ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 12a

Code Generation

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy

Simple Code Generator

- Generates target code from three-address code
- For each three-address code operator there is a target code operator (e.g., ADD for ‘+’)
- Computed results can be left in registers as long as possible, except:
  - their register is needed for another computation
  - just before a procedure call, jump, or labeled statement

Target Machine

- `n` general-purpose registers, `R0,R1,...,Rn-1`
- Instructions (*`op`* `source, destination`)
  - `MOV` (move *source* to destination)
  - `ADD` (add *source* to destination)
  - `SUB` (subtract *source* from destination)

Target Machine

Memory Addressing

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

contents(x): contents of register or memory

The cost is 1 *only* when memory is addressed.

Instruction Cost

- The cost of an instruction is one plus the added costs associated with the address modes used
- Examples

```
MOV B, R0          cost = 6
ADD c, R0
MOV R0, a

MOV b, a           cost = 6
ADD c, a

ADD R2, R1     cost = 3
MOV R1, a
```

Challenges

- Consider the statement `a := b + c`
- Possible target code generations

```
// assumes variables are in registers
ADD Rj, Ri      (cost 1)
// assumes b is in Ri
ADD c, Ri       (cost 2)
// good if c is going to be used later
MOV c, Rj       (cost 3)
ADD Rj, Ri
```

- Many different options

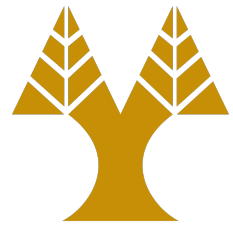
Register and Address Descriptors

- We use descriptors to keep track of register contents and address for names
  1. A register descriptor keeps track of what is currently in each register. We assume that initially the register descriptor shows that all registers are empty.
  2. An address descriptor keeps track of the location (or locations) where the current value of the name can be found at run-time. The location might be a register, a stack location, a memory address, or some set of these. This information can be stored in the symbol table.

Code Generation Algorithm

- For each three-address statement of the form `x := y op z` we perform the following actions.
  1. Invoke a function *getreg* to determine the location `L` where the result of the computation should be stored. `L` can be a register or memory location.
  2. Consult the address descriptor for `y` to determine `y'`, (one of) the current location(s) of `y` (prefer a register to a memory location). If the value of `y` is not already in `L`, generate the instruction `MOV y', L` to place a copy of `y` in `L`.
  3. Generate the instruction `OP z', L` where `z'` is a current location for `z` (prefer a register to a memory location).
  4. if the current values of of `y` and/or `z` have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of `x := y op z`, those registers no longer contain `y` and/or `z`, respectively.

Special Case



`x := y`

- If `y` is in a register, simply change the register and address descriptors to record that the value of `x` is now found in the register holding the value of `y`.
- If `y` has no next use and is not live on exit from the block, the register no longer holds the value of `y`.
- If `y` is only in memory, we use *getreg* to find a register in which to load `y` and make the register the location of `x`.

getreg

- Returns the location `L` to hold the value of `x` for the assignment `x := y op z`.
  1. If the name `y` is in a register that holds the value of no other names, and `y` is not live and has no next use after execution of the statement, then return the register of `y` for `L`.
  2. Failing (1), return an empty register for `L`.
  3. Failing (2), if `x` has a next use in the block, or `op` is an operator, such as indexing, that requires a register, find an occupied register `R`. Store the value of `R` into a memory location (by `MOV R, M`) and return `R`.
  4. If `x` is not used in the block, or no suitable occupied register can be found, select the memory location of `x` as `L`.

Example

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

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

STATEMENTS	CODE	REGISTERS	ADDRESSES
		registers empty	
t := a - b	MOV a,R0	R0 contains t	t in R0
	SUB b,R0		
u := a - c	MOV a,R1	R0 contains t	t in R0
	SUB c,R1	R1 contains u	u in R1
v := t + u	ADD R1,R0	R0 contains v	u in R1
		R1 contains u	v in R0
d := v + u	ADD R1,R0	R0 contains d	d in R0
	MOV R0,d		d in R0/m

Register Allocation and Assignment

- Instructions involving only register operands are shorter and faster than those involving memory operands
- Register Allocation
  - What values should reside in registers
- Register Assignment
  - Which register should value reside

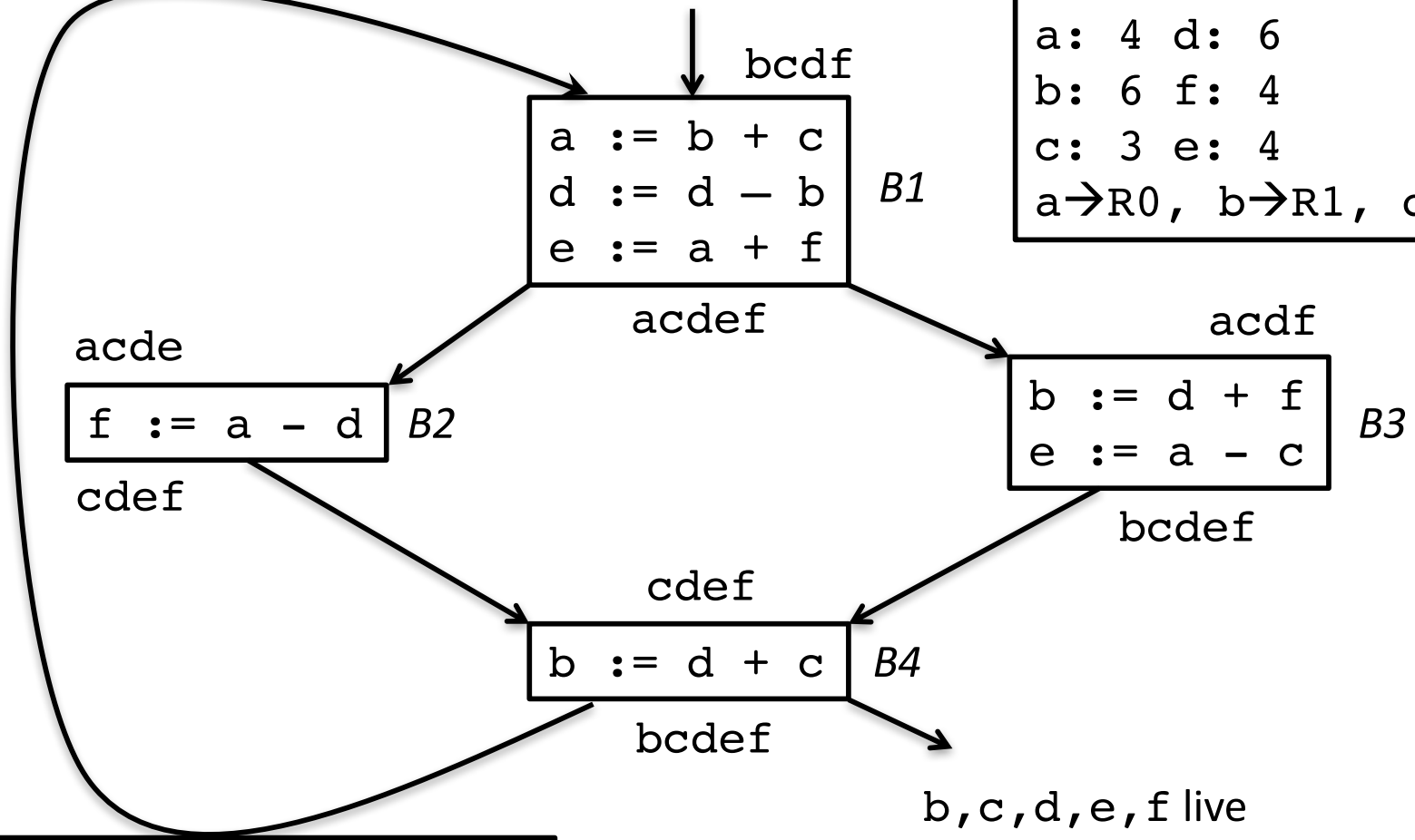
Usage Counts

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

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

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

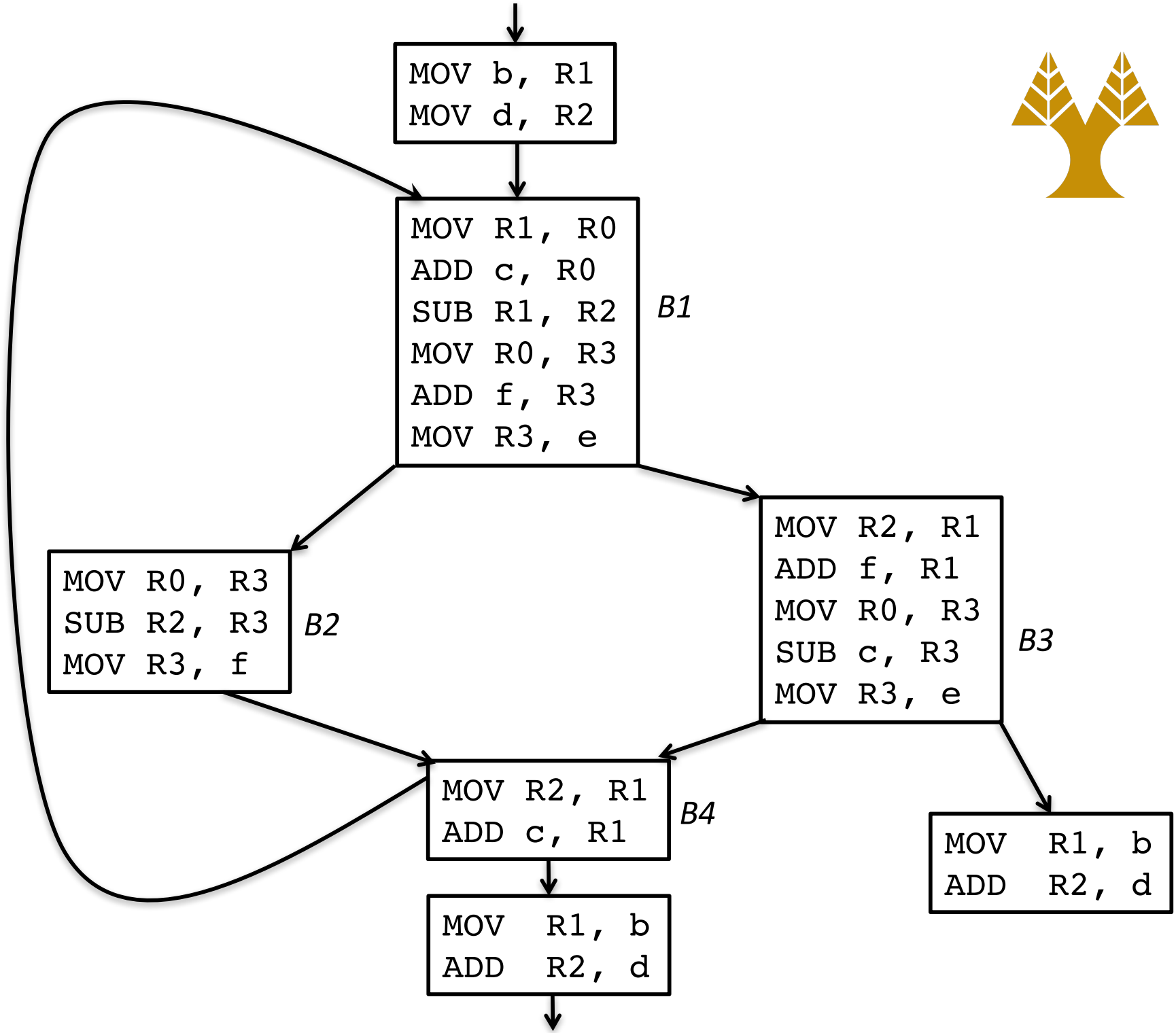
Usage cost of a
- a is live and is assigned on exit from B1
but not from B2, B3, or B4: ∑2**live*(a, B) = 2
- *use*(a, B1) = 0, *use*(a, B2) = 1, *use*(a, B3)=1, *use*(a, B4) = 0: ∑use(a, B) = 2
- Usage cost (a) = 4




```
a: 4  d: 6
b: 6  f: 4
c: 3  e: 4
a→R0, b→R1, d→R2
```

Usage cost of e
- e is live and is assigned only in B1

Usage cost of d
- d is used in B1, B2, B3, and B4 (4).
- d is live and is assigned in B1 (2x1).


MOV b, R1
MOV d, R2
MOV R1, R0
ADD c, R0
SUB R1, R2
MOV R0, R3
ADD f, R3
MOV R3, e
B1
MOV R0, R3
SUB R2, R3
MOV R3, f
B2
MOV R2, R1
ADD f, R1
MOV R0, R3
SUB c, R3
MOV R3, e
B3
MOV R2, R1
ADD c, R1
B4
MOV R1, b
ADD R2, d
MOV R1, b
ADD R2, d