Code Generation (Part -3)

Course : 2CS701/IT794 – Compiler Construction

Prof Monika Shah

Nirma University

Ref : Ch.9 Compilers Principles, Techniques, and Tools by Alfred Aho, Ravi Sethi, and Jeffrey Ullman

Glimpse

Part-1

- Introduction
- Code Generation Issues

Part-2

- Basic Code Generation Case Study
- Introduction to Basic Block, and Control Flow Diagram

Part-3

- **Algorithm : Code Generator using getReg( )**
- **Register Allocation Problem**
- **Use and Live to find optimal use of registers**
- **Global Register Allocation with Graph Coloring**

A Code Generator

- Generates target code
- Input : Sequence of three-address statements
- Approach :
  - Uses next-use information
  - Uses new function *getreg* to assign registers to variables
  - Computed results are kept in registers as long as possible, if:
    - 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

The Code Generation Algorithm

- For each statement $x := y$ op z

1. Find location $L = getreg\ (y, z)$
2. 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)
3. Generate
   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

The *getreg(y,z)* 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

Simplified version:

Return Register allocated to Y,
iff

- The register has no next-use
- No other register holds Y

Else, return any free register

Else, store any R to M
and return this freed R

Else, return Memory Location

Register and Address Descriptors

- A *register descriptor* : Specify what the register contains at a particular point in the code, e.g.

  `MOV y,R0` "**`R0`** contains **`a`**"

- An *address descriptor* : Specify locations, where value of variables(address) are stored at runtime" e.g.

  `MOV x,R0`
  `MOV R0,R1` "**`x`** in **`R0`** and **`R1`**"

Code Generation Example

Statements	*Code Generated*	*Register Descriptor*	*Address Descriptor*	
`t := a - b`	`MOV a,R0` `SUB b,R0`	Registers empty `R0` contains `t`	`t` in `R0`	getreg( ) returns R0 for a
`u := a - c`	`MOV a,R1` `SUB c,R1`	`R0` contains `t` `R1` contains `u`	`t` in `R0` `u` in `R1`	a is not in Registers getreg( ) returns R1 for a
`v := t + u`	`ADD R1,R0`	`R0` contains `v` `R1` contains `u`	`v` in `R0` `u` in `R1`	t is available in R0 getreg( ) returns R0 for t
`d := v + u`	`ADD R1,R0` `MOV R0,d`	`R0` contains `d`	`d` in `R0` **and** **memory**	v is available in R0 u is available in R1

Advantage and disadvantage of getReg

- +ve : Simple
- -ve : Registers hold values for single basic block
- -ve: sub-optimal
  - All live variables in registers are stored (flushed) at the end of a block
  - All live variables at entry are loaded in the registers

Register Allocation Problem

- Register access is faster than other memory
- Intermediate Code use many temporary variables and Registers are limited
- Register allocation :
  - Rewrite code to use fewer temporary variables than machine registers
  - Assign more temporaries to a register

Example

Consider the program

a := c + d

e := a + b

f := e - 1

r1 := r2 + r3
r1 := r1 + r4
r1 := r1 - 1

with the assumption that a and e die after use. Temporary a can be "reused" after e := a + b Can allocate a, e, and f all to one register (r1):

Basic Rule : t1 and t2 can share same registers , if at-most only one (t1 or t2) live at any point of time

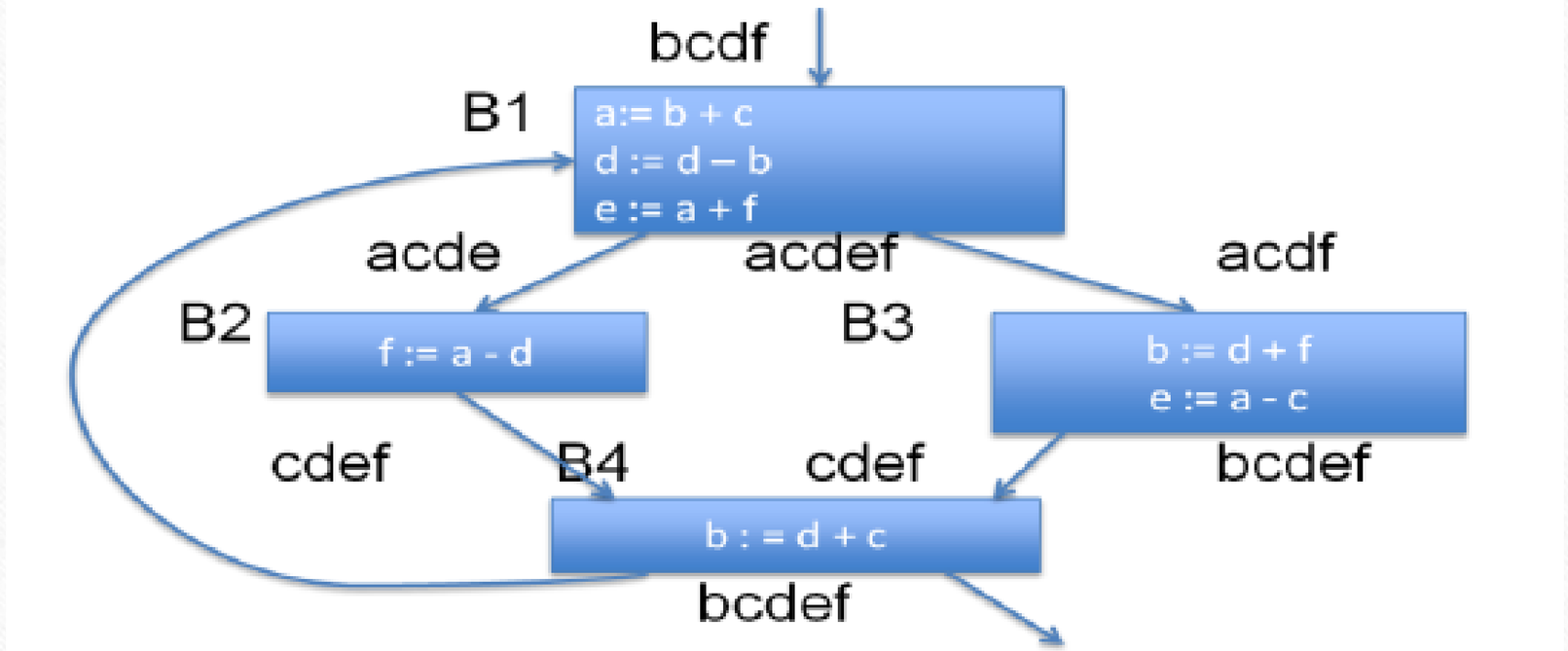
Solution to Reduce number of load and store

- **keep these registers consistent across basic block boundaries**
- Keeping variables in registers in looping code can result in big savings
- *Global register allocation* assigns variables to limited number of available registers and attempts to **keep these registers consistent across basic block boundaries**

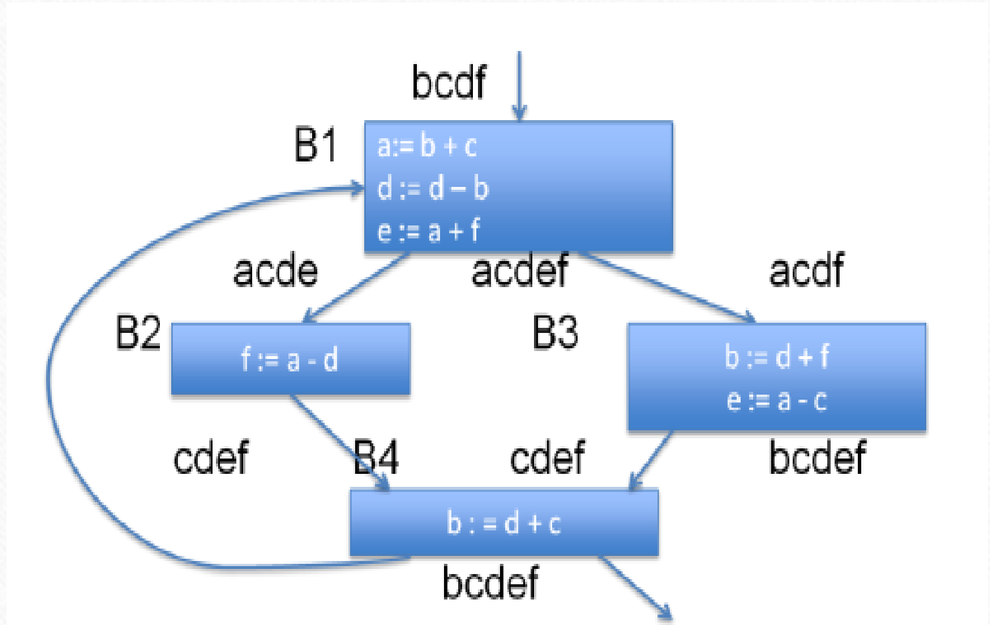
Allocating Registers in Loops

- 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
  $\sum_{B \in L} (use(x, B) + 2\ live(x, B))$
  where,
  - $use(x, B)$ is the number of times x is used in B
  - cost (use) is due to variables need to be loaded before use
  - $live(x, B)$ = true if x is live on exit from B
  - cost (live) is due to variables need to be stored at end of block if live at end,
  - cost (store) is due to variables need to be loaded at enty of block if live at beginning of block

Control flow Graph of an Sample code



Control flow Graph of an Sample code



ID	Use (B1)	Live (B1)	Use (B2)	Live (B2)	Use (B3)	Live (B3)	Use (B4)	Live (B4)	Total cost
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

Reduce cost of load and store variables

- Basic Approach to save cost of load and save
  - Keep variables allocated to registers
    - Do not store at end of block, if live after
    - Do not load at beginning of block, if live at entry of block
- What if Number of variables in a loop are more than number of machine registers?
  - Give priority to those variables having maximum cost benefit
  - For example. Variables b and d have maximum cost benefit as per previous slides

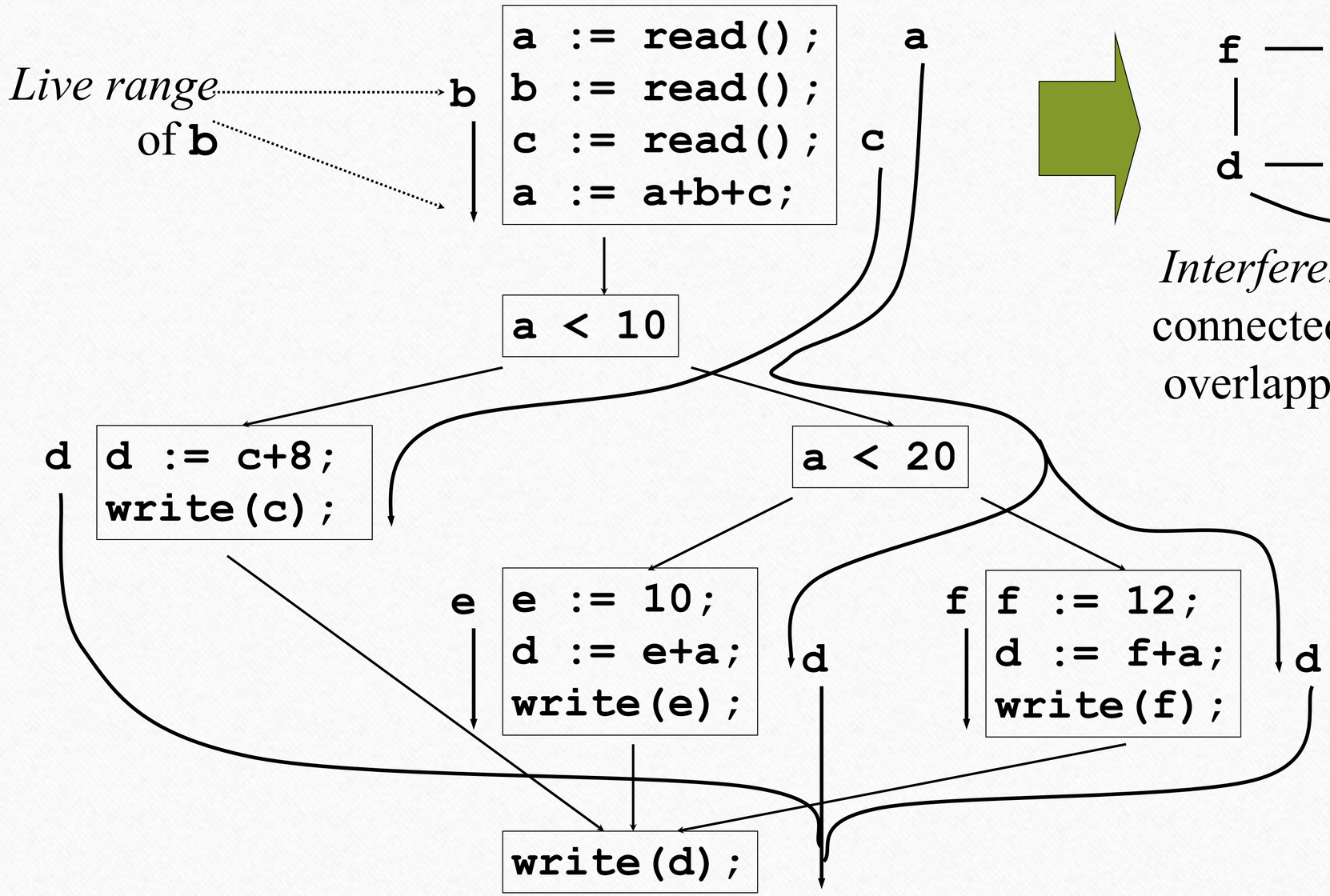
Global Register Allocation with Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored (spilled) to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a *conflict graph* (*interference graph*)
- Find a *k*-coloring for the graph, with *k* the number of registers

Register Allocation with Graph Coloring: Example

```
a := read();
b := read();
c := read();
a := a + b + c;
if (a < 10) {
    d := c + 8;
    write(c);
} else if (a < 20) {
    e := 10;
    d := e + a;
    write(e);
} else {
    f := 12;
    d := f + a;
    write(f);
}
write(d);
```

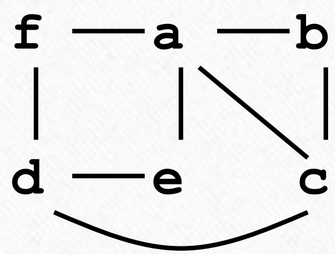
Register Allocation with Graph Coloring: Live Ranges



Interference graph: connected vars have overlapping ranges

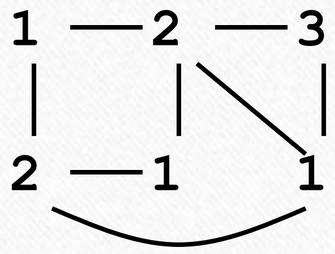
Register Allocation with Graph Coloring: Solution

Approach : Assign different color (registers) to connected variables



Interference graph

Solve



Three registers:

```
a = r2
b = r3
c = r1
d = r2
e = r1
f = r1
```

```
r2 := read();
r3 := read();
r1 := read();
r2 := r2 + r3 + r1;
if (r2 < 10) {
    r2 := r1 + 8;
    write(r1);
} else if (r2 < 20) {
    r1 := 10;
    r2 := r1 + r2;
    write(r1);
} else {
    r1 := 12;
    r2 := r1 + r2;
    write(r1);
}
write(r2);
```