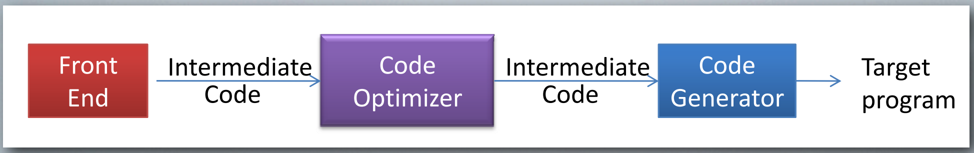
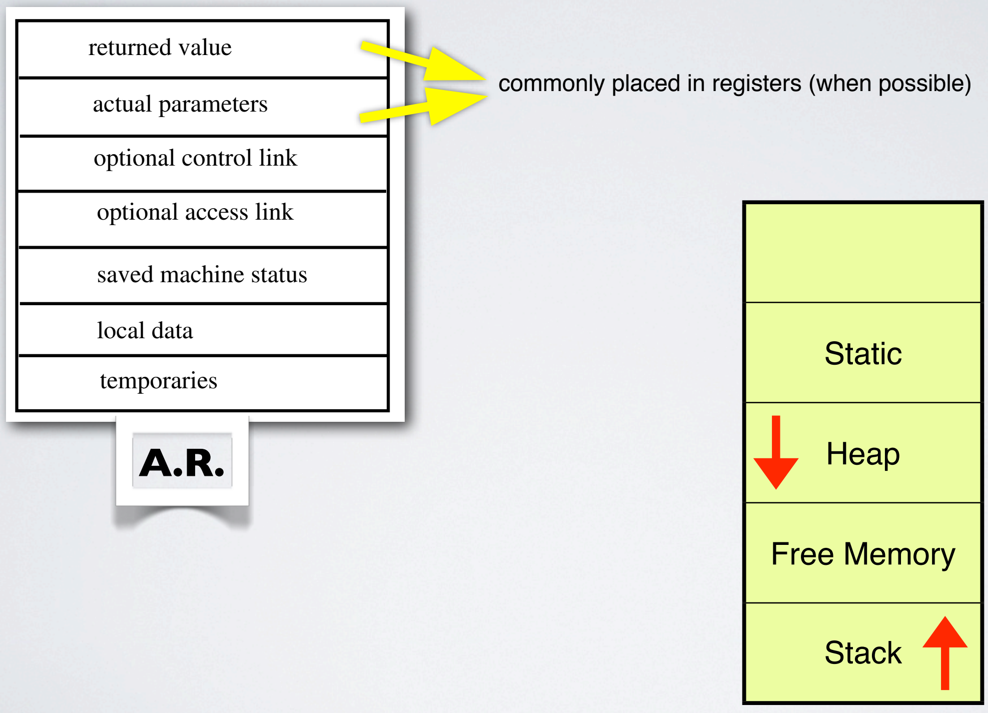
Code Generation



Memory Management



For some compiler, the intermediate code is a pseudo-code of a virtual machine.

- Interpreter of the virtual machine is invoked to execute the intermediate.
- No machine-dependent generation is needed.
- Usually with great overhead.

Examples:

◁ Pascal: P-code for the virtual P machine.
◁ JAVA: Bytecode for the virtual JAVA machine.

Machine-dependent issues

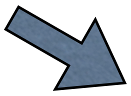
- Input and output formats:
  - The formats of the intermediate and the target program.
- Memory management:
  - Alignment, indirect addressing, paging, segment, . . .
  - Those you learned from your assembly language class.
- Instruction cost:
  - Special machine instructions to speed up execution.
  - Example:
    - Increment by 1.
    - Multiplying or dividing by 2.
    - Bit-wise manipulation.
    - Operators applied on a continuous block of memory space.
  - Pick a fastest instruction combination for a certain target machine.

Machine-dependent issues

Register allocation: in-between machine dependent and independent issues.

- C language allows the user to management a pool of registers.
- Some language leaves the task to compiler.
- Idea: save mostly used intermediate result in a register. However, finding an optimal solution for using a limited set of registers is NP-hard.

```
t=a+b
```



```
load R0,a
load R1,b
add  R0,R1
store R0,T
load R0,a
add  R0,b
store R0,T
```

Machine-independent issues

Techniques

- Analysis of dependence graphs.
- Analysis of basic blocks and flow graphs.
- Semantics-preserving transformations.
- Algebraic transformations.

The Target Program

• RISC (reduced instruction set computer)

– Many registers, three address instructions, simple addressing modes, simple instruction set arch

• CISC (complex instruction set computer) – few registers, two address instructions, various addressing modes, variable length instruction set

• Stack based Machine

– Pushing operands onto a stack

– Almost disappeared, then revived with Java Virtual Machine (JVM)

Machine-dependent issues

Choice of the target language: RISC

```
LD R0, y          //R0 = y
ADD R0, R1, R2    //R0 = R0=R1+R2
ST x, R0          // x = R0
```

A Simple Target Machine Model

Byte-addressable machine with *n* general-purpose registers, R_0, R_1, ... ,R_{n-1}

OPERATIONS	FORMAT
Load	`LD r,x` `(r=x)`
Store	`ST x,r` `(x=r)`
Computation	`OP r1, r2,r3` `(SUB r1, r2,r3 // r1=r2-r3)`
Unconditional jumps	`BR L`
Conditional jump	`Bcond r, L` `(BLTZ r, L)`

Addressing modes

format	addr	examples
`x`	Lval(x)	name
`a(r)`	Lval(a)+ Rval(r)	LD R1 a(R2)
`const(r)`	const+Rval(r)	LD R1, 100(R2)
`*r`	Rval(Rval(r))	LD R1, *(R2)
`*const(r)`	Rval(const+Rval(r))	LD R1, *100(R2)
`#const // immediate op`	nil	LD R1, #100

`x = y-z`

```
LD R1, y
LD R2, z
SUB R1,R1,R2
ST x, R1
```

`b = a[i]`

```
LD R1, i
MUL R1,R1,8
LD R2,a(R1)        //R2 ← Rval(Lval(a)+Rval(R1))
ST b, R2
```

```
a[j] = c
```

```
LD R1, c
LD R2, j
MUL R2, R2, 8
ST a(R2), R1
```

```
x = *i
```

```
LD R1, i
LD R2,0(R1)
ST b, R2
```

```
*p = y
```

```
LD R1, p
LD R2, y
ST 0(R1),R2
```

```
if x < y goto L
```

```
LD R1, x
LD R2, y
SUB R1, R1, R2
BLTZ R1, M
```

M is the label that represents the first machine instruction generated from the three-address instruction that has label L

Register allocation: in-between machine dependent and independent issues.

- C language allows the user to management a pool of registers.
- Some language leaves the task to compiler.
- Idea: save mostly used intermediate result in a register.

Finding an optimal solution for using a limited set of registers is NP-hard.

```
t=a+b
         load R0,a
         load R1,b
         add R0,R1
         store R0,T
```

Basic blocks

Maximal sequences of consecutive three-address instructions s.t.:

- The flow of control can only enter the basic block through the first instruction in the block (no jumps into the middle of the block)
- Control will leave the block without halting or branching, except possibly at the last instruction in the block.

- Partition the intermediate code into *basic blocks*
- The basic blocks become the nodes of a *flow graph,* whose edges indicate which blocks can follow which other blocks.

Partitioning three-address instructions into basic blocks.

Algorithm 8.5:

INPUT:

A sequence of three-address instructions.

OUTPUT:

A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD:

a) determine the *leaders:*

1. The first three-address instruction in the intermediate is a leader.

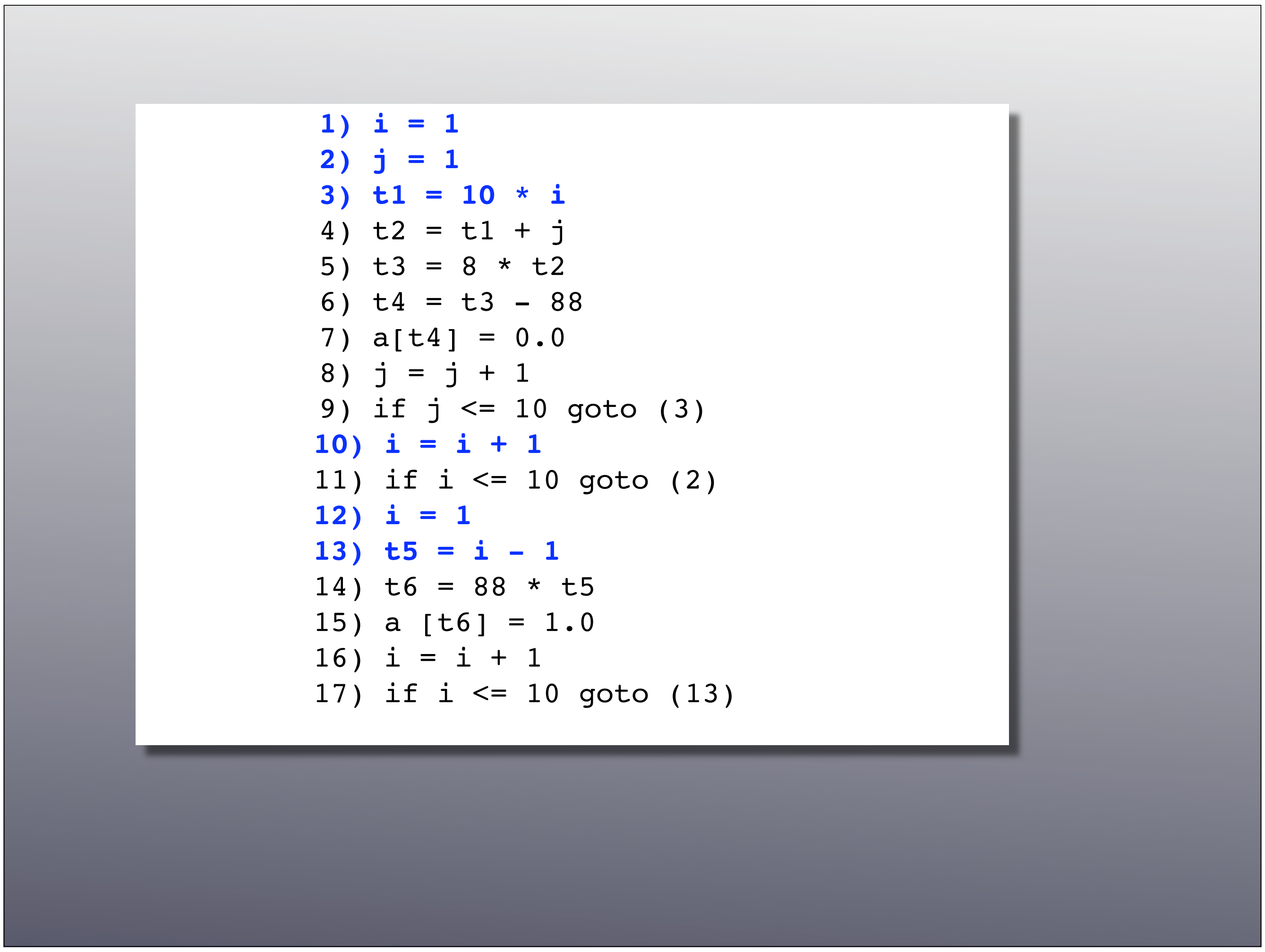
2. Any instruction that is the target of a conditional or unconditional jump is a leader.

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

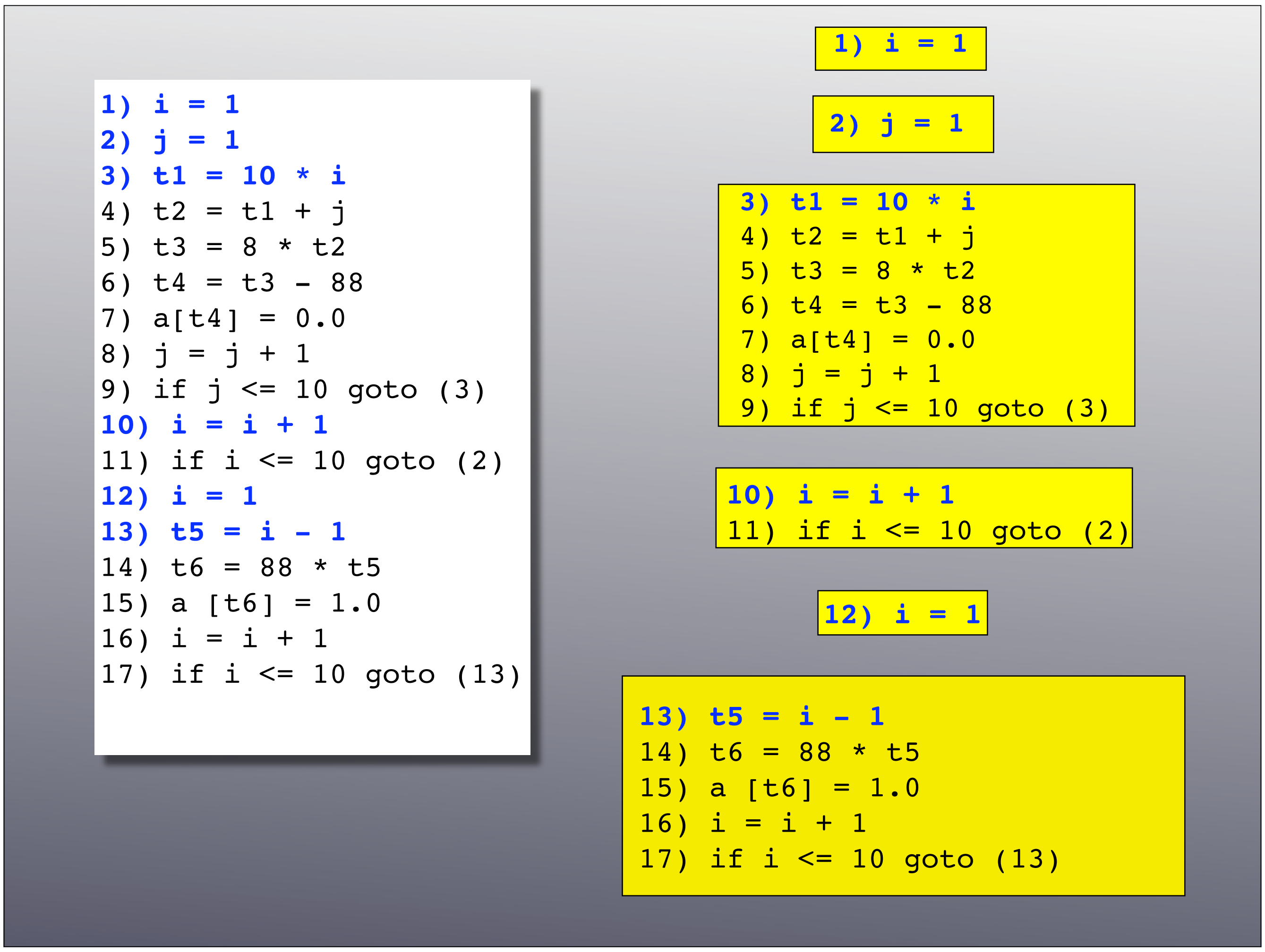
b) for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

```
 1) i = 1
 2) j = 1
 3) t1 = 10 * i
 4) t2 = t1 + j
 5) t3 = 8 * t2
 6) t4 = t3 - 88
 7) a[t4] = 0.0
 8) j = j + 1
 9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a [t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Leaders ?



```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a [t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```



```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a [t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

```
1) i = 1
```

```
2) j = 1
```

```
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
```

```
10) i = i + 1
11) if i <= 10 goto (2)
```

```
12) i = 1
```

```
13) t5 = i - 1
14) t6 = 88 * t5
15) a [t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a [t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

B1
```
i = 1
```

B2
```
j = 1
```

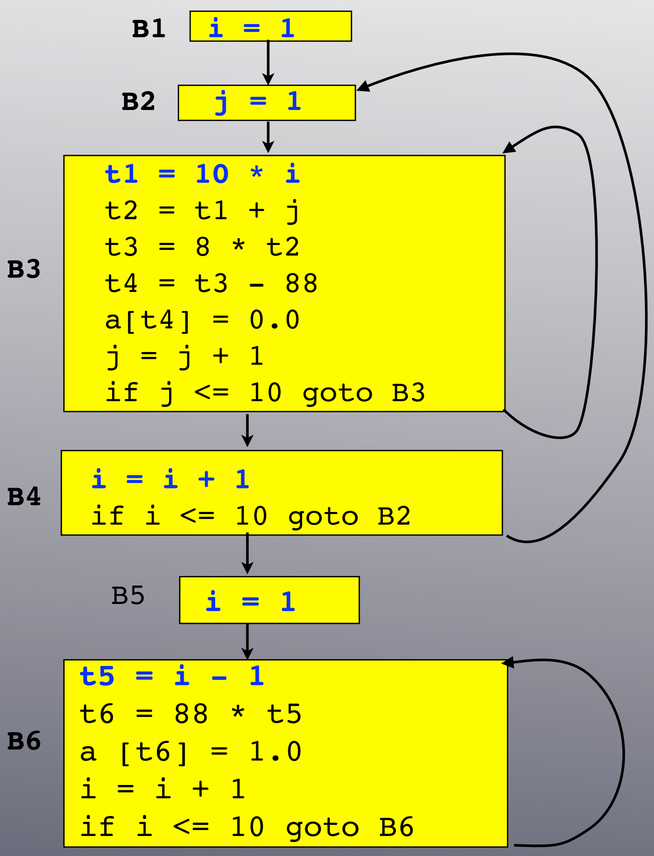
B3
```
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
a[t4] = 0.0
j = j + 1
if j <= 10 goto B3
```

B4
```
i = i + 1
if i <= 10 goto B2
```

B5
```
i = 1
```

B6
```
t5 = i - 1
t6 = 88 * t5
a [t6] = 1.0
i = i + 1
if i <= 10 goto B6
```

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a [t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```



Optimization of Basic Blocks

- Local optimization within each basic block
- Global optimization

We will focus on the former.

DAG Representation of Basic Blocks

Target: Construct a DAG for a basic block

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.

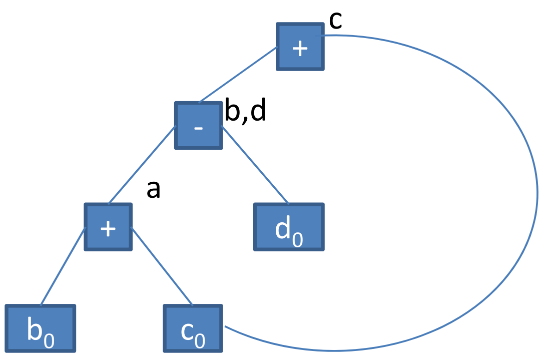
2. $\forall$statement **s** we associate a node N_s.

The **children** of N_s are those nodes corresponding to statements that are the last definitions, prior to ***s***, of the operands used by ***s***.

3. each node N_s is labeled by the operator applied at **s**.
   Attached to N_s is the list of variables for which it is the last definition within the block.

4. Certain nodes are designated *output nodes*. These are the nodes whose variables are live on exit from the block; that is, their values may be used later, in another block of the flow graph.

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



When we construct the node for the third statement **c=b+c**, we know that the use of **b** in **b+c** refers to the node labeled **-**, because that is the most recent definition of **b**. Thus, we do not confuse the values computed at statements one and three.

However, the node corresponding to the fourth statement **d=a-d** has the operator **-** and the nodes with attached variables **a** and **d_0** as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add **d** to the list of definitions for the node labeled

The DAG representation of a basic block lets us perform

- local common sub-expressions elimination
- dead code elimination
- reordering of statements that do not depend on one another
- reordering of operands of three-address instructions by applying algebraic laws

A Simple Code Generator

• Generate code for a single basic block

• How to use registers?

– In most machine architectures, some or all of the operands must be in registers

– Registers make good temporaries

– Hold values that are computed in one basic block and used in other blocks

– Often used with run time storage management

These are competing needs, since the number of registers available is limited.

```
• LD reg, mem

• ST mem, reg

• OP reg, reg, reg
```

Register and Address Descriptors

• For each available register, a register descriptor (RD) keeps track of the vars whose current value in that register

– Initially empty

• For each var, an address descriptor (AD) keeps track of the locations where the current value of the var can be found

– Location can be a register, a memory address, etc.

Generation Algorithm

• **For a three address instruction**, e.g. x=y+z, do:
– Use **getReg(x=y+z)** to select registers R_x, R_y, R_z for x, y , z
– If y is not in R_y, issue an instruction **LD R_y,y'** (y' is a location for y)
• Similarly for z
– Issue the instruction **ADD R_x,R_y,R_z**
• **Copy statement** x=y
We assume that **getReg()** will always choose the same register for both x and y
– If y is not already in register, generate **LD R_y,y'** (y' is a location for y)
– Adjust RD for R_y so it includes x
– change the AD for x so that its only location is R_y
• **Ending the basic block**
– for each variable x whose location descriptor does not say that its value is located in the memory location for x and if x is used at other blocks, issue
ST x,R (R is a register in which x's value exists at the end of the block)

Managing Register and Address Descriptors (RD and AD)

• For LD R, x
– Change RD for R so it holds only x
– Change AD for x by adding R as an additional location
• For ST x, R
– Change AD for x to include its own memory location
• For ADD Rx, Ry, Rz
– Change RD for Rx so it holds only x
– Change AD for x so its only location is Rx
– Remove Rx from the AD of any var other than x
• For x= y, after generating the load for y into register Ry, if needed, and after managing descriptors as for all load statements:
– Add x to the register descriptor for Ry.
– Change the address descriptor for x so that its only location is Ry.

Example

```
t = a – b
u = a – c
v = t + u
a = d
d = v + u
```

- t, u, v are temp vars, while a, b, c, d are global
- Assume registers are enough
  - Reuse registers whenever possible

- **For a three address instruction**, e.g. `x=y+z`, do:
  - Use *getReg(x=y+z)* to select registers R_x, R_y, R_z for x, y , z
  - If y is not in R_y, issue an instruction **LD R_y,y'** (y' is a location for y)
- Similarly for z
  - Issue the instruction **ADD R_x,R_y,R_z**
- **Copy statement** x=y

We assume that getReg() will always choose the same register for both x an y
- If y is not already in register, generate **LD R_y,y'** (y' is a location for y)
- Adjust RD for R_y so it includes x
- change the AD for x so that it only location is R_y

- **Ending the basic block**
  - for each variable x whose location descriptor does not say that its value is located in the memory location for x and if x is used at other blocks, issue **ST x, R** (R is a register in which x's value exists at the end of the block)

- For LD R, x
  - Change RD for R so it holds only x
  - Change AD for x by adding R as an additional location
- For ST x, R
  - Change AD for x to include its own memory location
- For ADD Rx, Ry, Rz
  - Change RD for Rx so it holds only x
  - Change AD for x so its only location is Rx
  - Remove Rx from the AD of any var other than x
- For x= y, after generating the load for y into register Ry, if needed, and after managing descriptors as for all load statements:
  - Add x to the register descriptor for Ry.
  - Change the address descriptor for x so that its only location is Ry.

R1	R2	R3	a	b	c	d	t	u	v
			a	b	c	d			

t = a - b LD R1, a; LD R2, b; SUB R2, R1, R2

R1	R2	R3	a	b	c	d	t	u	v
a	t		a,R1	b	c	d	R2		

u = a - c LD R3, c; SUB R1, R1, R3

R1	R2	R3	a	b	c	d	t	u	v
u	t	c	a	b	c,R3	d	R2	R1	

- **For a three address instruction**, e.g. `x=y+z`, do:

– Use *getReg(x=y+z)* to select registers R_x, R_y, R_z for x, y , z

– If y is not in R_y, issue an instruction **LD R_y,y'** (y' is a location for y)

- Similarly for z

– Issue the instruction **ADD R_x,R_y,R_z**

- **Copy statement** x=y

We assume that getReg() will always choose the same register for both x an y

– If y is not already in register, generate **LD R_y,y'** (y' is a location for y)

– Adjust RD for R_y so it includes x

–change the AD for x so that it only location is R_y

- **Ending the basic block**

– for each variable x whose location descriptor does not say that its value is located in the memory location for x and if x is used at other blocks, issue

ST x, R (R is a register in which x's value exists at the end of the block)

- For LD R, x

– Change RD for R so it holds only x

– Change AD for x by adding R as an additional location

- For ST x, R

– Change AD for x to include its own memory location

- For ADD Rx, Ry, Rz

– Change RD for Rx so it holds only x

– Change AD for x so its only location is Rx

– Remove Rx from the AD of any var other than x

- For x= y, after generating the load for y into register Ry, if needed, and after managing descriptors as for all load statements:

– Add x to the register descriptor for Ry.

– Change the address descriptor for x so that its only location is Ry.

v = t + u

ADD R3, R2, R1

R1	R2	R3
u	t	v

a	b	c	d	t	u	v
a	b	c	d	R2	R1	R3

a = d

LD R2, d

R1	R2	R3
u	a,d	v

a	b	c	d	t	u	v
R2	b	c	d,R2		R1	R3

- **For a three address instruction**, e.g. `x=y+z`, do:

– Use *getReg(x=y+z)* to select registers R_x, R_y, R_z for x, y , z

– If y is not in R_y, issue an instruction **LD R_y,y'** (y' is a location for y)

- Similarly for z

– Issue the instruction **ADD R_x,R_y,R_z**

- **Copy statement** x=y

We assume that getReg() will always choose the same register for both x an y

– If y is not already in register, generate **LD R_y,y'** (y' is a location for y)

– Adjust RD for R_y so it includes x

- **Ending the basic block**

– for each variable x whose location descriptor does not say that its value is located in the memory location for x and if x is used at other blocks, issue

ST x, R (R is a register in which x's value exists at the end of the block)

- For LD R, x

– Change RD for R so it holds only x

– Change AD for x by adding R as an additional location

- For ST x, R

– Change AD for x to include its own memory location

- For ADD Rx, Ry, Rz

– Change RD for Rx so it holds only x

– Change AD for x so its only location is Rx

– Remove Rx from the AD of any var other than x

- For x= y, after generating the load for y into register Ry, if needed, and after managing descriptors as for all load statements:

– Add x to the register descriptor for Ry.

– Change the address descriptor for x so that its only location is Ry.

d = v + u

ADD R1, R3, R1

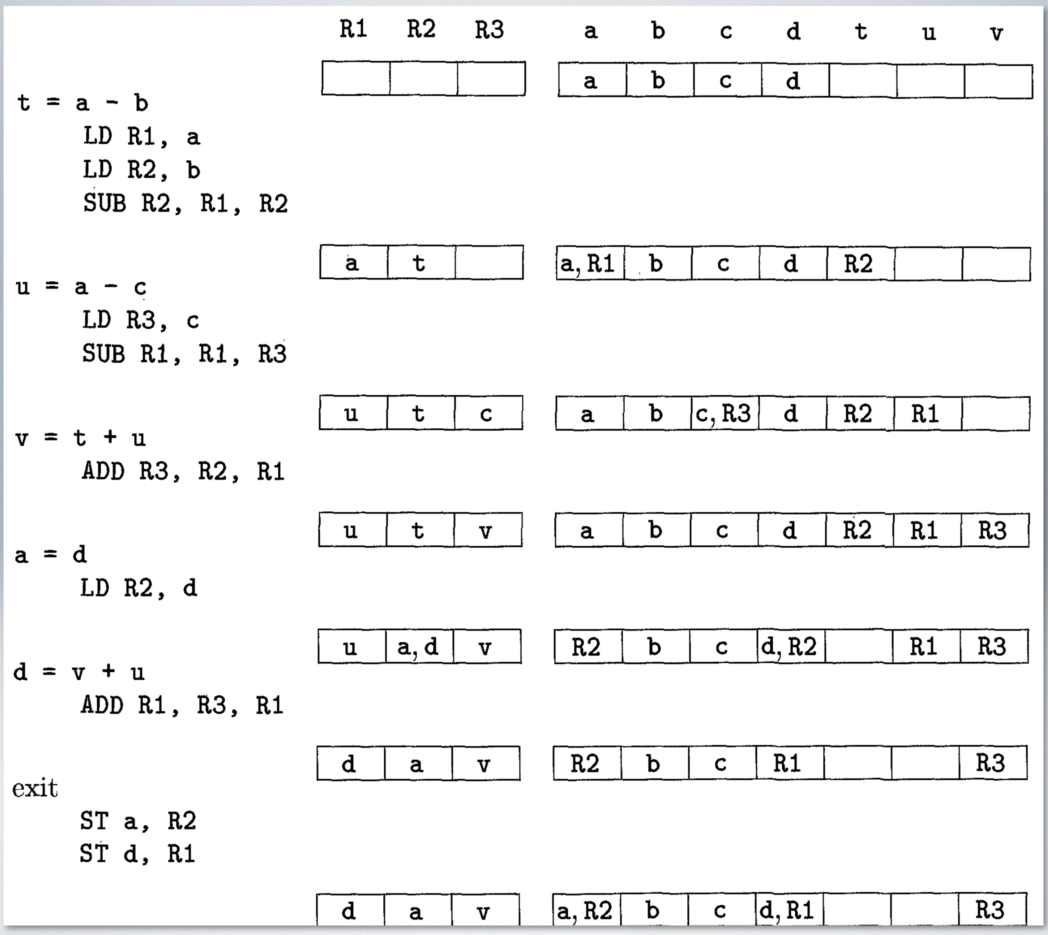
R1	R2	R3
d	a	v

a	b	c	d	t	u	v
R2	b	c	R1			R3

ST a, R2; ST d, R1

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
a,R2	b	c	d,R1			R3



	R1	R2	R3	a	b	c	d	t	u	v
				a	b	c	d			
t = a - b LD R1, a LD R2, b SUB R2, R1, R2										
	a	t		a, R1	b	c	d	R2		
u = a - c LD R3, c SUB R1, R1, R3										
	u	t	c	a	b	c, R3	d	R2	R1	
v = t + u ADD R3, R2, R1										
	u	t	v	a	b	c	d	R2	R1	R3
a = d LD R2, d										
	u	a, d	v	R2	b	c	d, R2		R1	R3
d = v + u ADD R1, R3, R1										
	d	a	v	R2	b	c	R1			R3
exit ST a, R2 ST d, R1										
	d	a	v	a, R2	b	c	d, R1			R3

getReg(I): selects registers for each memory location associated with the three-address instruction I

Function getReg()

- Consider picking Ry for y in x = y + z

– If y is in a register, do nothing

– If y is not in a register and there is a empty one, choose it as Ry

PROBLEM: y is not in a register, and there are no empty registers we proceed in the following way:

Let v be one of the var in R

- We are OK if v is somewhere else besides R
- We are OK if v is x
- We are OK if v is not used later
- if we are not OK perform **Spill, namely** ST v, R

Selection of the register *Rx.* The issues and options are almost as for *y,* so we shall only mention the differences.

- Since a new value of *x* is being computed, a register that holds only *x* is always an acceptable choice for *Rx.* This statement holds even if *x* is one of *y* and *z,* since our machine instructions allows two registers to be the same in one instruction.
- If *y* is not used after instruction *I* and *Ry* holds only *y* after being loaded, if necessary, **and, if y is not a temporary, AD(y) contains y,** then *Ry* can also be used as *Rx.* A similar option holds regarding *z* and *Rz.*

- Case when *I* is a copy instruction *x* = *y.*

We pick the register *Ry* as above. Then, we always choose *Rx* = *Ry.*