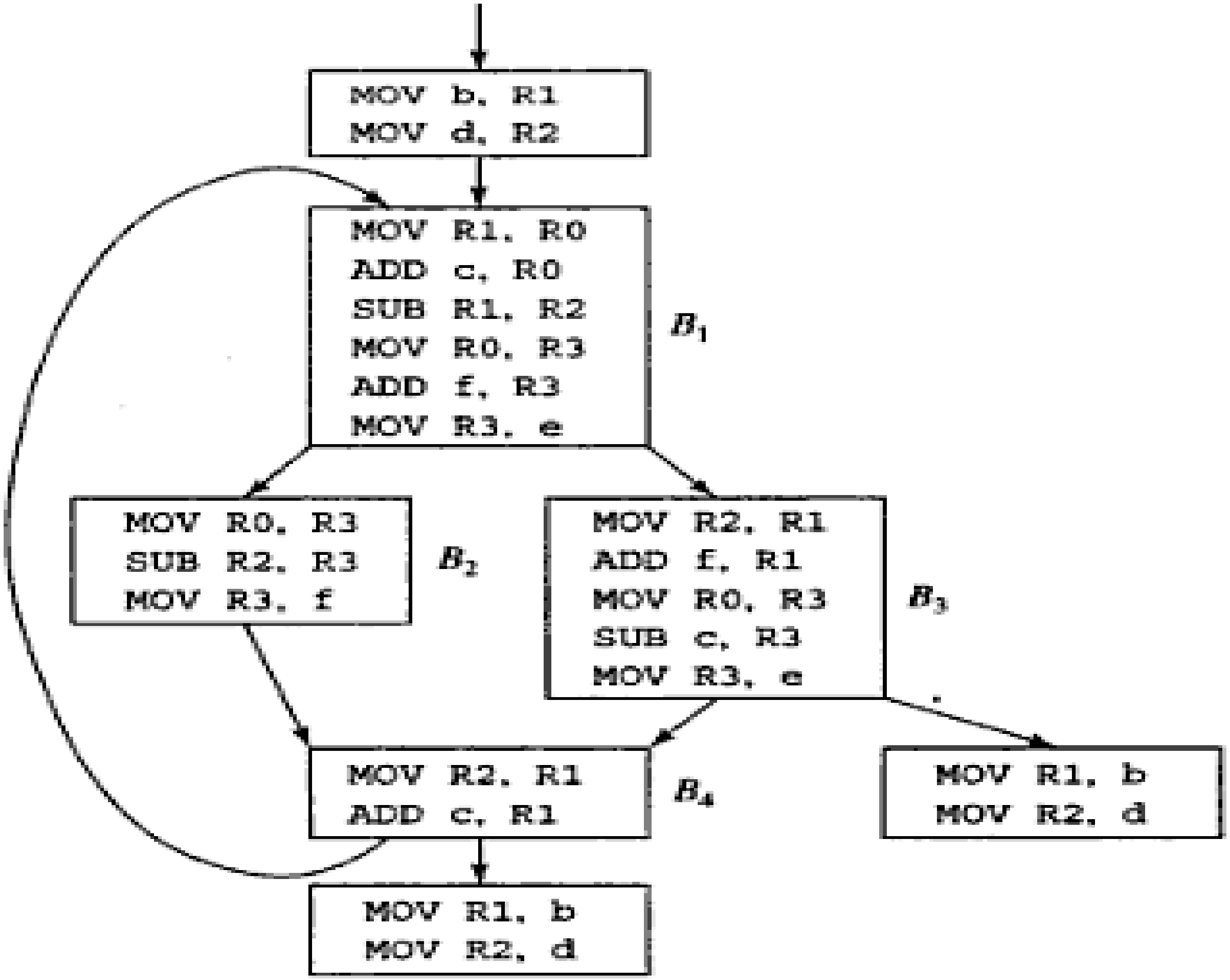
e: (0+2)+(0+2)+(1+0)+(0+0) = 5

f: (1+0)+(1+0)+(0+2)+(0+0) = 4



bcdf
a = b + c
d = d - b
e = a + f
B1
acdef
acde
f = a - d
B2
cdef
acdf
b = d + f
e = a - c
B3
bcdef
b, d, e, f live
cdef
b = d + c
B4
bcdef
b, c, d, e, f live

	B1	B2	B3	B4	Total count
A	(0+2)	(1+0)	(1+0)	(0+0)	4
B	(2+0)	(0+0)	(0+2)	(0+2)	6
C	(1+0)	(0+0)	(1+0)	(1+0)	3
D	(1+2)	(1+0)	(1+0)	(1+0)	6
E	(0+2)	(0+0)	(0+2)	(0+0)	4
F	(1+0)	(0+2)	(1+0)	(0+0)	4



Code sequence using global register assignment

Register assignment for outer loops

- If an outer loop L1 contains an inner loop L2, the names allocated registers in L2 need not be allocated registers in L1-L2.
- However if name x is allocated a register in loop L1 but not in L2, we must store x on entrance to L2 and load x on exit from L2.
- Similarly, if we choose to allocate x a register in L2 but not L1, we must load x on entrance to *L2* and store x on exit from L2.

Register allocation by graph coloring

- When a reg is needed for a computation but all available reg are in use, the contents of one of the used reg must be stored(spilled) into memory location.

- Perform dataflow liveness analysis over a CFG:
  - collect variable liveness info at every program point

- Build an interference graph:
  - each node represents a temporary value
  - each edge represents an interference between two temporaries—they can't be assigned to the same register

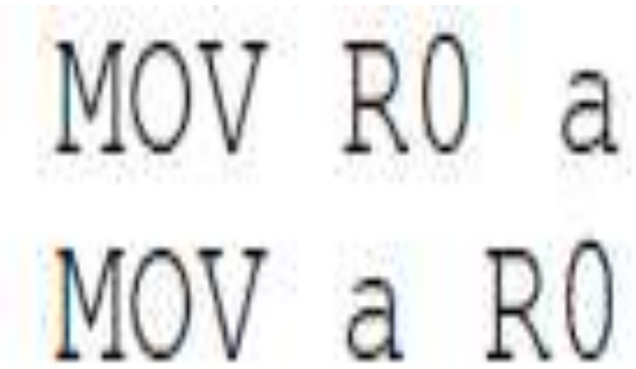
- Color the interference graph:
  - colors = registers; want to use as few colors as possible
  - for a machine with K registers, decide whether the interference graph is K-colorable
  - if yes, the coloring gives a register assignment
  - If no, need to spill some temporaries to memory

Peephole Optimization

- Statement by statement code generation strategy often produces target code that contains redundant instructions and suboptimal constructs.
- No guarantee that resulting code is optimal under any mathematical measure.
- ***Peephole optimization:*** a short sequence of target instructions(peephole instructions) that can be replaced by shorter or faster sequence instructions.

- Repeated passes over the target code are necessary to get the maximum benefit.
- Characteristics of peephole optimization
  - Redundant- instruction elimination
  - Flow of control optimizations
  - Algebraic simplifications
  - Use of machine idioms

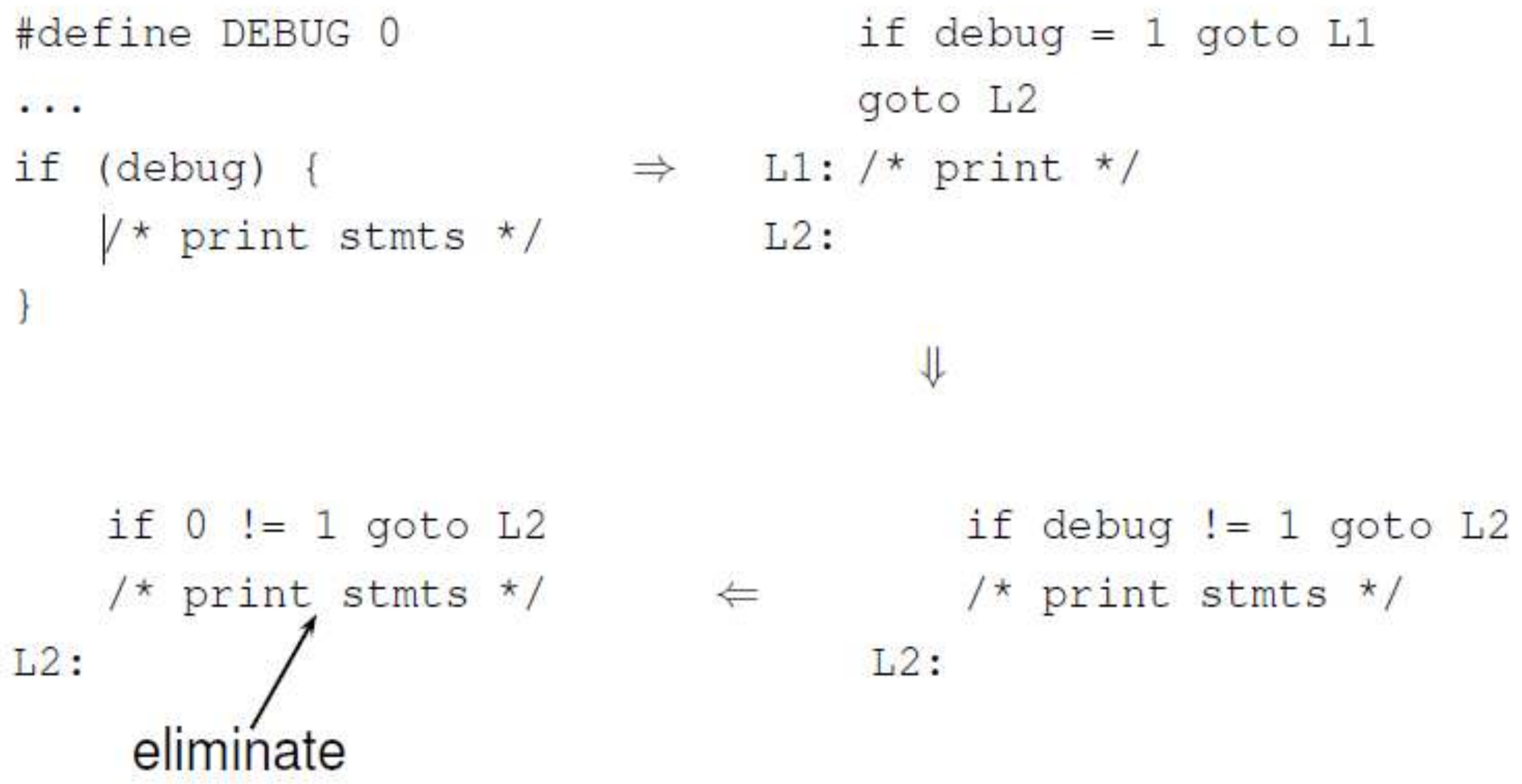
Redundant Loads and Stores



```
MOV R0 a
MOV a R0
```

Unreachable code

- Unlabeled instruction following an unconditional jump may be eliminated



```
#define DEBUG 0
...
if (debug) {
    /* print stmts */
}
```

⇒

```
     if debug = 1 goto L1
     goto L2
L1: /* print */
L2:
```

⇓

```
    if debug != 1 goto L2
    /* print stmts */
L2:
```

⇐

```
    if 0 != 1 goto L2
    /* print stmts */
L2:
```

Flow of control optimizations

```
        goto L1
        ...
L1:     goto L2
          ⇓
        goto L2
        ...
L1:     goto L2
```

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

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

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

Algebraic simplification

- Eliminate the instructions like the following
  - x = x+0
  - x = x*1

  These type of instructions are often produced by straightforward intermediate code generation algorithms, and they can be easily eliminated by peephole optimization.

Reduction in strength

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

ex,

$X^2 = X * X$

Use of machine idioms

- The target machine may have hardware instructions to implement certain specific operations efficiently.
- Detecting situations that permit the use of these instructions can reduce execution time significantly.
- Ex,
  - Increment and decrement addressing modes