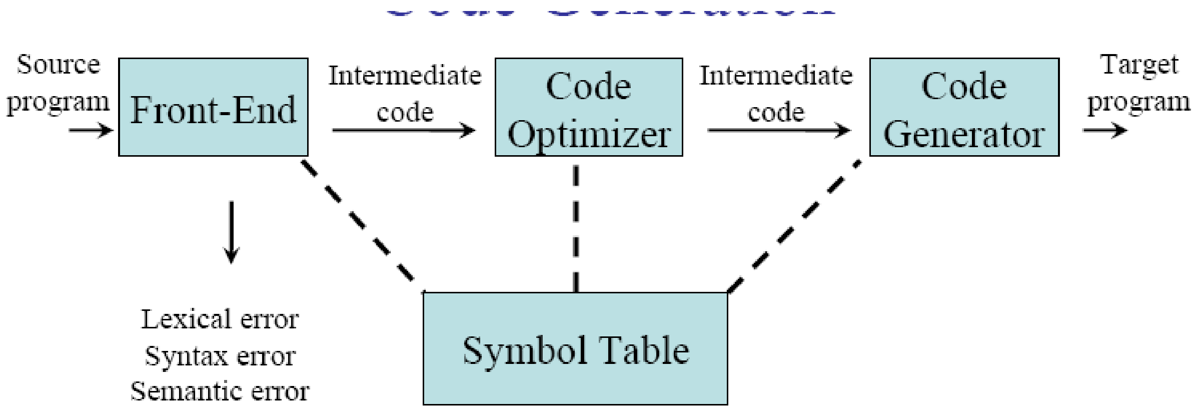
Code Generation & Optimization

How the target codes are generated optimally from an intermediate form of programming language.



Code produced by compiler must be correct and high quality. Source-to-target program transformation should be *semantics preserving* and effective use of target machine resources. Heuristic techniques should be used to generate good but suboptimal code, because generating optimal code is un-decidable.

Code Generator Design Issues

The details of code generation are dependent on the target language and operating system. Issues such as memory management, instruction selection, register allocation, evaluation order are in almost all code – generation problems.

The issues to code generator design includes:

Input to the code generator: The input to the code generator is intermediate representation together with the information in the symbol table. What type of input postfix, three-address, dag or tree.

Target Program: Which one is the out put of code generator: Absolute machine code (executable code), Re-locatable machine code (object files for linker), Assembly language (facilitates debugging), Byte code forms for interpreters (e.g. JVM)

Target Machine: Implementing code generation requires thorough understanding of the target machine architecture and its instruction set.

Instruction Selection: Efficient and low cost instruction selection is important to obtain efficient code.

Register Allocation: Proper utilization of registers improve code efficiency

Choice of Evaluation order: The order of computation effect the efficiency of target code.

The Target Machine

Consider a hypothetical target computer is a byte-addressable machine (word = 4 bytes) and n general propose registers, R0, R1, …, Rn-1. It has two address instruction of the form:

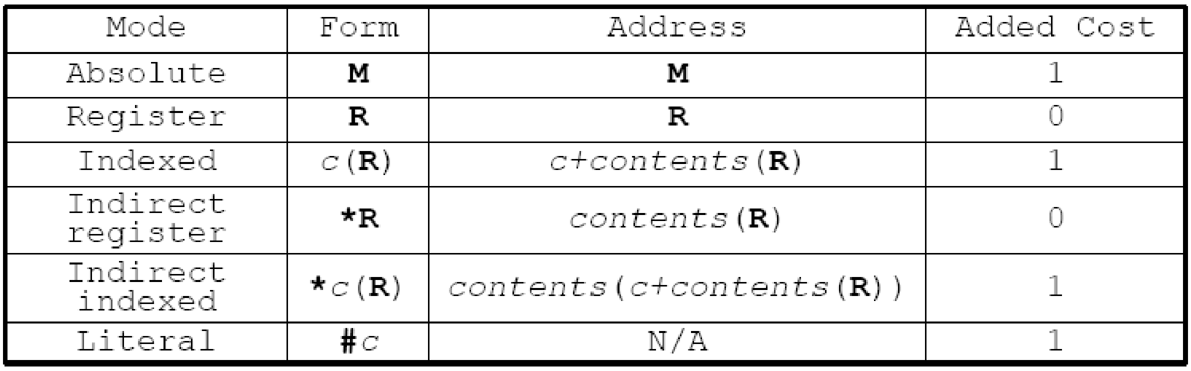
op source, destination

It has the following op-codes :
MOV (move content of source to destination),
ADD (add content of source to destination)

SUB (subtract content of source from destination .)
MUL (multiply content of source with destinanation)

The source and destination *of instructions are specified by combining register and memory location with address modes.* The address mode together with assembly forms and associated cost are:

Addressing modes:



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

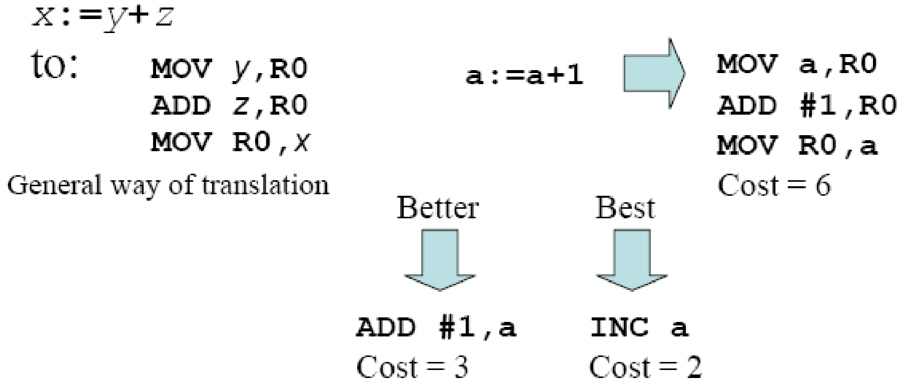
Instruction Costs

- Machine is a simple, non-super-scalar processor with fixed instruction costs
- Realistic machines have deep pipelines, I-cache, D-cache, etc.
- Define the cost of instruction
  = 1 + cost(*source*-mode) + cost(*destination*-mode)

Instruction	operation
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
```

General way of translation

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

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

Cost = 6

```
ADD #1,a
```

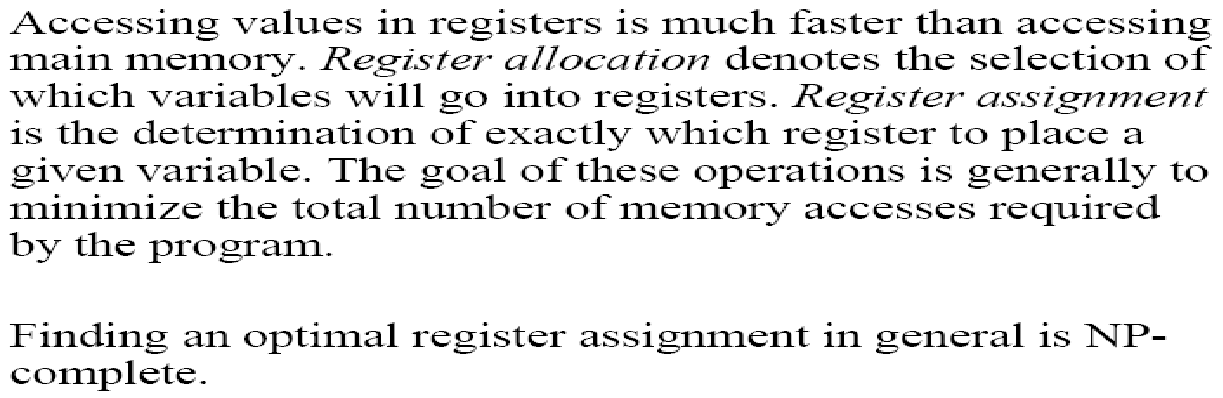
Cost = 3

```
INC a
```

Cost = 2

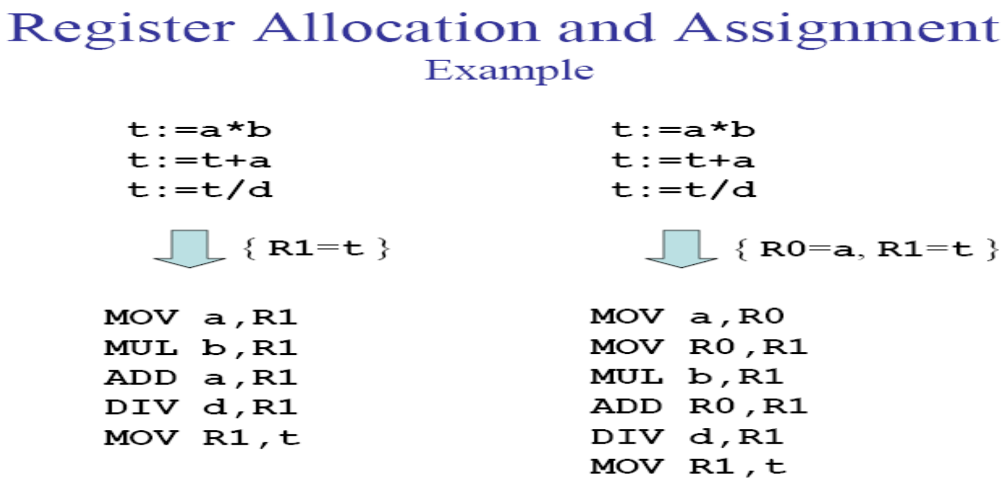
Picking the shortest sequence of instructions is often a good approximation of the optimal result

Register Allocation and Assignment



Accessing values in registers is much faster than accessing main memory. *Register allocation* denotes the selection of which variables will go into registers. *Register assignment* is the determination of exactly which register to place a given variable. The goal of these operations is generally to minimize the total number of memory accesses required by the program.

Finding an optimal register assignment in general is NP-complete.



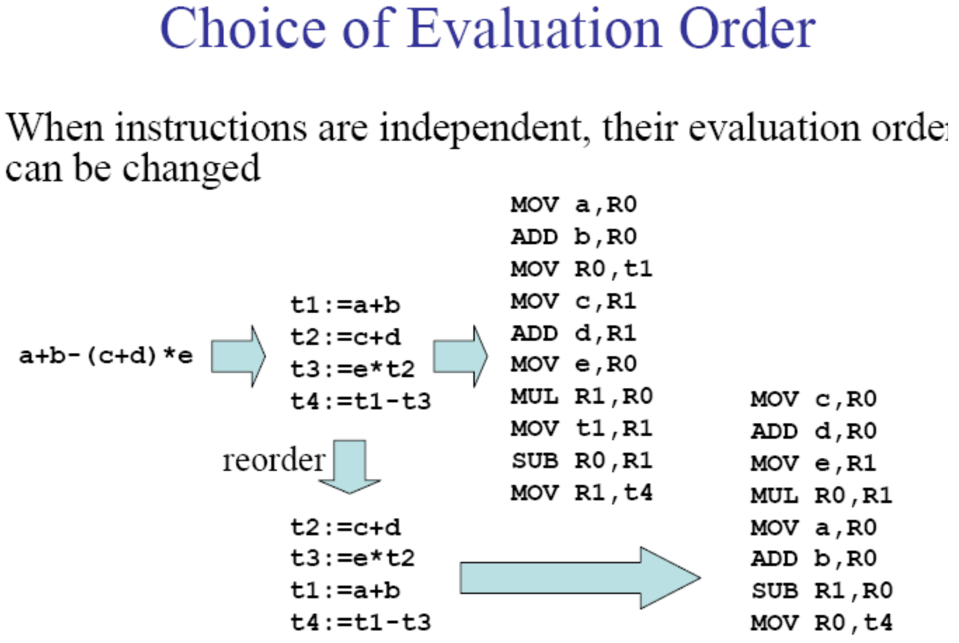
Register Allocation and Assignment

Example

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

  { R1=t }              { R0=a, R1=t }

MOV  a,R1             MOV  a,R0
MUL  b,R1             MOV  R0,R1
ADD  a,R1             MUL  b,R1
DIV  d,R1             ADD  R0,R1
MOV  R1,t             DIV  d,R1
                      MOV  R1,t
```



Choice of Evaluation Order

When instructions are independent, their evaluation orde: can be changed

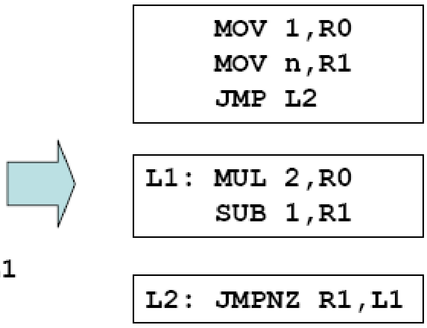
```
                                 MOV a,R0
                                 ADD b,R0
                                 MOV R0,t1
               t1:=a+b           MOV c,R1
a+b-(c+d)*e    t2:=c+d           ADD d,R1
               t3:=e*t2          MOV e,R0
               t4:=t1-t3         MUL R1,R0
                                 MOV t1,R1      MOV c,R0
         reorder                 SUB R0,R1      ADD d,R0
                                 MOV R1,t4      MOV e,R1
                                                MUL R0,R1
               t2:=c+d                          MOV a,R0
               t3:=e*t2                         ADD b,R0
               t1:=a+b                          SUB R1,R0
               t4:=t1-t3                        MOV R0,t4
```

Basic Blocks and Control Flow Graphs

Basic Blocks

A *basic block* is a sequence of consecutive instructions in which flow of control enters by one entry point and exit to another point without halt or branching except at the end.

```
    MOV 1,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```



```
    MOV 1,R0
    MOV n,R1
    JMP L2
```

```
L1: MUL 2,R0
    SUB 1,R1
```

```
L2: JMPNZ R1,L1
```

Basic Blocks and Control Flow Graphs

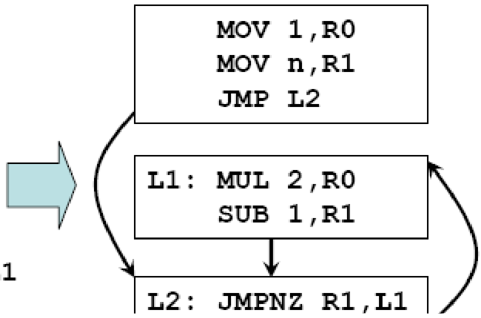
Flow Graphs

A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges.

A flow graph can be defined at the intermediate code level or target code level.

The nodes of flow graphs are the basic blocks and flow-of-control to immediately follow node connected by directed arrow.

```
    MOV 1,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```



```
    MOV 1,R0
    MOV n,R1
    JMP L2
```

```
L1: MUL 2,R0
    SUB 1,R1
```

```
L2: JMPNZ R1,L1
```

Basic Blocks Construction Algorithm

Input: A sequence of three-address statements

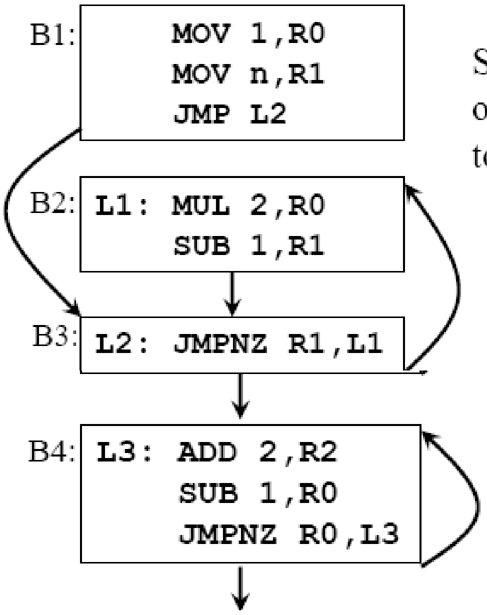
Output: A list of basic blocks with each three-address statement in exactly one block

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 conditional or goto is a leader
   - c. Any statement that immediately follows conditional or 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

Loops

A *loop* is a collection of basic blocks, such that

- All blocks in the collection are *strongly connected*
- The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry



```
B1:     MOV 1,R0
        MOV n,R1
        JMP L2
B2: L1: MUL 2,R0
        SUB 1,R1
B3: L2: JMPNZ R1,L1
B4: L3: ADD 2,R2
        SUB 1,R0
        JMPNZ R0,L3
```

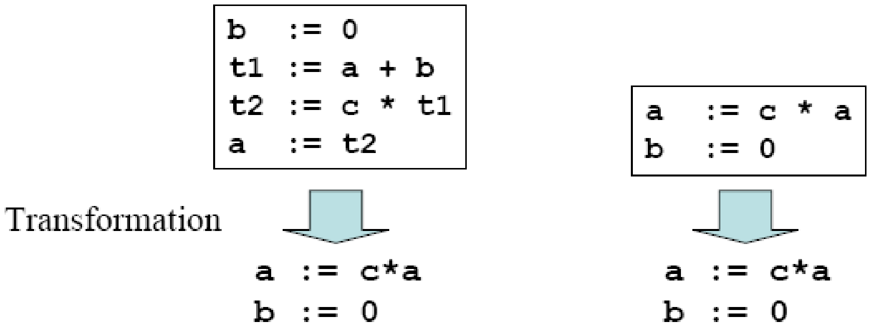
Strongly connected components: there is path of length of one or more from one node to another to make a cycle. Such as {B2,B3}, {B4}

Entries: B3, B4

A loop that consist no other loop is called inner loop

Equivalence of Basic Blocks

Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions



```
b   := 0
t1  := a + b
t2  := c * t1
a   := t2
```

```
a   := c * a
b   := 0
```

Transformation

```
a  := c*a
b  := 0
```

```
a  := c*a
b  := 0
```

Blocks are equivalent, assuming `t1` and `t2` are *dead*: no longer used (no longer *live*)

Transformations on Basic Blocks

A *code-improving transformation* is a ***code optimization*** to improve speed or reduce code size

Global transformations are performed across basic blocks

Local transformations are only performed on single basic blocks

Transformations must be safe and preserve the meaning of the code

A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form

Some local transformation are:

- Common-Subexpression Elimination
- Dead Code Elimination
- Renaming Temporary Variables
- Interchange of Statements
- Algebraic Transformations

Common-Subexpression Elimination

Remove redundant computations

Look at 2nd and 4th: compute same expression

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

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

Look at 1st and 3rd : b is redefine in 2nd therefore different in 3rd, not the same expression

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

```
t1 := b * c
t2 := a - t1
t4 := t2 + t1
```

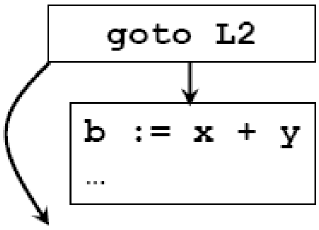
Dead Code Elimination

Remove unused statements

```
b := a + 1
a := b + c
…
```

```
b := a + 1
…
```

Assuming **a** is *dead* (not used)



```
goto L2
```

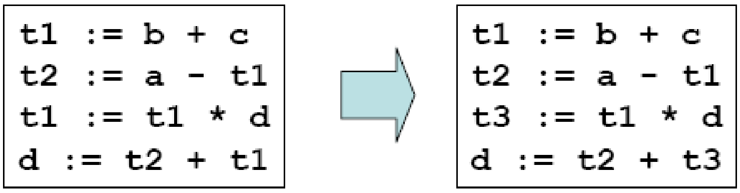
```
b := x + y
…
```

Remove unreachable code

Renaming Temporary Variables

Temporary variables that are dead at the end of a block can be safely renamed

The basic block is transforms into an equivalent block in which each statement that defines a temporary defines a new temporary. Such a basic block is called *normal-form block* or *simple block*.



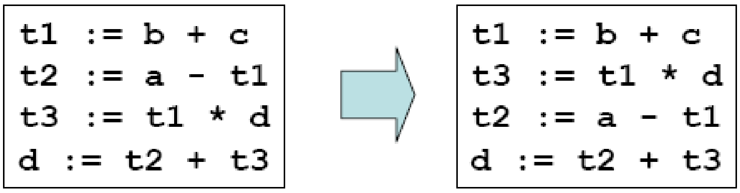
```
t1  := b + c
t2  := a - t1
t1  := t1 * d
d := t2 + t1
```

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

Normal-form block

Interchange of Statements

Independent statements can be reordered without effecting the value of block to make its optimal use.



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

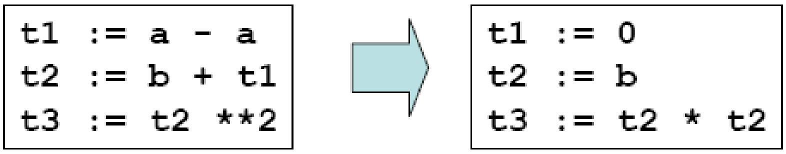
```
t1  := b + c
t3  := t1 * d
t2  := a - t1
d  := t2 + t3
```

Note that normal-form blocks permit all statement interchanges that are possible

Algebraic Transformations

Change arithmetic operations to transform blocks to algebraic equivalent forms

Simplify expression or replace expensive expressions by cheaper ones.



```
t1  := a - a
t2  := b + t1
t3  := t2 **2
```

```
t1  := 0
t2  := b
t3  := t2 * t2
```

In statement 3, usually require a function call

Transforms to simple and equivalent statement

Next-Use Information

Next-use information is needed for dead-code elimination and register assignment (if the name in a register is no longer needed, then the register can be assigned to some other name)

If *i*: $x = \dots$ and *j*: $y = x + z$ are two statements *i* & *j*, then *next-use* of *x* at *i* is *j*.

Next-use is computed by a backward scan of a basic block and performing the following actions on statement

```
            i: x := y op z
```

- Add liveness/next-use info on *x*, *y*, and *z* to statement *i* (whatever in the symbol table)
- Before going up to the previous statement (scan up):
  - Set *x* info to "not live" and "no next use"
  - Set *y* and *z* info to "live" and the next uses of *y* and *z* to *i*

All nontemporary variables and temporary that is used across the block are considered live.

Computing Next-Use

Example

Step 1

i: `a := b + c`

j: `t := a + b` [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Attach current live/next-use information
Because info is empty, assume variables are live
(Data flow analysis Ch.10 can provide accurate information)

Step 2

i: `a := b + c`

live(**a**) = true	*nextuse*(**a**) = *j*
live(**b**) = true	*nextuse*(**b**) = *j*
live(**t**) = false	*nextuse*(**t**) = none

j: `t := a + b` [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Compute live/next-use information at *j*

Computing Next-Use

Step 3 *i*: `a := b + c` [*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false,
nextuse(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**c**) = none]

j: `t := a + b` [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Attach current live/next-use information to *i*

live(**a**) = false	*nextuse*(**a**) = none
live(**b**) = true	*nextuse*(**b**) = *i*
live(**c**) = true	*nextuse*(**c**) = *i*
live(**t**) = false	*nextuse*(**t**) = none

Step 4

i: `a := b + c` [*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false,
nextuse(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**c**) = none]

j: `t := a + b` [*live*(**a**) = false, *live*(**b**) = false, *live*(**t**) = false,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Compute live/next-use information *i*

Code Generator

Generates target code for a sequence of three-address statements using next-use information

Uses new function *getreg* to assign registers to variables

Computed results are kept in registers as long as possible, which means:

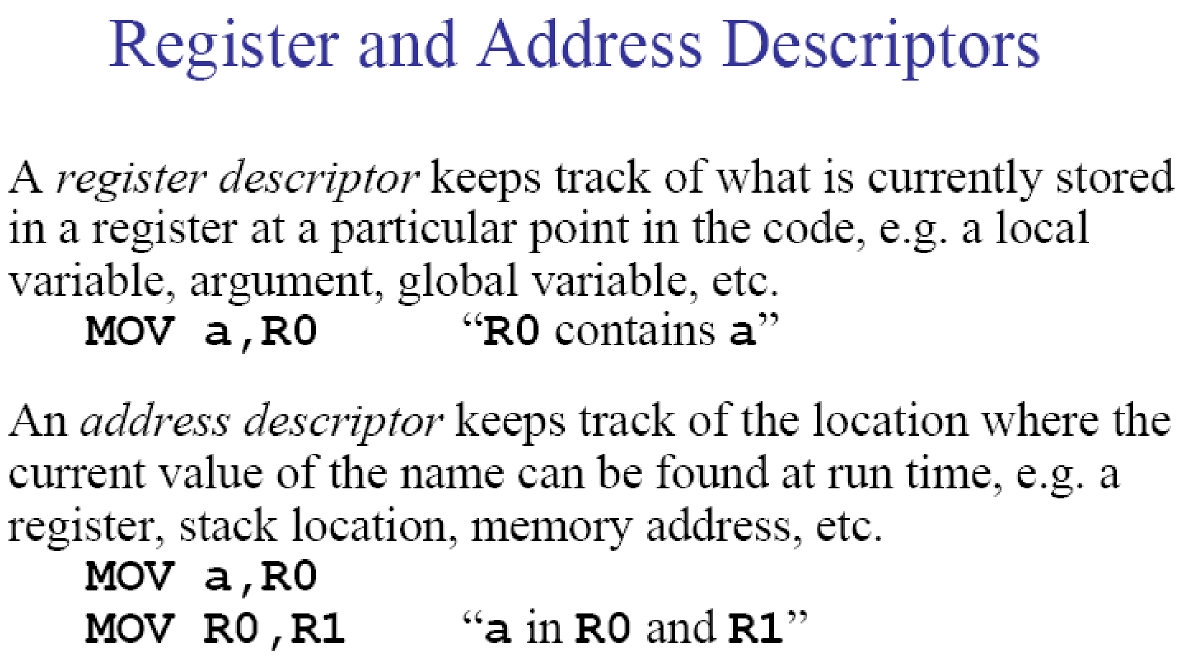
- Result is needed in another computation
- Register is kept up to a procedure call or end of block

Checks if operands to three-address code are available in registers

Code Generation Algorithm

For each statement *x* := *y* op *z*

1. Set location *L* = *getreg*(*y*, *z*)   // to store the result of *y op z*
2. If $y \notin L$ then generate   //L is address descriptor --wait!
   MOV *y'*,*L*   //to place copy of y in L
   where *y'* denotes one of the locations where the value of *y* is available (choose register if possible)
3. Generate instruction
   OP *z'*,*L*
   where *z'* is one of the locations of *z*;
   Update register/address descriptor of *x* to include *L*
4. If *y* and/or *z* has no next use and is stored in register, update register descriptors to remove *y* and/or *z*



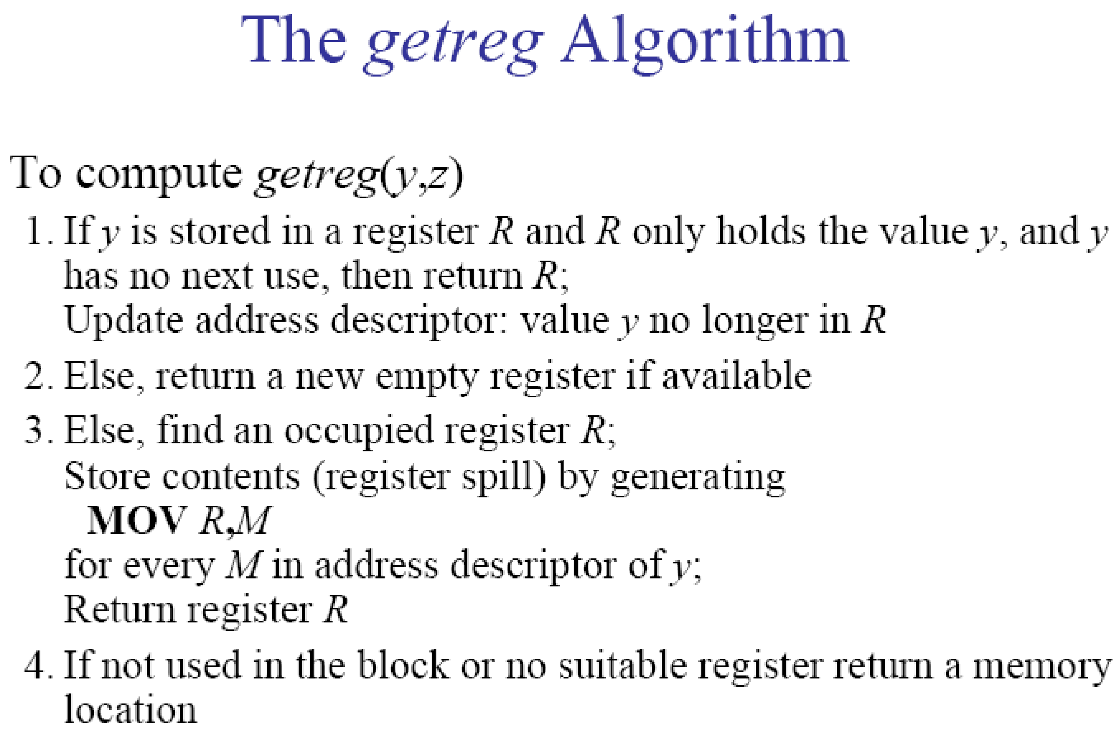
Register and Address Descriptors

A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

```
    MOV a,R0          “R0 contains a”
```

An *address descriptor* keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

```
    MOV a,R0
    MOV R0,R1         “a in R0 and R1”
```



The *getreg* Algorithm

To compute *getreg*(*y*,*z*)

1. If *y* is stored in a register *R* and *R* only holds the value *y*, and *y* has no next use, then return *R*;
   Update address descriptor: value *y* no longer in *R*
2. Else, return a new empty register if available
3. Else, find an occupied register *R*;
   Store contents (register spill) by generating
   MOV *R*,*M*
   for every *M* in address descriptor of *y*;
   Return register *R*
4. If not used in the block or no suitable register return a memory location

Code Generation

Example

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

Statements	Code Generated	Register Descriptor	Address Descriptor
t := a - b	MOV a,R0 SUB b,R0	Registers empty 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

Peephole Optimization

Statement-by-statement code generation often produce redundant instructions that can be optimize to save time and space requirement of target program.

Examines a short sequence of target instructions in a window (*peephole*) and replaces the instructions by a faster and/or shorter sequence whenever possible.

Applied to intermediate code or target code

Typical optimizations:

- Redundant instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Eliminating Redundant Loads and Stores

Consider

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

This type code is not generated by our algorithm of page 25

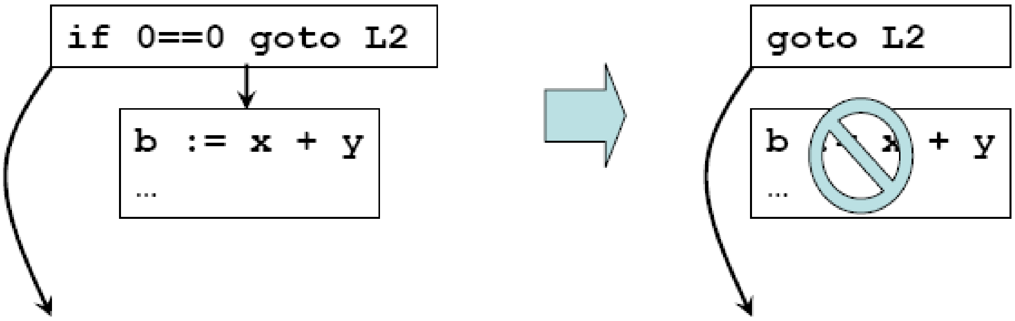
The second instruction can be deleted because first ensures value of a in R0, but only if it is not labeled with a target label

- Peephole represents sequence of instructions with at most one entry point

The first instruction can also be deleted if *live*(**a**) = false

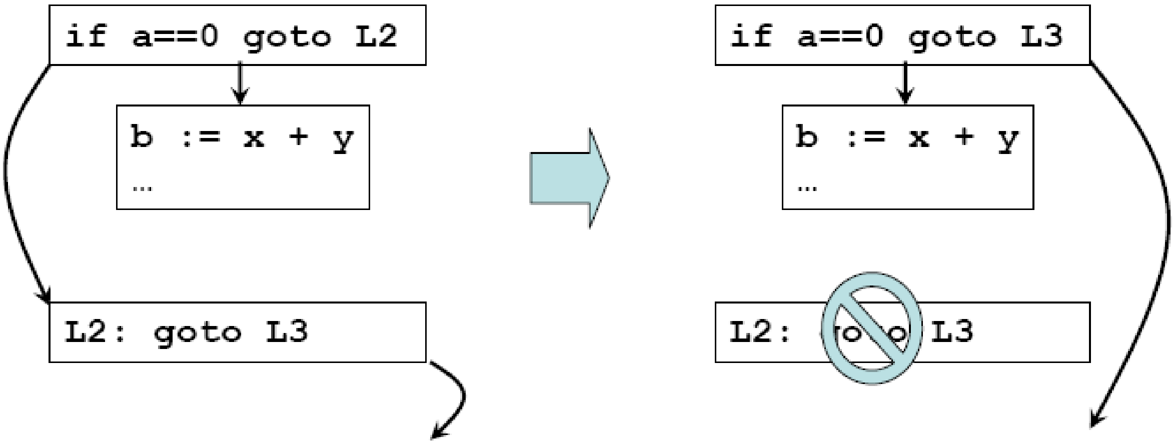
Deleting Unreachable Code

An unlabeled instruction immediately following an unconditional jump can be removed



Branch Chaining

Shorten chain of branches by modifying target labels



```
if a==0 goto L2

  b := x + y
  …

L2: goto L3
```

```
if a==0 goto L3

  b := x + y
  …

L2: goto L3
```

Remove redundant jumps as well

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

```
if a < b goto L2
goto L3
……
```

Other Peephole Optimizations

Reduction in strength: replace expensive arithmetic operations with cheaper ones

```
…
a := x ^ 2
b := y / 8
```

```
…
a := x * x
b := y >> 3
```

Utilize machine idioms (use addressing mode inc)

```
…
a := a + 1
```

```
…
inc a
```

Algebraic simplifications

```
…
a := a + 0
b := b * 1
```

```
…
```

Run Time Storage management

A compiler contains a block of storage from the operating system for the compiled program to run in. This run time storage might be sub-divided to hold

1. The generated target code
2. data objects and
3. a counterpart of the control stack to keep track of procedure activation.

- The size of generated target code is fixed at compile time so it can be placed in a statically determined area – low end of memory.
- Some of data objects may also be known at compile time so these too can be placed in to statically determined area.
- The addresses of these data objects can be compiled into target code
- For the activation of procedure, when a call occurs, execution of an activation is interrupted and information about the status of the machine such as value of program counter, machine register is saved into stack until the control returns from call to the activation.
- Data objects whose life times are contained in that of an activation can be allocated on the stack along with other information associated with the activation.
- Separate area of run time storage , called heap, holds other information.

The management of run time storage by sub-division is:

Code
Static Data
Stack
Heap

- The size of stack and heap may change during execution.
- By convention , stack grows down and heap grows up

Information needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record:

An activation record is a collection of fields, starting from the field for temporaries as

Returned value	←value returned after execution
actual parameter	← used by the calling procedure to call procedure.
optional control link	← points to the activation record of the caller.
optional access link	← Non local data held in other activation record.
saved machine state	← State of the machine just before procedure call
local data	← Data that are local to an execution.
temporaries	← Temporary values used for evaluation of expression

Since , run time allocation and de-allocation of activation records occurs as part of procedure call-return sequences, following three address statements are in focus.

1. call
2. return
3. halt
4. action – a place holder for other statements

Now , consider the following input to the code generator.

Three address code

```
/*Code for procedure
C */
action 1
call p
action 2
halt
```

```
/*code for procefure
p  */
action 3
return
```

Activation record for C 64 bytes

0:	return address
	Array
56:	i
60:	j

Activation record for p(88 b)

0:	return address
4:	Buffer
84:	n

Using the static allocation,

A call statement in the intermediate code is implemented by two target machine instruction MOV and GOTO

- The code constructed from procedure C and p above using arbitrary address 100 and 200 as:

Assume action takes cost of 20 bytes. – MOV and GOTO + 3 constants cost = 20 bytes
The target code for the input above will be as:

```
100: ACTION1
120: MOV #140,364   /* saves return address 140 */
132: ACTION2
160: HLT
……
/*Code for P */
200: ACTION3
220: GOTO *364    /* returns to address saved in location 364 */
…….
    /* 300-363 hold activation record for c */
300: /* return address */
304: /*local data for c */
……
/* 364-451 holds activation record for P */
364: /*return address */
368: /* local data for p */
```

- The MOV instruction at address 120 saves the return address 140 in machine status field - the first word in activation record of p.
- The GOTO instruction at 132 transfers control to first instruction to the target code of called procedure.
- *364 represents 140 when GOTO statement at address 220 is executed, control then returns to 140.