



NEW YORK UNIVERSITY

G22.2130-001
Compiler Construction
Lecture 13:
Code Generation II

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu



Copyright © Randy Glasbergen. www.glasbergen.com

Main Tasks of Code Generator

- **Instruction selection:** choosing appropriate target-machine instructions to implement the IR statements
- **Registers allocation and assignment:** deciding what values to keep in which registers
- **Instruction ordering:** deciding in what order to schedule the execution of instructions

Principal Uses of Registers

- In many machines, operands of an instruction must be in registers.
- Registers make good temporaries.
- Registers are used to hold global values, generated in basic block and used in another.
- Registers help with run-time storage management.

Simple Code Generator

- For basic blocks
- Assume we have one choice of machine instructions
- Quick summary
 - Consider each three-address instruction in turn
 - Decide what loads are necessary to get needed operands in registers
 - Generate the loads
 - *LD reg, mem*
 - Generate the instruction itself
 - *ST mem, reg*
 - Generate store if needed
 - *OP reg, reg, reg*

Simple Code Generator

- We need a data structure that tells us:
 - What program variables have their value in registers, and which registers if so.
 - Whether the memory location associated with the variable has the latest value.
- So we need two structures
 - For each available register: a **register descriptor** keeps track of variable names whose current value is in that register
 - For each program variable: an **address descriptor** keeps track of the location(s) where the current value can be found

Simple Code Generator

- Assume there are enough registers
- `getReg(I)` function
 - input: three-address code instruction `I`
 - Output: Selects register for each memory location associated with `I`
 - Has access to all register and variable descriptors

Simple Code Generator

For a three-address instruction such as $x = y + z$, do the following:

1. Use $getReg(x = y + z)$ to select registers for x , y , and z . Call these R_x , R_y , and R_z .
2. If y is not in R_y (according to the register descriptor for R_y), then issue an instruction $LD R_y, y'$, where y' is one of the memory locations for y (according to the address descriptor for y).
3. Similarly, if z is not in R_z , issue an instruction $LD R_z, z'$, where z' is a location for z .
4. Issue the instruction $ADD R_x, R_y, R_z$.

SPECIAL CASE: For copy instructions in the form of $x = y$ we assume $getReg$ will always choose the **same register for x and y**

Ending the basic block: For each variable whose memory location is not up to date generate $ST x, R$ (R is the register where x exists at end of the block)

Managing Register and Address Descriptors

For the instruction `LD R, x`

- (a) Change the register descriptor for register R so it holds only x .
- (b) Change the address descriptor for x by adding register R as an additional location.

For the instruction `ST x, R`, change the address descriptor for x to include its own memory location.

For an operation such as `ADD Rx, Ry, Rz` implementing a three-address instruction $x = y + z$

- (a) Change the register descriptor for R_x so that it holds only x .
- (b) Change the address descriptor for x so that its only location is R_x . Note that the memory location for x is *not* now in the address descriptor for x .
- (c) Remove R_x from the address descriptor of any variable other than x .

Managing Register and Address Descriptors

When we process a copy statement $x = y$, after generating the load for y into register R_y , if needed, and after managing descriptors as for all load statements

- (a) Add x to the register descriptor for R_y .
- (b) Change the address descriptor for x so that its only location is R_y .

Example

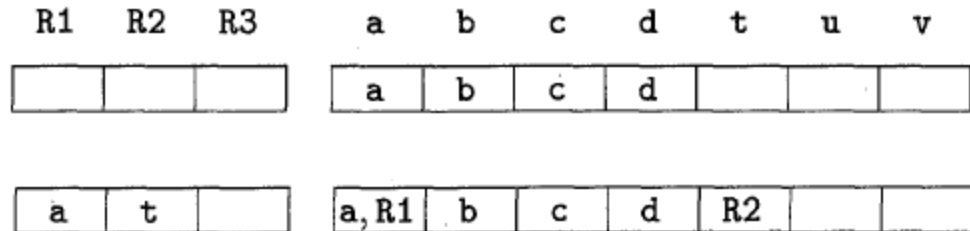
$t = a - b$

$u = a - c$

$v = t + u$

$a = d$

$d = v + u$



```

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

For the instruction LD R, x

- Change the register descriptor for register R so it holds only x .
- Change the address descriptor for x by adding register R as an additional location.

For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction $x = y + z$

- Change the register descriptor for R_x so that it holds only x .
- Change the address descriptor for x so that its only location is R_x . Note that the memory location for x is *not* now in the address descriptor for x .
- Remove R_x from the address descriptor of any variable other than x .

Example

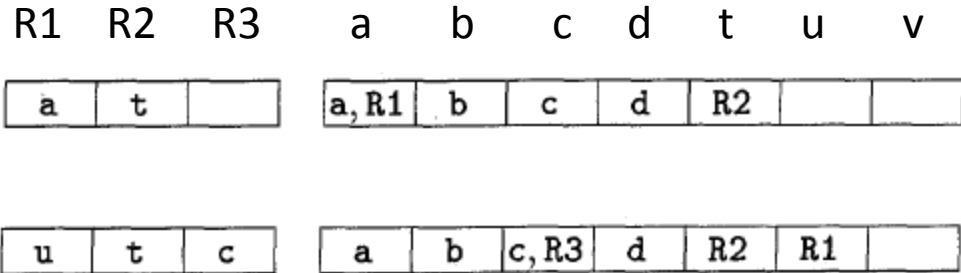
$t = a - b$

$u = a - c$

$v = t + u$

$a = d$

$d = v + u$



$u = a - c$

LD R3, c

SUB R1, R1, R3

For the instruction LD R, x

- Change the register descriptor for register R so it holds only x .
- Change the address descriptor for x by adding register R as an additional location.

For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction $x = y + z$

- Change the register descriptor for R_x so that it holds only x .
- Change the address descriptor for x so that its only location is R_x . Note that the memory location for x is *not* now in the address descriptor for x .
- Remove R_x from the address descriptor of any variable other than x .

Example

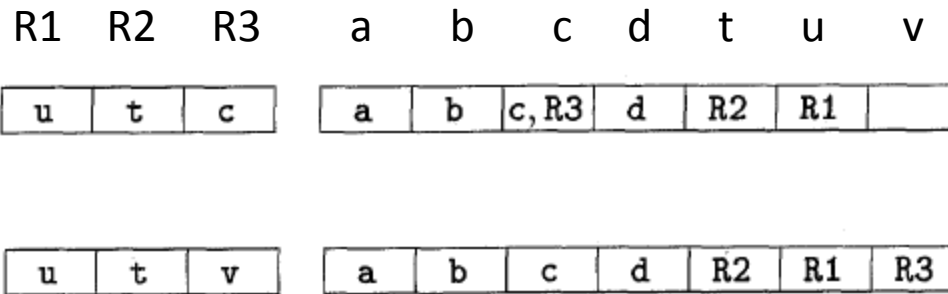
$t = a - b$

$u = a - c$

$v = t + u$

$a = d$

$d = v + u$



$v = t + u$

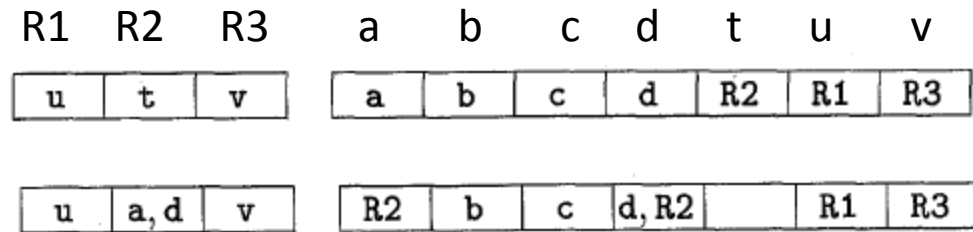
ADD R3, R2, R1

For an operation such as $\text{ADD } R_x, R_y, R_z$ implementing a three-address instruction $x = y + z$

- Change the register descriptor for R_x so that it holds only x .
- Change the address descriptor for x so that its only location is R_x .
Note that the memory location for x is *not* now in the address descriptor for x .
- Remove R_x from the address descriptor of any variable other than x .

Example

$t = a - b$
 $u = a - c$
 $v = t + u$
 $a = d$
 $d = v + u$



$a = d$
 LD R2, d

For the instruction LD R, x

- Change the register descriptor for register R so it holds only x .
- Change the address descriptor for x by adding register R as an additional location.

When we process a copy statement $x = y$, after generating the load for y into register R_y , if needed, and after managing descriptors as for all load statements (per rule 1):

- Add x to the register descriptor for R_y .
- Change the address descriptor for x so that its only location is R_y .

Example

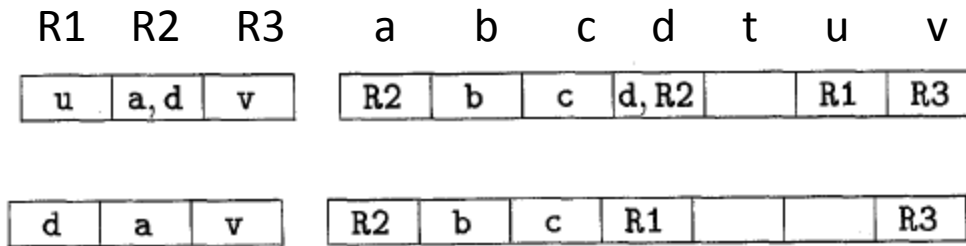
$t = a - b$

$u = a - c$

$v = t + u$

$a = d$

$d = v + u$



$d = v + u$

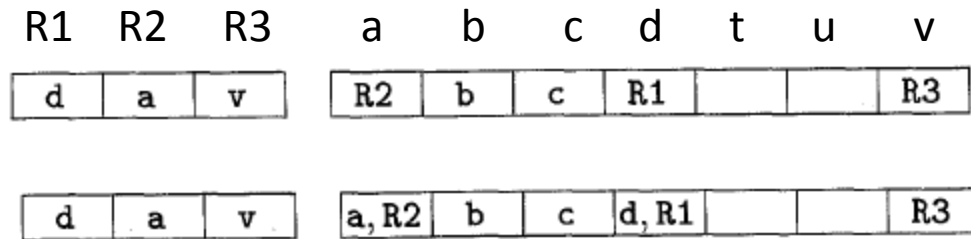
ADD R1, R3, R1

For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction $x = y + z$

- Change the register descriptor for R_x so that it holds only x .
- Change the address descriptor for x so that its only location is R_x . Note that the memory location for x is *not* now in the address descriptor for x .
- Remove R_x from the address descriptor of any variable other than x .

Example

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



exit

ST a, R2

ST d, R1

Ending the basic block: For each variable whose memory location is not up to date generate $ST\ x, R$ (R is the register where x exists at end of the block)

For the instruction $ST\ x, R$, change the address descriptor for x to include its own memory location.

How getReg works?

If y is currently in a register, pick a register already containing y as R_y . Do not issue a machine instruction to load this register, as none is needed.

If y is not in a register, but there is a register that is currently empty, pick one such register as R_y .

What if neither of the above cases are feasible?

How getReg works?

Let R be a candidate register and v is one of the variables stored in R .

If the address descriptor for v says that v is somewhere besides R , then we are OK.

If v is x , the value being computed by instruction I , and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.

Otherwise, if v is not used later (that is, after the instruction I , there are no further uses of v , and if v is live on exit from the block, then v is recomputed within the block), then we are OK.

If we are not OK by one of the first two cases, then we need to generate the store instruction $ST\ v, R$ to place a copy of v in its own memory location. This operation is called a *spill*.

Pick the register with the fewest number of spilled values.

Peephole Optimization

- Improvement of running time or space requirement of target program
- Can be applied to intermediate code or target code
- Peephole: is a small sliding window on a program
- Replace instructions in the peephole by faster/shorter sequence whenever possible
- May require repeated passes for best results

Peephole Optimization: Eliminating Redundant Loads/Stores

```
LD a, R0  
ST R0, a
```

Optimization is obvious

BUT

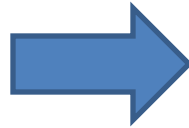
Store instruction must not have a label (why?)

-> the load and store must be in the same basic block

Peephole Optimization: Eliminating Unreachable Code

- Unlabeled instruction immediately following an unconditional jump
- Eliminate jumps over jumps

```
    if debug == 1 goto L1
    goto L2
L1: print debugging information
L2:
```



```
    if debug != 1 goto L2
    print debugging information
L2:
```

Peephole Optimization: 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:
```



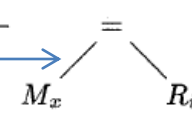
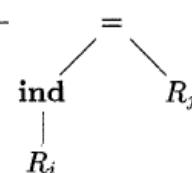
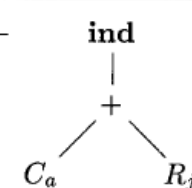
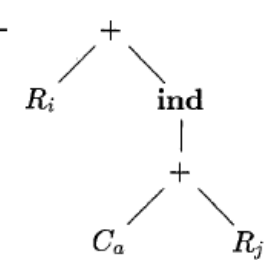
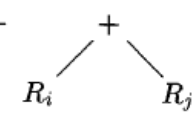
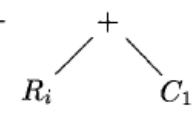
```
    if a < b goto L2  
    goto L3  
    . . .  
L3:
```

Peephole Optimization: Algebraic Simplification and Reduction in Strength

- Get rid of expressions like $X = X + 0$ or $X = X * 1$
- Reduction in strength: replace expensive operations with cheaper ones
 - $x^2 \rightarrow x * x$
 - fixed point instead of floating point
 - Some multiplications with left shifts
 - ...

Tree-Translation Scheme

- Method of code-generation
- Intermediate code is in the form of tree
- replacement ← template {action}
- Tree matching
- Stops when tree is reduced to one node, or no more matching can be done

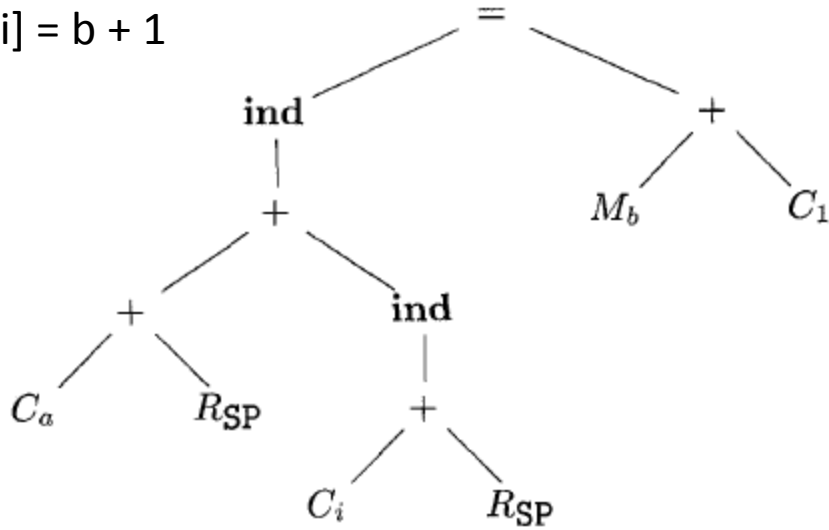
1)	$R_i \leftarrow C_a$	{ LD R_i , # a }
2)	$R_i \leftarrow M_x$	{ LD R_i , x }
3)	$M \leftarrow$ 	{ ST x , R_i }
4)	$M \leftarrow$ 	{ ST $*R_i$, R_j }
5)	$R_i \leftarrow$ 	{ LD R_i , $a(R_j)$ }
6)	$R_i \leftarrow$ 	{ ADD R_i , R_i , $a(R_j)$ }
7)	$R_i \leftarrow$ 	{ ADD R_i , R_i , R_j }
8)	$R_i \leftarrow$ 	{ INC R_i }

Action

Template

Replacement

$a[i] = b + 1$

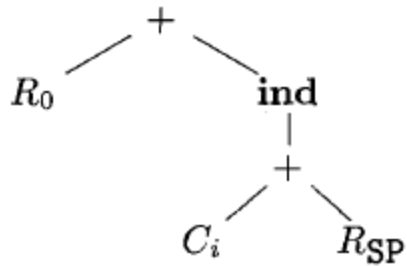
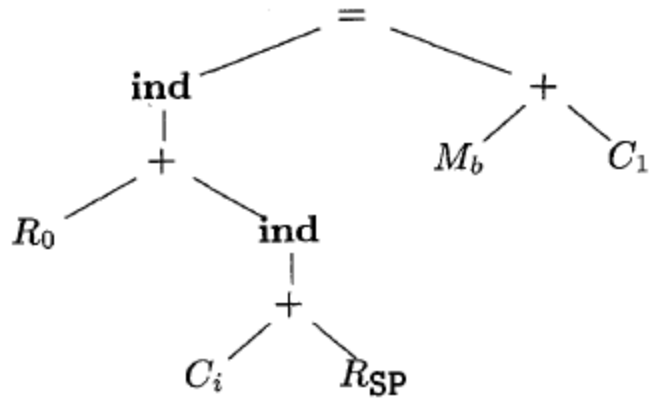


$R_0 \leftarrow C_a$ { LD R0, #a }

$R_0 \leftarrow$ { ADD R0, R0, SP }

1)	$R_i \leftarrow C_a$	{ LD R_i , #a }
2)	$R_i \leftarrow M_x$	{ LD R_i , x }
3)	$M \leftarrow$	{ ST x, R_i }
4)	$M \leftarrow$	{ ST * R_i , R_j }
5)	$R_i \leftarrow$	{ LD R_i , a(R_j) }
6)	$R_i \leftarrow$	{ ADD R_i , R_i , a(R_j) }
7)	$R_i \leftarrow$	{ ADD R_i , R_i , R_j }
8)	$R_i \leftarrow$	{ INC R_i }

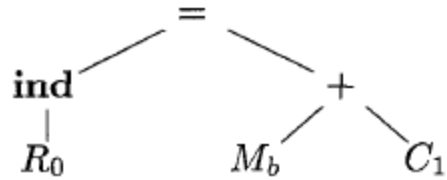
$a[i] = b + 1$



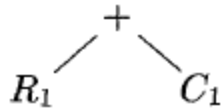
ADD R0, R0, i(SP).

1)	$R_i \leftarrow C_a$	{ LD R_i , #a }
2)	$R_i \leftarrow M_x$	{ LD R_i , x }
3)	$M \leftarrow \begin{matrix} = \\ M_x \quad R_i \end{matrix}$	{ ST x, R_i }
4)	$M \leftarrow \begin{matrix} = \\ \text{ind} \quad R_j \\ \\ R_i \end{matrix}$	{ ST * R_i , R_j }
5)	$R_i \leftarrow \begin{matrix} \text{ind} \\ \\ + \\ C_a \quad R_j \end{matrix}$	{ LD R_i , a(R_j) }
6)	$R_i \leftarrow \begin{matrix} + \\ R_i \quad \text{ind} \\ \\ + \\ C_a \quad R_j \end{matrix}$	{ ADD R_i , R_i , a(R_j) }
7)	$R_i \leftarrow \begin{matrix} + \\ R_i \quad R_j \end{matrix}$	{ ADD R_i , R_i , R_j }
8)	$R_i \leftarrow \begin{matrix} + \\ R_i \quad C_1 \end{matrix}$	{ INC R_i }

$$a[i] = b + 1$$



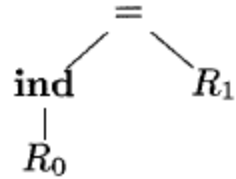
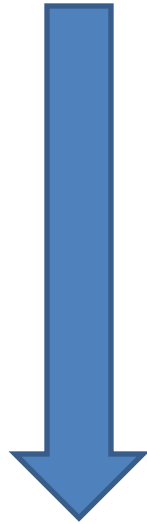
$R_i \leftarrow M_x$ {LD R1, b}



{INC R1}

1)	$R_i \leftarrow C_a$	{ LD R_i , #a }
2)	$R_i \leftarrow M_x$	{ LD R_i , x }
3)	$M \leftarrow \begin{array}{c} = \\ \swarrow \quad \searrow \\ M_x \quad R_i \end{array}$	{ ST x, R_i }
4)	$M \leftarrow \begin{array}{c} = \\ \swarrow \quad \searrow \\ \text{ind} \quad R_j \\ \\ R_i \end{array}$	{ ST * R_i , R_j }
5)	$R_i \leftarrow \begin{array}{c} \text{ind} \\ \\ + \\ \swarrow \quad \searrow \\ C_a \quad R_j \end{array}$	{ LD R_i , a(R_j) }
6)	$R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad \text{ind} \\ \quad \quad \\ \quad \quad + \\ \quad \quad \swarrow \quad \searrow \\ \quad \quad C_a \quad R_j \end{array}$	{ ADD R_i , R_i , a(R_j) }
7)	$R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad R_j \end{array}$	{ ADD R_i , R_i , R_j }
8)	$R_i \leftarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ R_i \quad C_1 \end{array}$	{ INC R_i }

$a[i] = b + 1$



{ ST *R0, R1 }

```
LD R0, #a
ADD R0, R0, SP
ADD R0, R0, i(SP)
LD R1, b
INC R1
ST *R0, R1
```

1)	$R_i \leftarrow C_a$	{ LD R_i , #a }
2)	$R_i \leftarrow M_x$	{ LD R_i , x }
3)	$M \leftarrow \begin{array}{c} = \\ / \quad \backslash \\ M_x \quad R_i \end{array}$	{ ST x, R_i }
4)	$M \leftarrow \begin{array}{c} = \\ / \quad \backslash \\ \text{ind} \quad R_j \\ \\ R_i \end{array}$	{ ST * R_i , R_j }
5)	$R_i \leftarrow \begin{array}{c} \text{ind} \\ \\ + \\ / \quad \backslash \\ C_a \quad R_j \end{array}$	{ LD R_i , $a(R_j)$ }
6)	$R_i \leftarrow \begin{array}{c} + \\ / \quad \backslash \\ R_i \quad \text{ind} \\ \quad \quad \\ \quad \quad + \\ \quad \quad / \quad \backslash \\ \quad \quad C_a \quad R_j \end{array}$	{ ADD R_i , R_i , $a(R_j)$ }
7)	$R_i \leftarrow \begin{array}{c} + \\ / \quad \backslash \\ R_i \quad R_j \end{array}$	{ ADD R_i , R_i , R_j }
8)	$R_i \leftarrow \begin{array}{c} + \\ / \quad \backslash \\ R_i \quad C_1 \end{array}$	{ INC R_i }

So

- skim: 8.8, 8.9.3, 8.9.4, 8.9.5, 8.10, 8.11
- Read: rest of 8.6->8.9

End of New Material!!