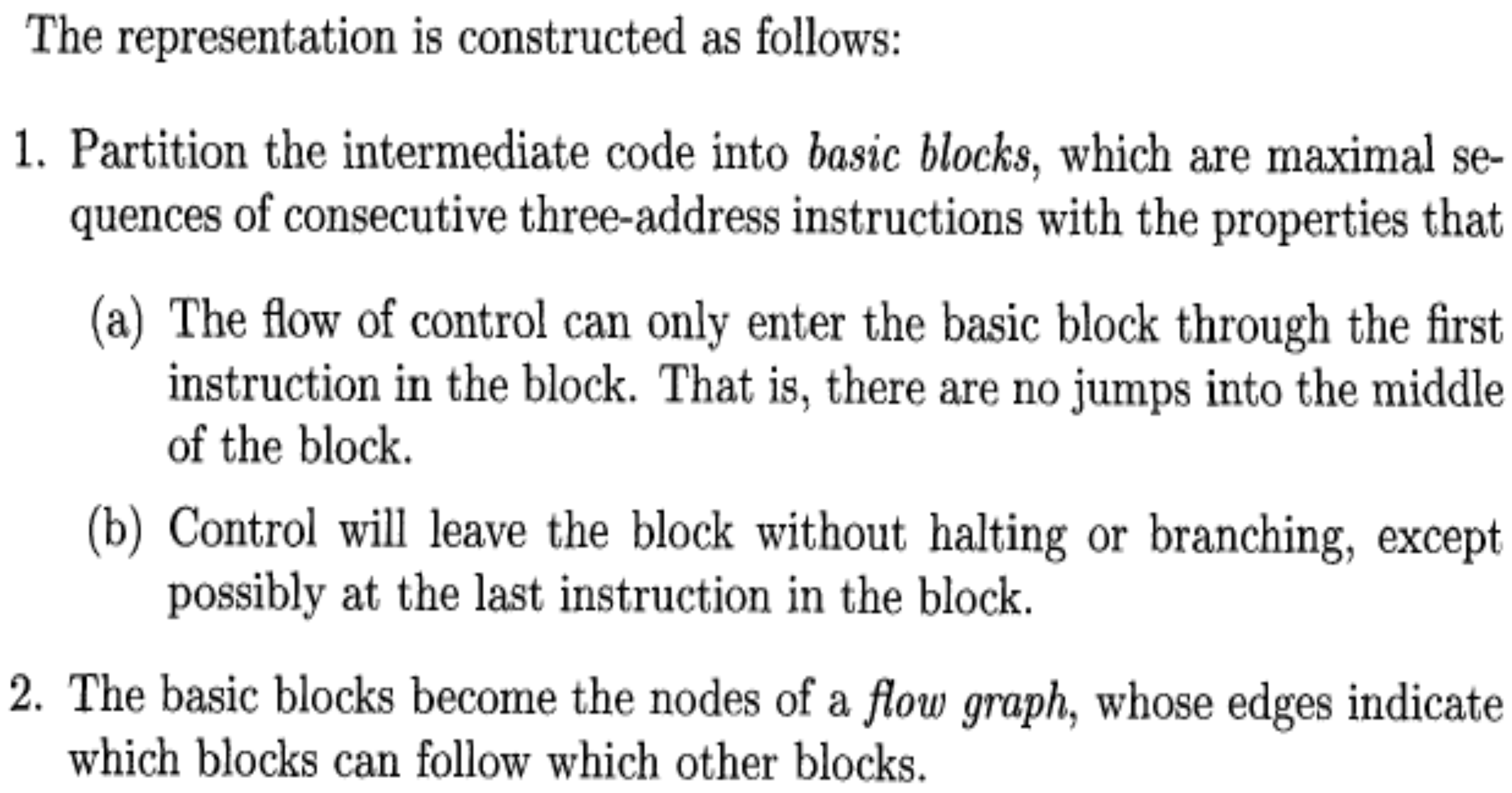
COMPILER CONSTRUCTION
CS F363

BITS Pilani, Dubai Campus

Basic Blocks & Flow Graphs



The representation is constructed as follows:

1. Partition the intermediate code into *basic blocks*, which are maximal sequences of consecutive three-address instructions with the properties that

   (a) 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.

   (b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.

2. The basic blocks become the nodes of a *flow graph*, whose edges indicate which blocks can follow which other blocks.

Basic Blocks & Flow Graphs

Algorithm 8.5: Partitioning three-address instructions into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

1. The first three-address instruction in the intermediate code is a leader.

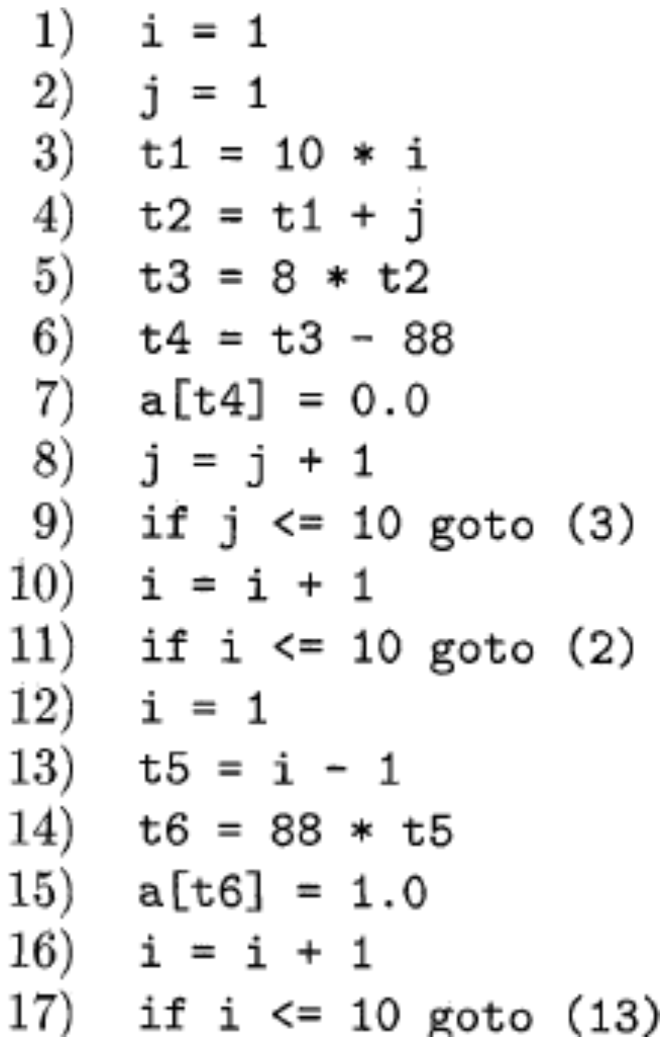
2. Any instruction that is the target of a conditional or unconditional jump is a leader.

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

BITS Pilani
Dubai Campus

innovate achieve lead

Sample Intermediate Code



```
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)
```

Leaders: 1, 2, 3, 10, 12, 13

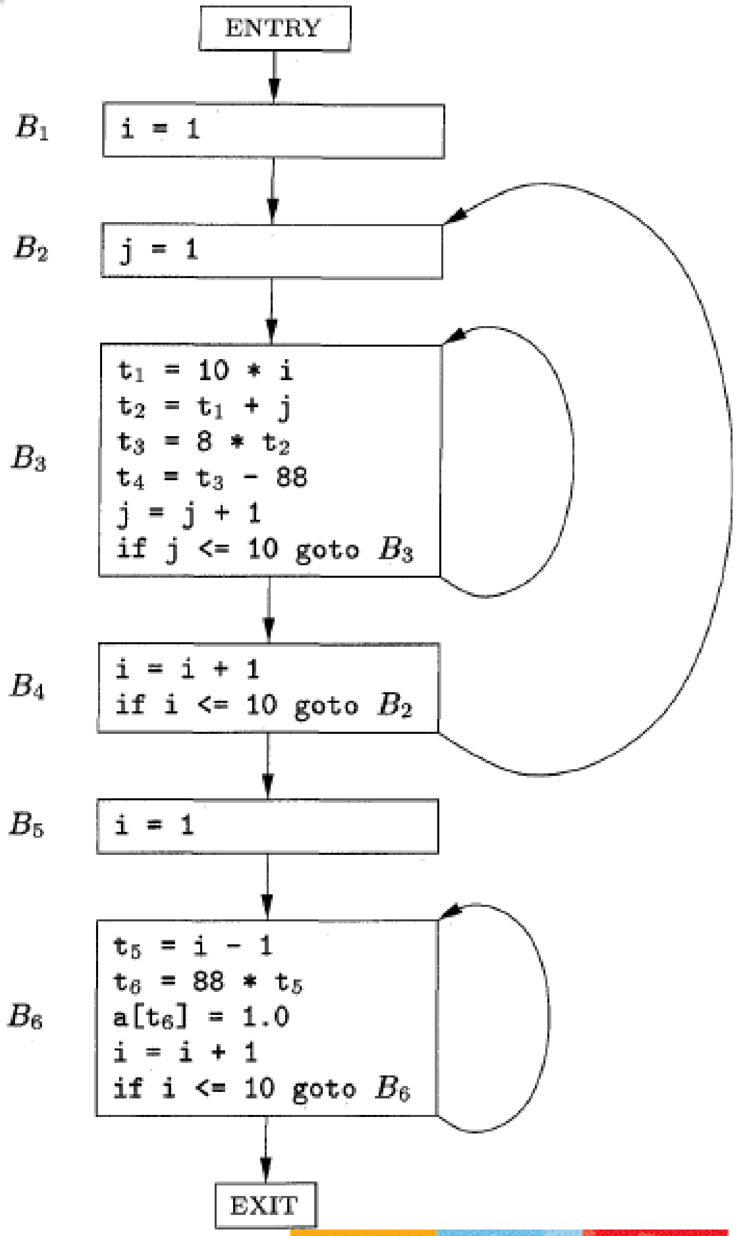
B1: 1
B2: 2
B3: 3-9
B4: 10, 11
B5: 12
B6: 13 - 17

Flow Graphs

```
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)
```

Leaders: 1, 2, 3, 10, 12, 13

B1: 1
B2: 2
B3: 3-9
B4: 10, 11
B5: 12
B6: 13 - 17



Simple Code Generator

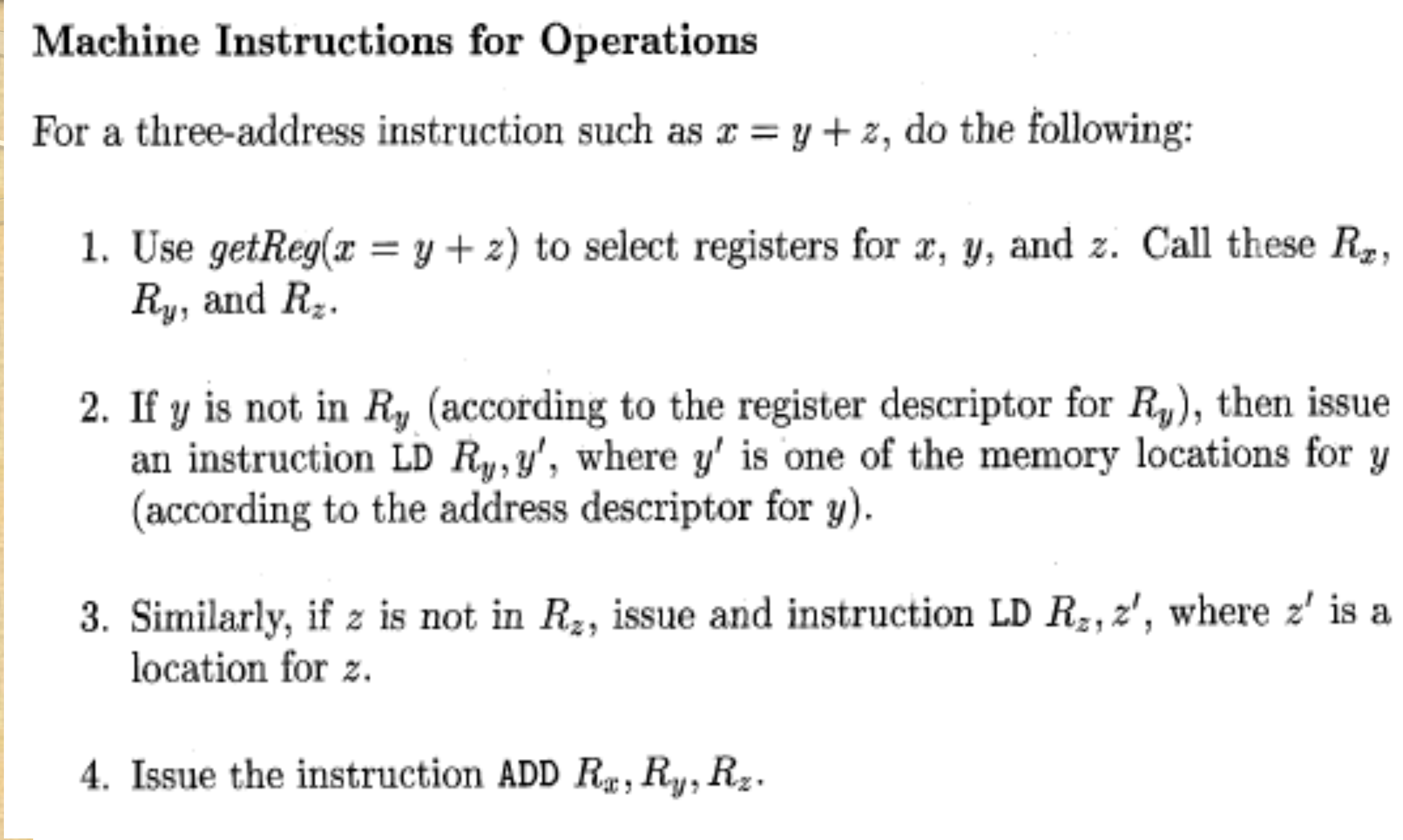
- A simple code generator
- Input: A three-address code basic block with possible temporaries local to the block

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

- t,u,v are temporaries local to the block
- a, b, c, d are program variables

BITS Pilani
Dubai Campus

Simple Code Generator



Machine Instructions for Operations

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 and instruction `LD` R_z, z', where z' is a location for z.
4. Issue the instruction `ADD` R_x, R_y, R_z.

Simple Code Generator

Machine Instructions for Copy Statements (x = y)

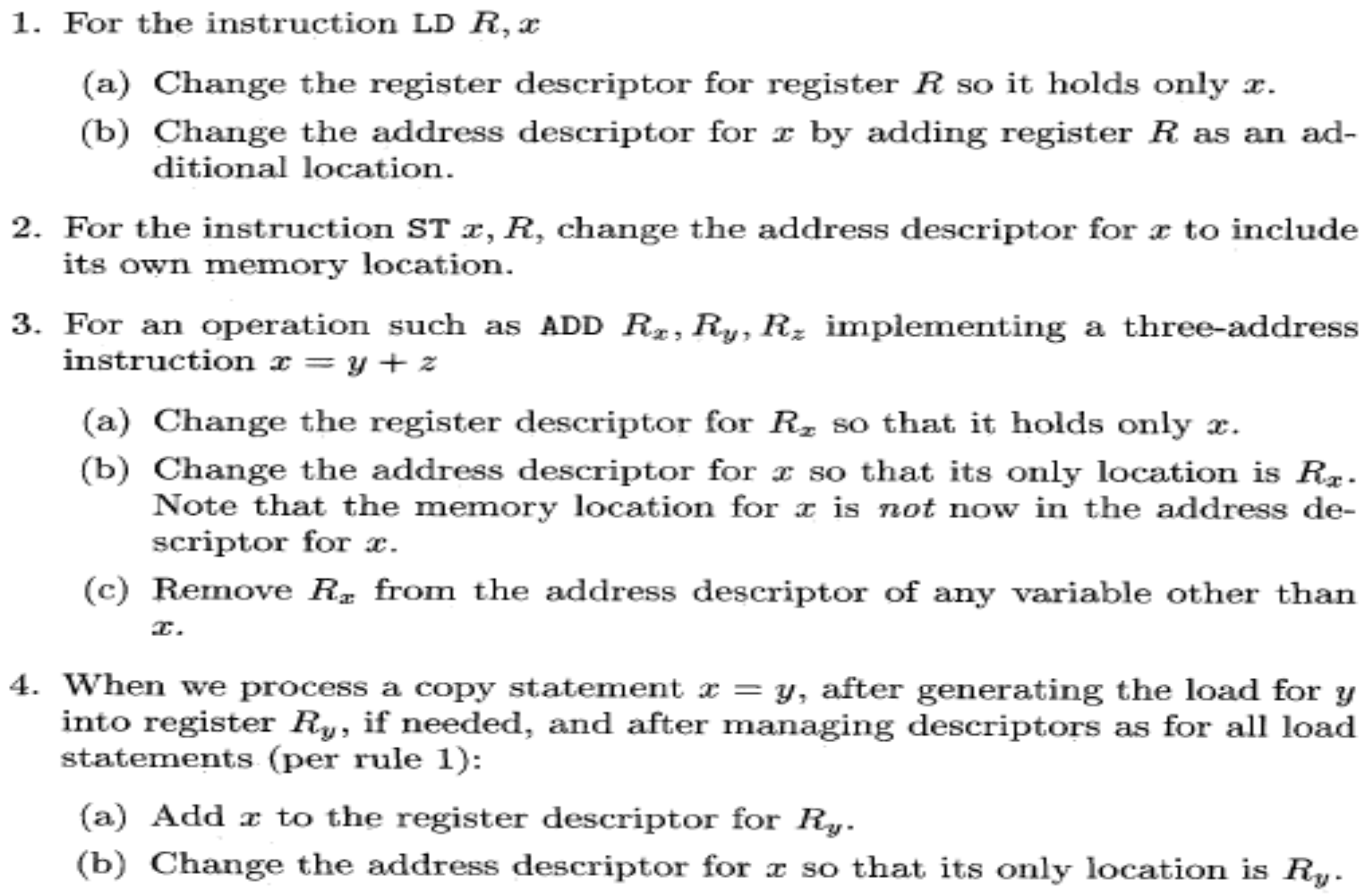
- We assume that getReg will always choose the same register for both x and y.
- If y is not already in that register Ry, then generate the machine instruction LD Ry, y
- If y was already in Ry, we do nothing.

Simple Code Generator

Ending the Basic Block

- If the variable is a temporary used only within the block, when the block ends, we can forget about the value of the temporary and assume its register is empty.
- However, if the variable is live on exit from the block, or if we don't know which variables are live on exit, then we need to assume that the value of the variable is needed later.
- In that case, for each variable x whose address descriptor does not say that its value is located in the memory location for x, we must generate the instruction ST x,R where R is a register in which x's value exists at the end of the block.

Updating the Data Structures



1. 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.

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

3. For an operation such as ADD R_x, R_y, R_z 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.

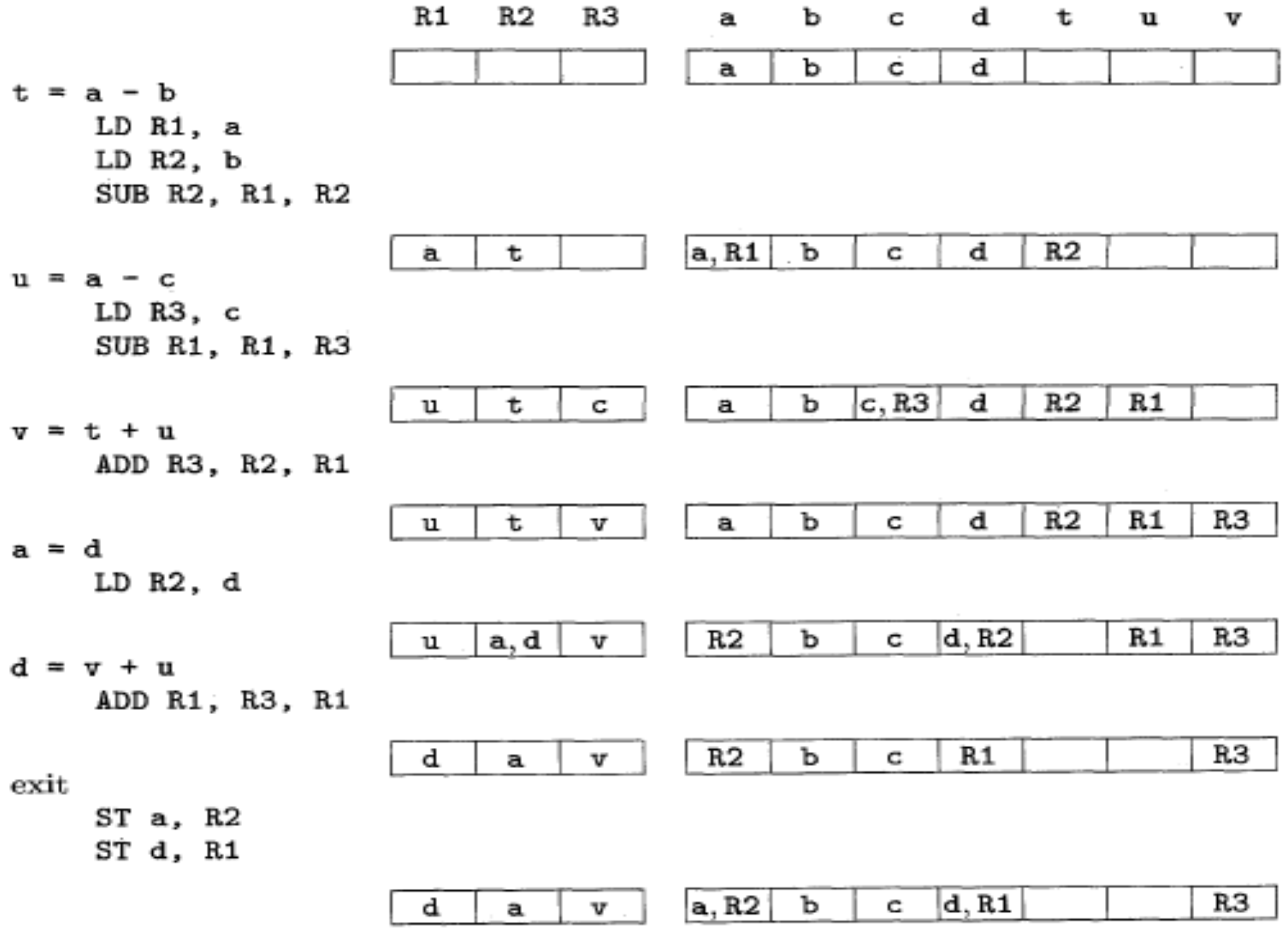
4. When we process a copy statement $x = y$, after generating the load for y into register R_y, if needed, and after managing descriptors as for all load statements (per rule 1):

   (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.

Simple Code generator- example

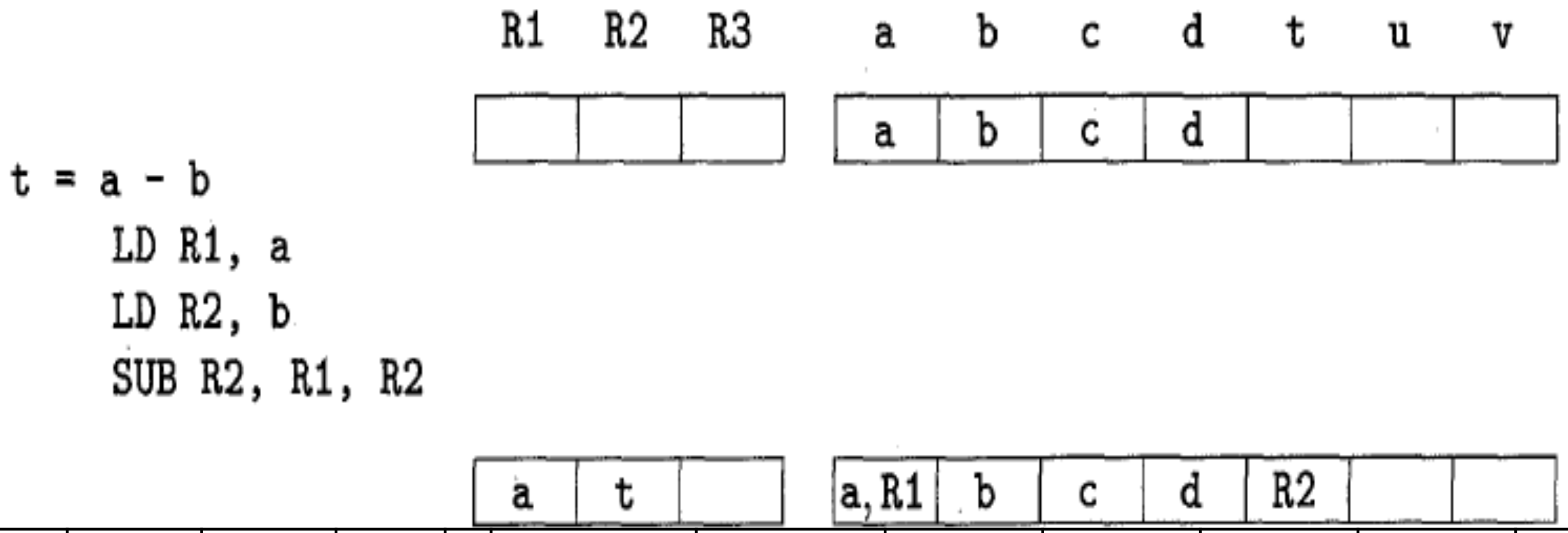
```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```



```
t = a - b
        LD R1, a
        LD R2, b
        SUB R2, R1, R2
u = a - c
        LD R3, c
        SUB R1, R1, R3
v = t + u
        ADD R3, R2, R1
a = d
        LD R2, d
d = v + u
        ADD R1, R3, R1
exit
        ST a, R2
        ST d, R1
```

	R1	R2	R3	a	b	c	d	t	u	v
				a	b	c	d			
t = a - b	a	t		a, R1	b	c	d	R2		
u = a - c	u	t	c	a	b	c, R3	d	R2	R1	
v = t + u	u	t	v	a	b	c	d	R2	R1	R3
a = d	u	a, d	v	R2	b	c	d, R2		R1	R3
d = v + u	d	a	v	R2	b	c	R1			R3
exit	d	a	v	a, R2	b	c	d, R1			R3

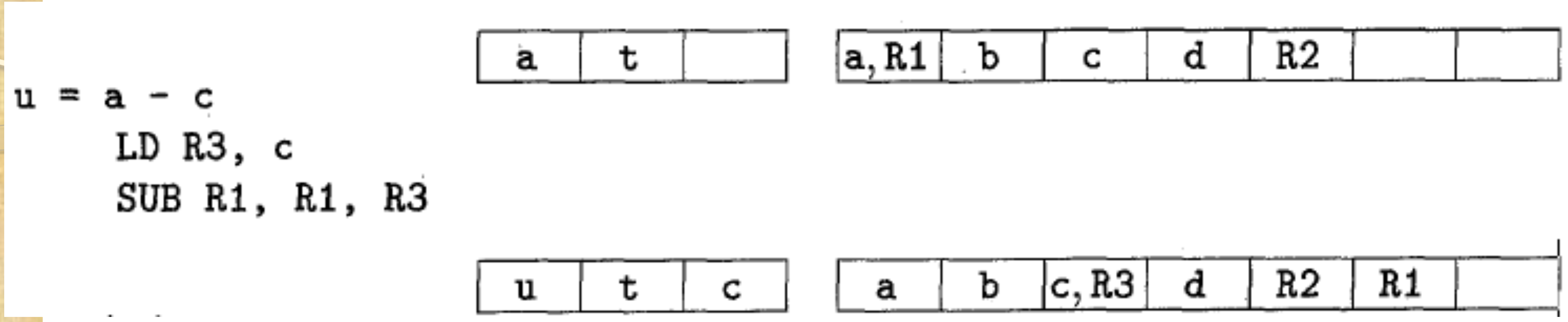
Example-Steps



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

	R1	R2	R3		a	b	c	d	t	u	v
LD R1, a	a				a, R1	b	c	d			
LD R2, b	a	b			a, R1	b, R2	c	d			
SUB R2, R1, R2	a	t			a, R1	b	c	d	R2		

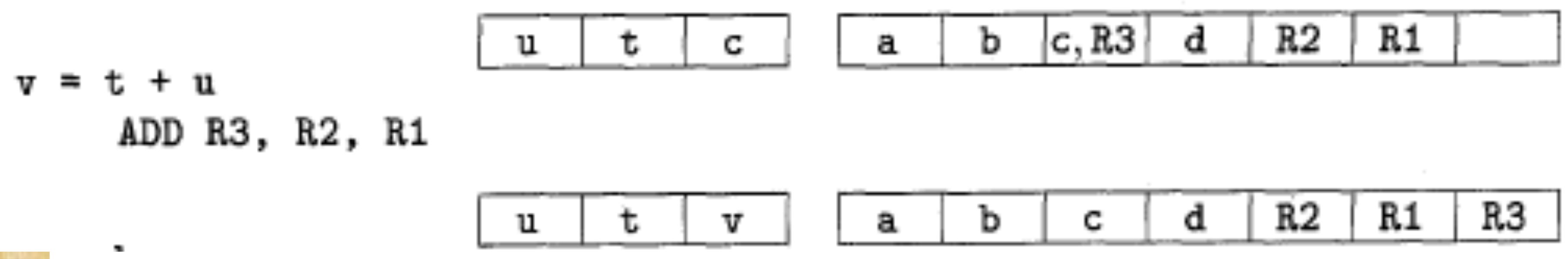
Example-Steps



```
u = a - c
      LD R3, c
      SUB R1, R1, R3
```

	R1	R2	R3		a	b	c	d	t	u	v
Previous	a	t			a, R1	b	c	d	R2		
LD R3, **c**	a	t	c		a, R1	b	c, R3	d	R2		
SUB RI, R1, R3	u	t	C		a	b	c,R3	d	R2	R1	

Example-Steps

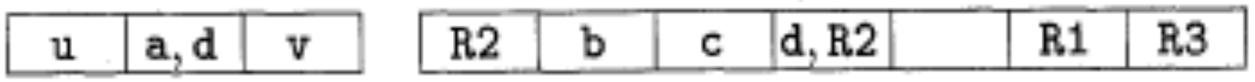


```
v = t + u
    ADD R3, R2, R1
```

	R1	R2	R3		a	b	c	d	t	u	v
Previous	u	t	c		a	b	c,R3	d	R2	R1	
ADD R3 , R2 , R1	u	t	v		a	b	c	d	R2	R1	R3

Example-Steps

```
a = d
    LD R2, d
```

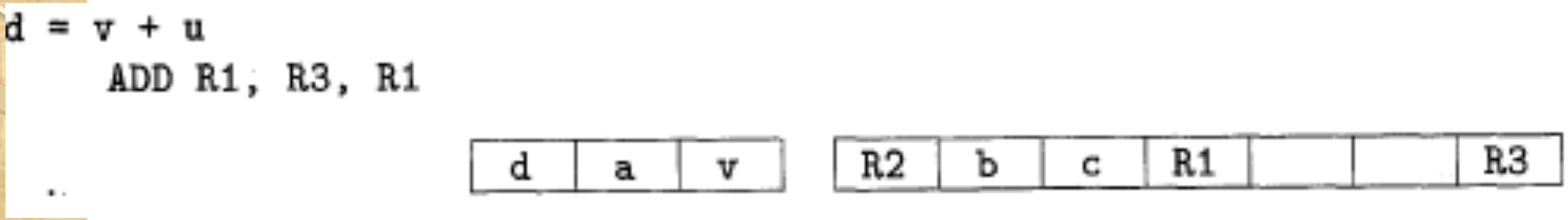


u	a, d	v

R2	b	c	d, R2		R1	R3

	R1	R2	R3		a	b	c	d	t	u	v
Previous	u	t	v		a	b	c	d	R2	R1	R3
LD R2, d	u	d, a	v		R2	b	c	d,R2		R1	R3

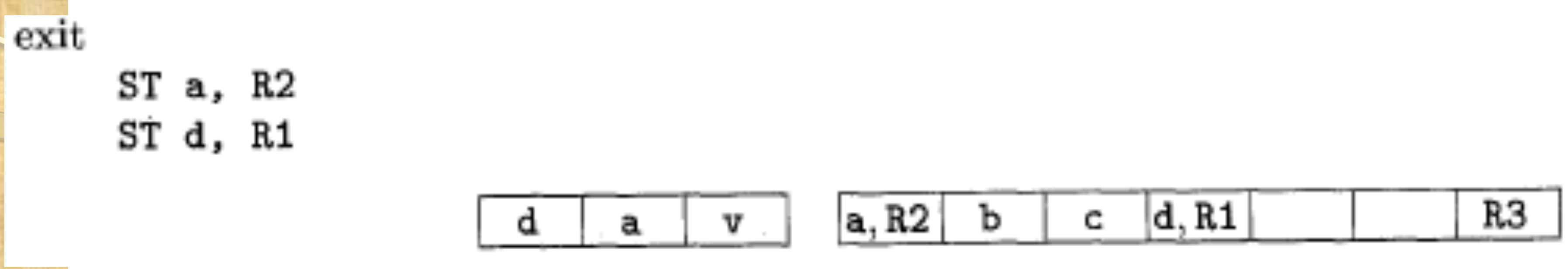
Example-Steps



```
d = v + u
    ADD R1, R3, R1
..
```

	R1	R2	R3		a	b	c	d	t	u	v
Previous	u	d, a	v		R2	b	c	d,R2		R1	R3
ADD R1, R3, R1	d	a	v		R2	b	c	R1			R3

Example-Steps



```
exit
        ST a, R2
        ST d, R1
```

d	a	v

a, R2	b	c	d, R1			R3

	R1	R2	R3		a	b	c	d	t	u	v
Previous	d	a	v		R2	b	c	R1			R3
ST a,R2	d	a	v		a, R2	b	c	R1			R3
ST d, R1	d	a	v		a,R2	b	c	d,R1			R3