Simple Code Generator, Register allocation

Simple Code Generator

- Generates target code for a sequence of three-address statements
  - Next-use information is used
- For each operator in a statement there is a target-language operator
- Uses new function *getreg* to assign registers to variables

Simple Code Generator

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

Example

- a := b+c
- ADD Ri, Rj if ‘b’ and ‘c’ are in registers Ri and Rj and this costs 1 and result in Rj
- ADD c, Ri if ‘c’ is not in register and ‘b’ in Register and this costs 2
- MOV c, Rj

  ADD Rj, Ri will costs 3 for the same scenario

Data structures used

- Register Descriptor – used to keep track of which variable 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`**”

Data structures used

- Address Descriptor – used to keep track of the location where the current value of the variable 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 Code Generation Algorithm

Input : Sequence of 3-address statements from a basic block. For each statement *x* := *y* op *z*

- Set location *L* = *getreg*(*y*, *z*) to store the result of y op z
- If $y \notin L$ then generate
  MOV *y'*,*L*
  where *y'* denotes one of the locations where the value of *y* is available - choose register if possible

The Code-Generation Algorithm

- Generate
  `OP` *z'*,*L*
  where *z'* is one of the locations of *z*;
  Update register/address descriptor of *x* to include *L*
- If *y* and/or *z* has no next use and is stored in register, update register descriptors to remove *y* and/or *z*

getreg ( ) algorithm

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. Return a memory location

Example

- Consider the following example:

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

- Three address:

  t: = a-b

  u := a-c

  v := t + u

  d := v +u

Code generation Sequence

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

Other types of Statements

Statement	i in Register Ri		i in Memory Mi		i in Stack	
	Code	Cost	Code	Cost	Code	Cost
a := b[i]	MOV b[Ri], R	2	MOV Mi, R MOV b[R], R	4	MOV Si(A), R MOV b(R), R	4
a[i] := b	MOV b, a[Ri]	3	MOV Mi, R MOV b, a[R]	5	MOV Si(A), R MOV b, a(R)	5

Other types of Statements

Statement	p in Register Rp		p in Memory Mp		p in Stack	
	Code	Cost	Code	Cost	Code	Cost
a := *p	MOV *Rp, a	2	MOV Mp, R MOV *R, R	3	MOV Sp(A), R MOV *R, R	3
*p := a	MOV a, *Rp	2	MOV Mp, R MOV a, *R	4	MOV a, R MOV R, *Sp(A)	4

Conditional Statements

- Conditional Jumps implemented
  - Branch if the value of a register is negative, zero, positive, non-negative, non-zero, non-positive
  - Uses a set of condition codes to indicate whether the computed quantity of a register is zero, positive or negative

Conditional Statements

- First case: if x < y goto z , is to be evaluated, then subtract y from x which is in register R and then jump to z if R is negative
- Second case: CMP x, y sets the condition code to positive if x> y and so on

Conditional Statements

- CMP x, y
- CJ < z
  - Jump to z if value is negative
- <, >, <=, >=, ==, !=

Example

- x := y + z
- If x < 0 goto z

--

```
MOV y, R0
ADD z, R0
MOV R0, x      // x is the condition code
CJ < z
```

Register Allocation

- The *getreg* algorithm is simple but not optimal
  - All live variables in registers are stored at the end of a block
- *Global register allocation* assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in loops can be beneficial

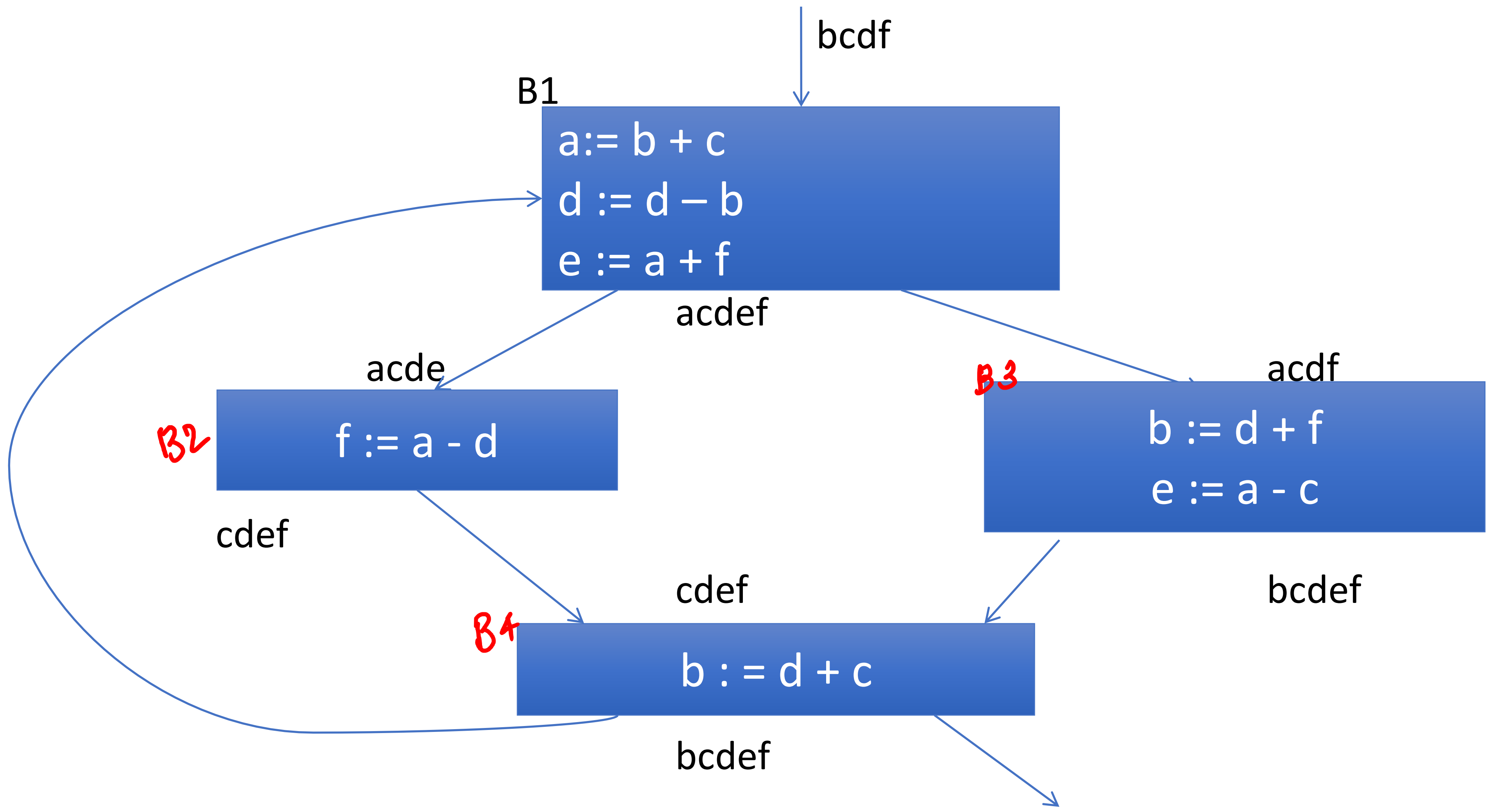
Register allocation

- Suppose loading a variable x has a cost of 2
- Suppose storing a variable x has a cost of 2
- Benefit of allocating a register to a variable x within a loop L is

  $$\sum_{B \in L} \left(use(x, B) + 2\, live(x, B) \right)$$

  where
  - $use(x, B)$ is the number of times x is used in B prior to any definition of x
  - $live(x, B)$ = true if x is live on exit from B and is assigned a value in B

Example



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

Example

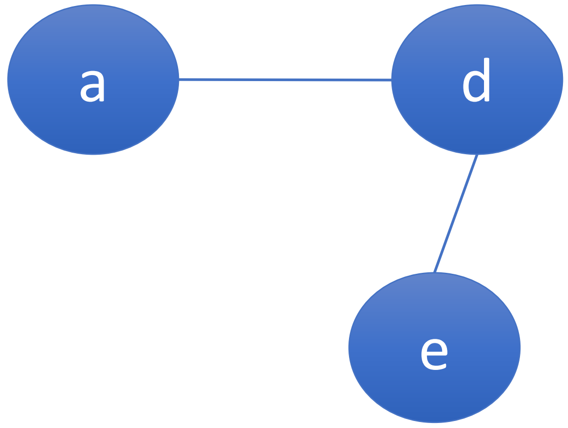
Block→	B1		B2		B3		B4		Total
↓ VARIABLE	Use	Live	Use	Live	Use	Live	Use	Live	
a	0	1	1	0	1	0	0	0	4
b	2	0	0	0	0	1	0	1	6
c	1	0	0	0	1	0	1	0	3
d	1	1	1	0	1	0	1	0	6
e	0	1	0	0	0	1	0	0	4
f	1	0	0	1	1	0	0	0	4

Global Register Allocation - Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored to free a register - Spilling
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a interference graph based on how variable interfere with each other
- Find a *k*-coloring for the graph, with *k* the number of registers

Register interference graph

- Nodes are symbolic registers
- Edge connects two nodes if one is live at a point where other is defined



So need two registers

Summary

- Simple code generator algorithm
- Register descriptor and address descriptor
- Register allocation
  - Use and live statistics
  - Graph coloring