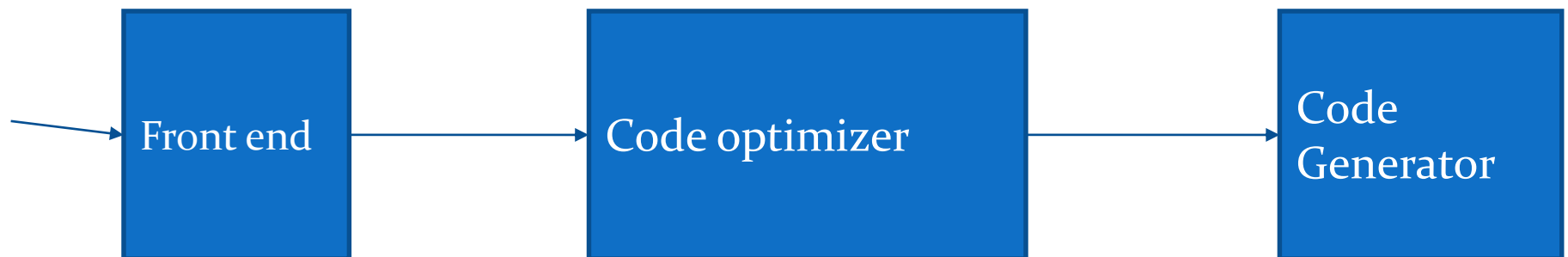


Code Generation

The final phase of a compiler is code generator

It receives an intermediate representation (IR) with supplementary information in symbol table

Produces a **semantically equivalent** target program



Code Generation

- Main tasks:
 - Instruction selection
 - Register allocation and assignment
 - Instruction ordering

Code Generation

- Issues in the Design of Code Generator
 - The most important criterion is that it produces correct code
- Input to the code generator
 - IR + Symbol table
 - We assume front end produces low-level IR, i.e. values of names in it can be directly manipulated by the machine instructions (e.g. Three address code).
 - Syntactic and semantic errors have been already detected
- The target program
 - Common target architectures are: RISC, CISC and Stack based machines

Code Generation

- Complexity of mapping depends on
 - the level of the IR
 - 3AC (Quadruples / Triples / Indirect triples)
 - VM instructions (bytecodes / stack machine codes)
 - Linear representations (postfix)
 - Graphical representation (syntax trees / DAGs)
 - the nature of the instruction-set architecture
 - RISC (many registers, simple addressing modes, simple ISA)
 - CISC (few registers, variety of addressing modes, several register classes, variable length instructions, instructions with side-effects)
 - Stack machine (push / pop, stack top uses registers, used in JVM, JIT compilation)
 - the desired quality of the generated code

Code Generation

$x = y + z$

```
LD    R0, y
ADD   R0, z
ST     x, R0
```

$a = b + c$
 $d = a + e$

```
LD R0, b
ADD R0, c
ST  a, R0
LD  R0, a
ADD R0, e
ST  d, R0
```

Code Generation

- Register allocation
- Two subproblems
 - Register allocation: selecting the set of variables that will reside in registers at each point in the program
 - Register assignment: selecting specific register that a variable will reside in
- Finding an optimal assignment of registers to variables is NP-Complete
- Complications imposed by the hardware architecture

Code Generation

- Instruction Selection
- Straight forward code if efficiency is not an issue

a=b+c
d=a+e

Mov R0, b
Add R0, c
Mov a, R0
Mov R0, a
Add R0, e
Mov d, R0

a=a+1

Mov R0, a
Add R0, #1
Mov a, R0

Code Generation

- Instruction Selection
- Straight forward code if efficiency is not an issue

a=b+c
d=a+e

Mov R0, b
Add R0, c
Mov a, R0
Mov R0, a
Add R0, e
Mov d, R0

(redundant code, can be eliminated)

a=a+1

Mov R0, a
Add R0, #1
Mov a, R0

(if there is an
instruction like
“Inc a”, that can
be used instead)

Code Generation

- Partition the intermediate code into basic blocks
 - Basic blocks
 - sequence of statements in which flow of control enters at beginning and leaves at the end
- Algorithm to identify basic blocks
 - *determine leader*
 - *first statement is a leader*
 - *any target of a goto statement is a leader*
 - *any statement that follows a goto statement is a leader*
 - *for each leader its basic block consists of the leader and all statements up to next leader*

Code Generation

- Flow graphs
 - add control flow information to basic blocks
 - nodes are the basic blocks
 - there is a directed edge from B1 to B2 if B2 can follow B1 in some execution sequence, i.e.
 - either there is a jump from the last statement of B1 to the first statement of B2, or
 - B2 follows B1 in natural order of execution
 - initial node: block with first statement as leader
- The flow of control can only enter the basic block through the first instruction in the block. That is, there are 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.

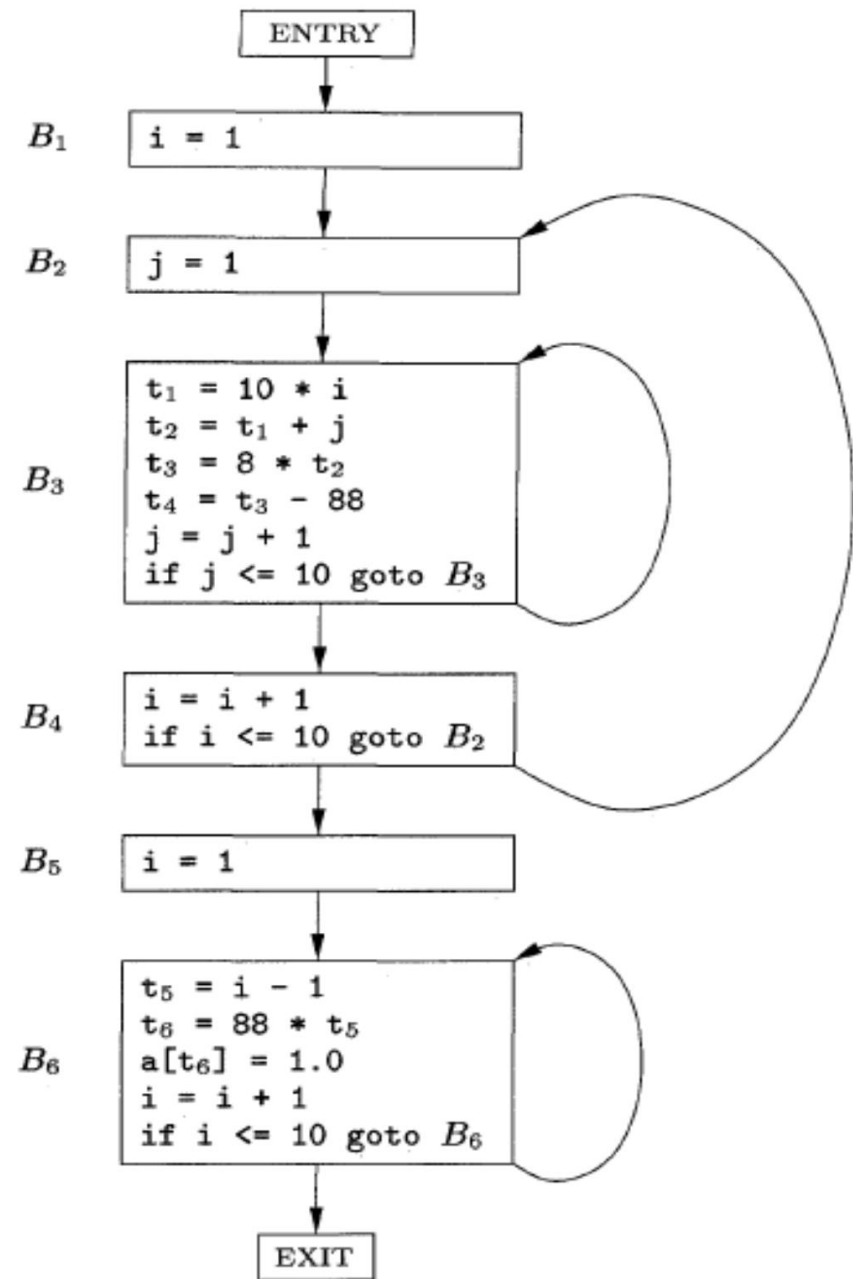
Example: Intermediate code to set a 10*10 matrix to an identity matrix

```
for i from 1 to 10 do
    for j from 1 to 10 do
         $a[i, j] = 0.0;$ 
for i from 1 to 10 do
     $a[i, i] = 1.0;$ 
```

```
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)
```

Example: Intermediate code to set a 10*10 matrix to an identity matrix

```
for i from 1 to 10 do
  for j from 1 to 10 do
     $a[i, j] = 0.0;$ 
  for i from 1 to 10 do
     $a[i, i] = 1.0;$ 
```



Code Generation

- If a register contains a value for a name that is no longer needed, we should re-use that register for another name
- (rather than using a memory location) .
- So it is useful to determine whether/when a name is used again in a block
- *Definition: Given statements i , j , and variable x , If i assigns a value to x , and j has x as an operand, and no intervening statement assigns a value to x , then j uses the value of x computed at i .*

EXAMPLE -1:

(5) $X := \dots$

... (no ref to X) ...

(14) $\dots := \dots X \dots$

X is live at (5)

(because the value computed at (5) is used later in the basic block.)

X's next use (after (5)) is at **(14)**. (It is a good idea to **keep X in a register** between (5) and (14).)

EXAMPLE -2:

(14) $\dots := \dots X \dots$

... (no ref to X) ...

(25) $X := \dots$

X is dead at (14)

(because its value has no further use in the block.)

Don't keep X in a register after (14).

Liveness and next-use information

- Requirement is to determine for each three address statement $x=y+z$ what the next uses of x , y and z are.
- Algorithm:
 - scan each basic block backwards
 - assume all temporaries are dead on exit and all user variables are live on exit
- Example: Suppose we are scanning
 $i : X := Y + Z$ in backward scan
- Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x , y , and z .
- In the symbol table, set x to "not live" and "no next use".
- In the symbol table, set y and z to "live" and the next uses of y and z to i .

Statement number	Instruction	Liveness and Nextuse initialized	Modified liveness and nextuse after statement 'j'	Modified liveness and nextuse after statement 'i'	Comments
i	a := b + c	<i>live(a)</i> = true, <i>live(b)</i> = true, <i>live(t)</i> = true, <i>nextuse(a)</i> = none, <i>nextuse(b)</i> = none, <i>nextuse(t)</i> = none	<i>live(a)</i> = true <i>nextuse(a)</i> = j <i>live(b)</i> = true <i>nextuse(b)</i> = j <i>live(t)</i> = false <i>nextuse(t)</i> = none	<i>live(a)</i> = false <i>nextuse(a)</i> = none <i>live(b)</i> = true <i>nextuse(b)</i> = i <i>live(c)</i> = true <i>nextuse(c)</i> = i <i>live(t)</i> = false <i>nextuse(t)</i> = none	'a' is said to be false and no nextuse. 'b' and 'c' are said to live at statement 'i' and their next use is at 'i'. 't's liveness
					and nextuse remains as it is computed in statement 'j'
j	t := a + b	<i>live(a)</i> = true, <i>live(b)</i> = true, <i>live(t)</i> = true, <i>nextuse(a)</i> = none, <i>nextuse(b)</i> = none, <i>nextuse(t)</i> = none	<i>live(a)</i> = true <i>nextuse(a)</i> = j <i>live(b)</i> = true <i>nextuse(b)</i> = j <i>live(t)</i> = false <i>nextuse(t)</i> = none	<i>live(a)</i> = true <i>nextuse(a)</i> = j <i>live(b)</i> = true <i>nextuse(b)</i> = j <i>live(t)</i> = false <i>nextuse(t)</i> = none	'a' and 'b' are used here and hence their nextuse is at 'j' and they are said to be live. 't' is computed and hence nextuse is none and live information is false

	SyTab-Info			Instr.-Info		
	live			next_use		
i	x	y	z	x	y	z
(4) x := z+y	F	T	T	4	4	
(3) y := z-7	F	F	T		3	4 4
(2) z := x*5	T	F	F	2		3
(1) x := y+z	F	T	T	1	1	2

- The data in each row reflects the state in the symbol table and in the data section of instruction i **after** i has been processed.

Example

STATEMENT

1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

7: no temporary is live
6: t_6 :use(7), t_4 t_5 not live
5: t_5 :use(6)
4: t_4 :use(6), t_1 t_3 not live
3: t_3 :use(4), t_2 not live
2: t_2 :use(3)
1: t_1 :use(4)

Symbol Table

t1	dead	use in 4
t2	dead	use in 3
t3	dead	use in 4
t4	dead	use in 6
t5	dead	use in 6
t6	dead	use in 7

Example of Reuse

1		
2		
3	t_1	t_2
4		t_3
5	t_4	
6		t_5
7	t_6	

$$1: t_1 = a * a$$

$$2: t_2 = a * b$$

$$3: t_2 = 2 * t_2$$

$$4: t_1 = t_1 + t_2$$

$$5: t_2 = b * b$$

$$6: t_1 = t_1 + t_2$$

$$7: X = t_1$$

A Simple Code Generator

- Leave computed result in a register as long as possible
- Store in memory only at the end of a basic block or when that register is needed for another computation
 - On exit from a basic block, store only live variables which are not in their memory locations already
 - (use address descriptors to determine the latter)
 - If liveness information is not known, assume that all variables are live at all time

Code Generator

- consider each statement
- remember if operand is in a register
- Use Register descriptor
 - Keep track of what is currently in each register.
 - Initially all the registers are empty
- Register descriptors
 - Tracks <register, variable name> pairs
 - A single register can contain values of multiple names, if they all are copies

Code Generator

- Use Address descriptor
 - Keep track of location where current value of the name can be found at runtime
 - Tracks `<variable name, locations>` pairs
 - The location might be a register, stack, memory address or a set of those
 - A single name may have its value in multiple locations, such as, memory, register, and stack

Code Generator

1. for each $X = Y \text{ op } Z$ do
2. invoke a function **getreg** to determine location **L** where **X** must be stored. Usually L is a register.
3. Consult **address descriptor of Y** to determine its present location. Prefer a register for Y' (where Y' represents the copied value of Y)
4. If value of Y not already in L, generate
Mov L, Y'
5. Generate **op L, Z'**
6. Again prefer a register for Z.

Code Generator

7. Update **address descriptor of X** to indicate X is in L.
8. If L is a register, **update register descriptor of L** to indicate that it contains X and remove X from all other register descriptors.
9. If current value of Y and/or Z have **no next use and are dead on exit from block** and are in registers,
 change register descriptor to indicate that they no longer contain Y and/or Z.

Function getreg

1. If **Y is in register** (that holds no other values) and **Y is not live** and **has no next use** after $X = Y \text{ op } Z$

then **return register of Y for L**. ---(1)

2. Failing (1) **return an empty register** ---(2)

3. Failing (2) if X has a next use in the block or op requires register **then get a register R , store its content v into M** (by $\text{Mov } M, v$) and use it.

4. else **select memory location X as L**

(If we are not OK with one of the first two cases, then we need to generate the move instruction $\text{Mov } M, v$ to place a copy of v in its own memory location. This operation is called a spill. - [Register Spilling](#))

Stmt	code	reg desc	addr desc
$t_1 = a - b$	mov R_0, a sub R_0, b	R_0 contains t_1	t_1 in R_0
$t_2 = a - c$	mov R_1, a sub R_1, c	R_0 contains t_1 R_1 contains t_2	t_1 in R_0 t_2 in R_1
$t_3 = t_1 + t_2$	add R_0, R_1	R_0 contains t_3 R_1 contains t_2	t_3 in R_0 t_2 in R_1
$d = t_3 + t_2$	add R_0, R_1 mov d, R_0	R_0 contains d	d in R_0 and memory

Stmt	code	reg desc	addr desc
$t_1 = a - b$	mov R_0, a sub R_0, b	R_0 contains t_1	t_1 in R_0
$t_2 = a - c$	mov R_1, a sub R_1, c	R_0 contains t_1 R_1 contains t_2	t_1 in R_0 t_2 in R_1
$t_3 = t_1 + t_2$	add R_0, R_1	R_0 contains t_3 R_1 contains t_2	t_3 in R_0 t_2 in R_1
$d = t_3 + t_2$	add R_0, R_1 mov d, R_0	R_0 contains d	d in R_0 d in R_0 and memory

Global register allocation

Previous algorithm does local (block based) register allocation

This results that all live variables be stored at the end of block

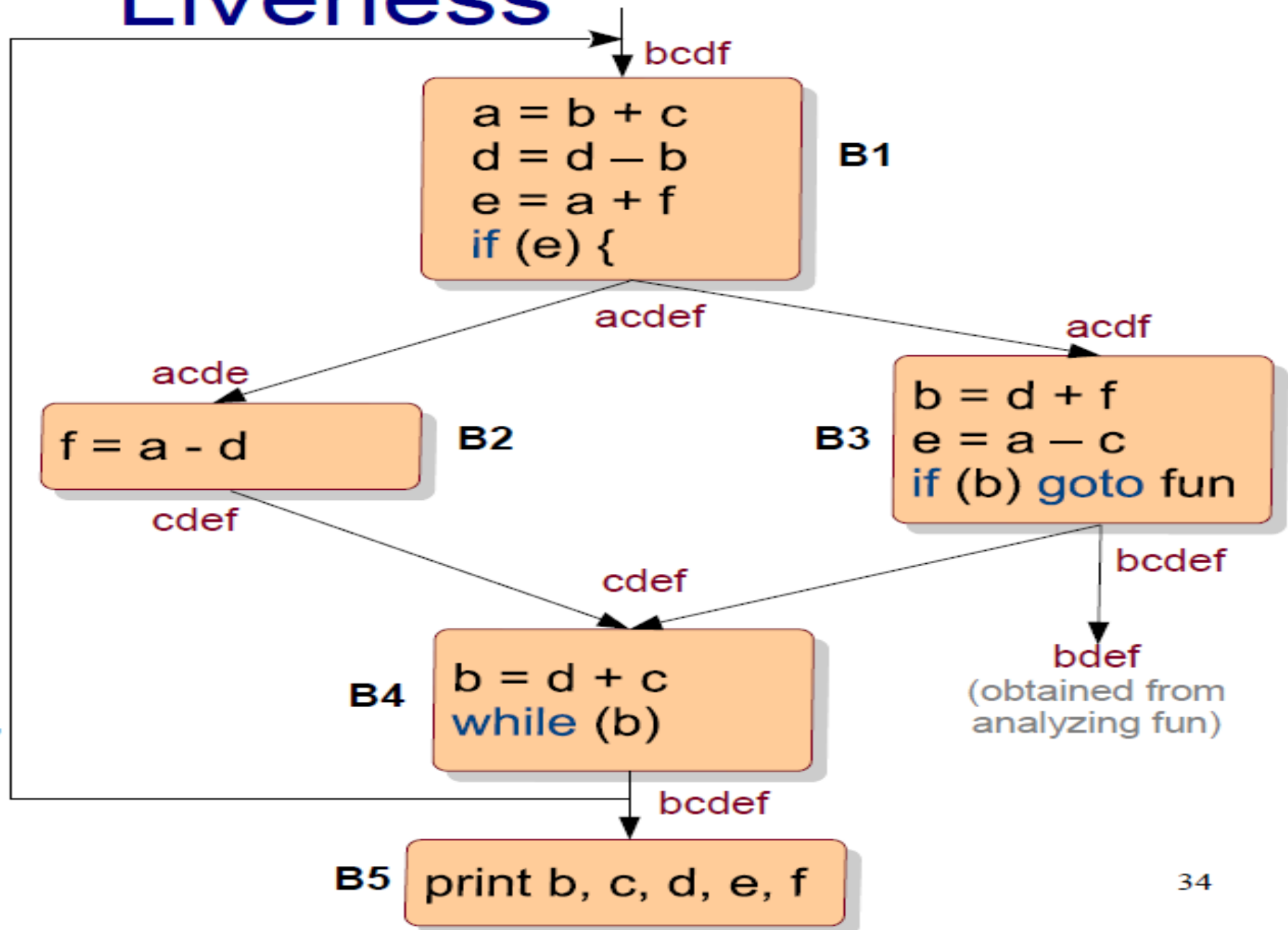
To save some of these stores and their corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally)

Some options are:

- Keep values of variables used in loops inside registers

- Use graph coloring approach for more globally allocation

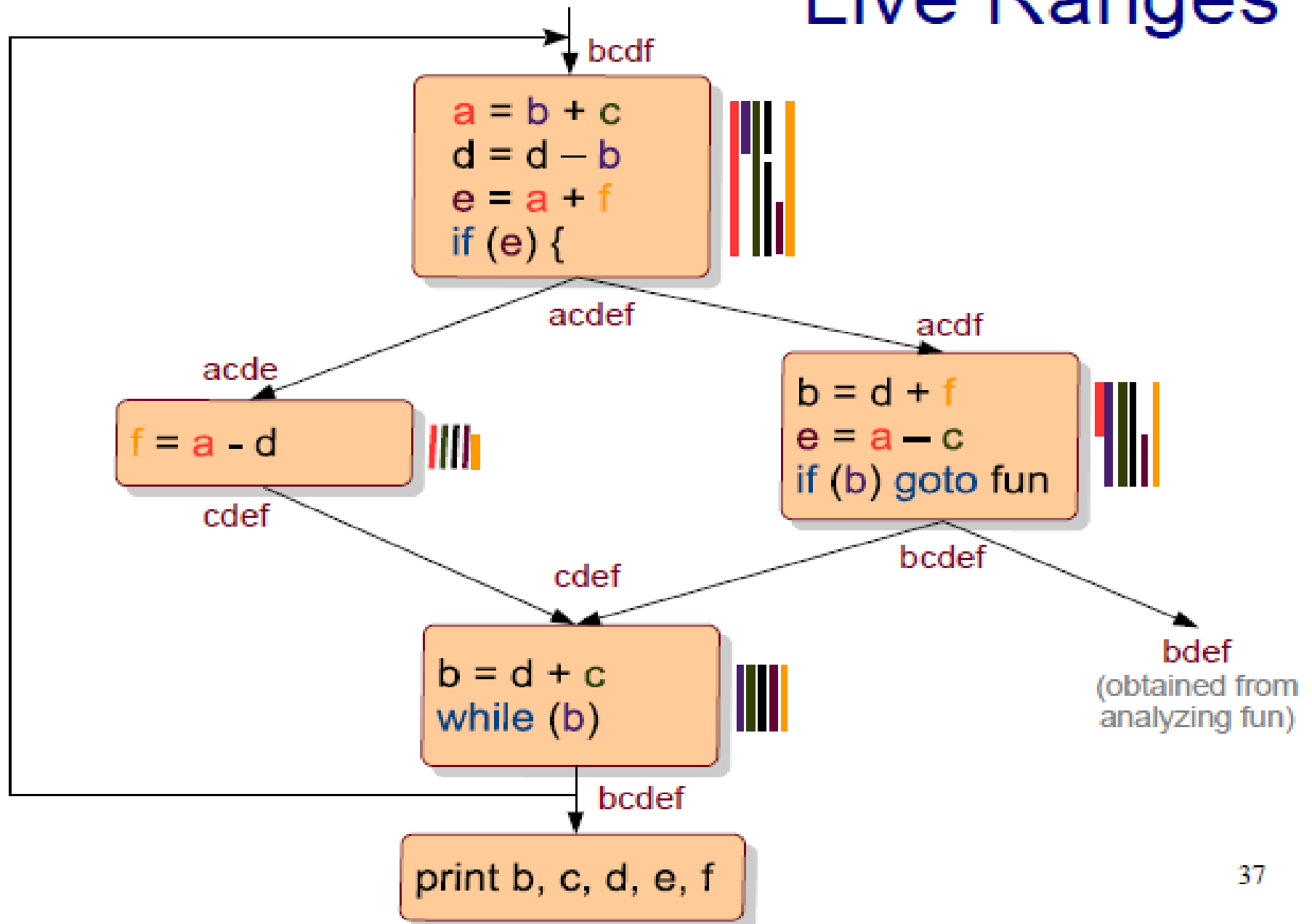
Liveness

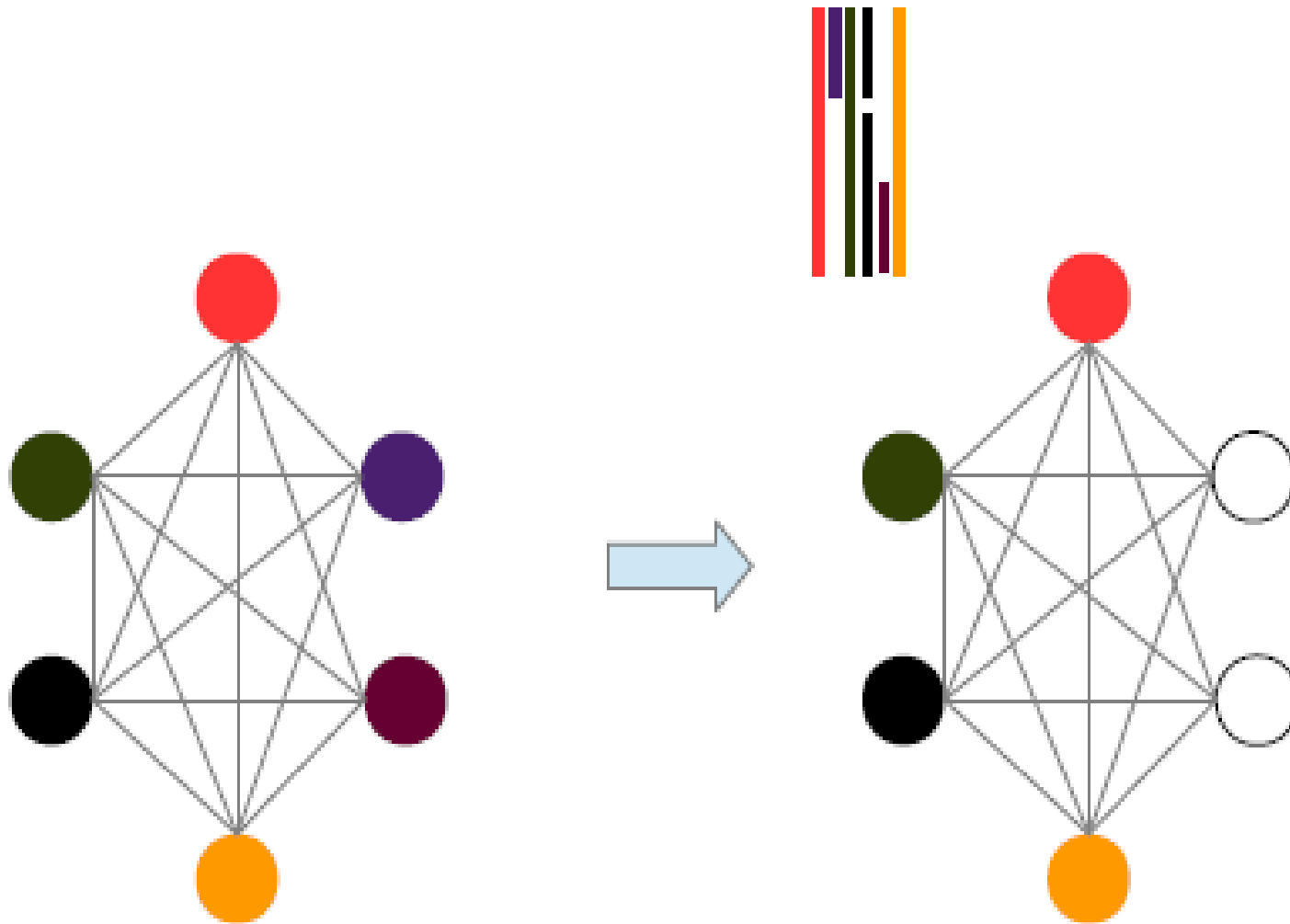


Register Allocation as Graph Coloring

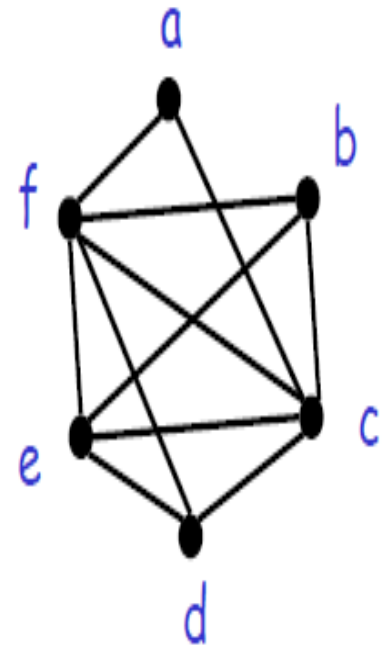
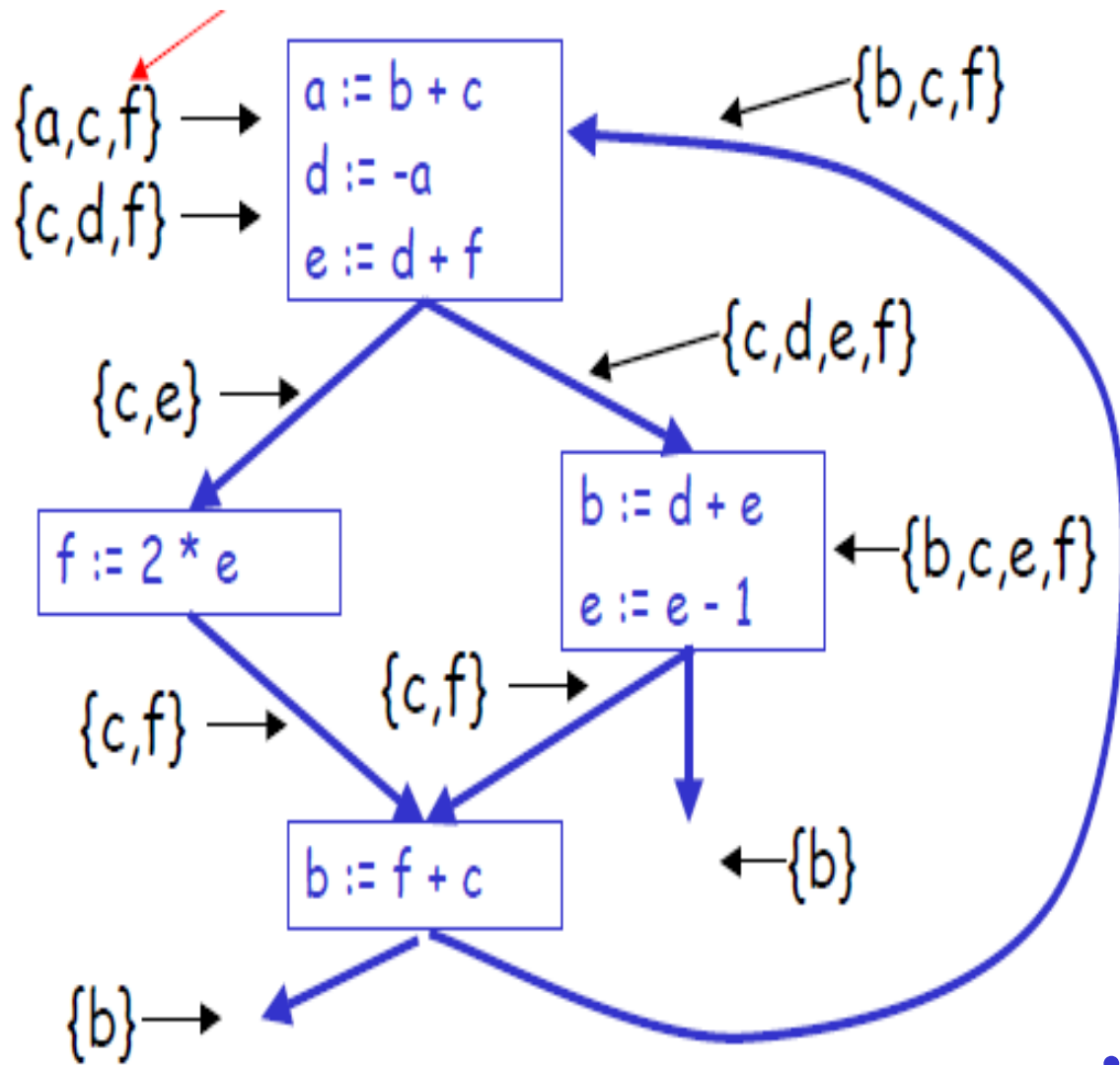
- Vertices: Variables (or their instances)
- Edges: Co-Live information
 - If x and y are live at the same program point, add an (undirected) edge between x and y .
- Vertex coloring colors neighbors differently.
 - Thus, vertex coloring colors x and y differently, if they are live at the same program point.
 - This means, x and y should not use the same register.
 - **Corollary:** if x and z have the same color, they can reuse the register (at different program points).

Live Ranges





This means, in basic block B1, **b** and **e** could use the same register.



Register Interference Graph (RIG)

- b and c cannot be in the same register
- b and d can be in the same register

Graph Coloring and Register Spill

- Coloring gave us the maximum number of registers required for the program.
- However, in practice, the number of registers is fixed.
- Therefore, we need to generate spill code for storing a variable into memory
 - (ST x, R) and then reload the register with the next variable (LD R, y)