Code Generation

SINI ANNA ALEX

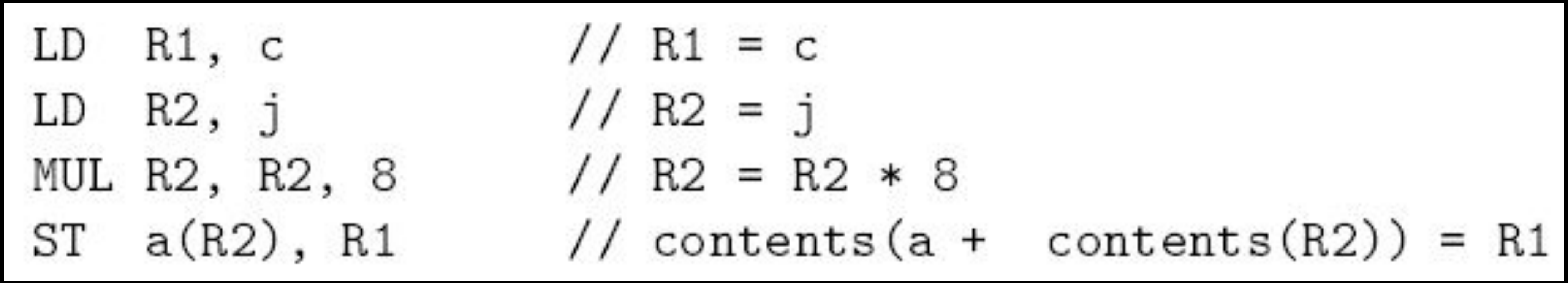
5/8/2020 9:17 AM

Machine Instructions for `b = a[i]`

```
LD  R1, i            // R1 = i
MUL R1, R1, 8        // R1 = R1 * 8
LD  R2, a(R1)        // R2 = contents(a + contents(R1))
ST  b, R2            // b = R2
```

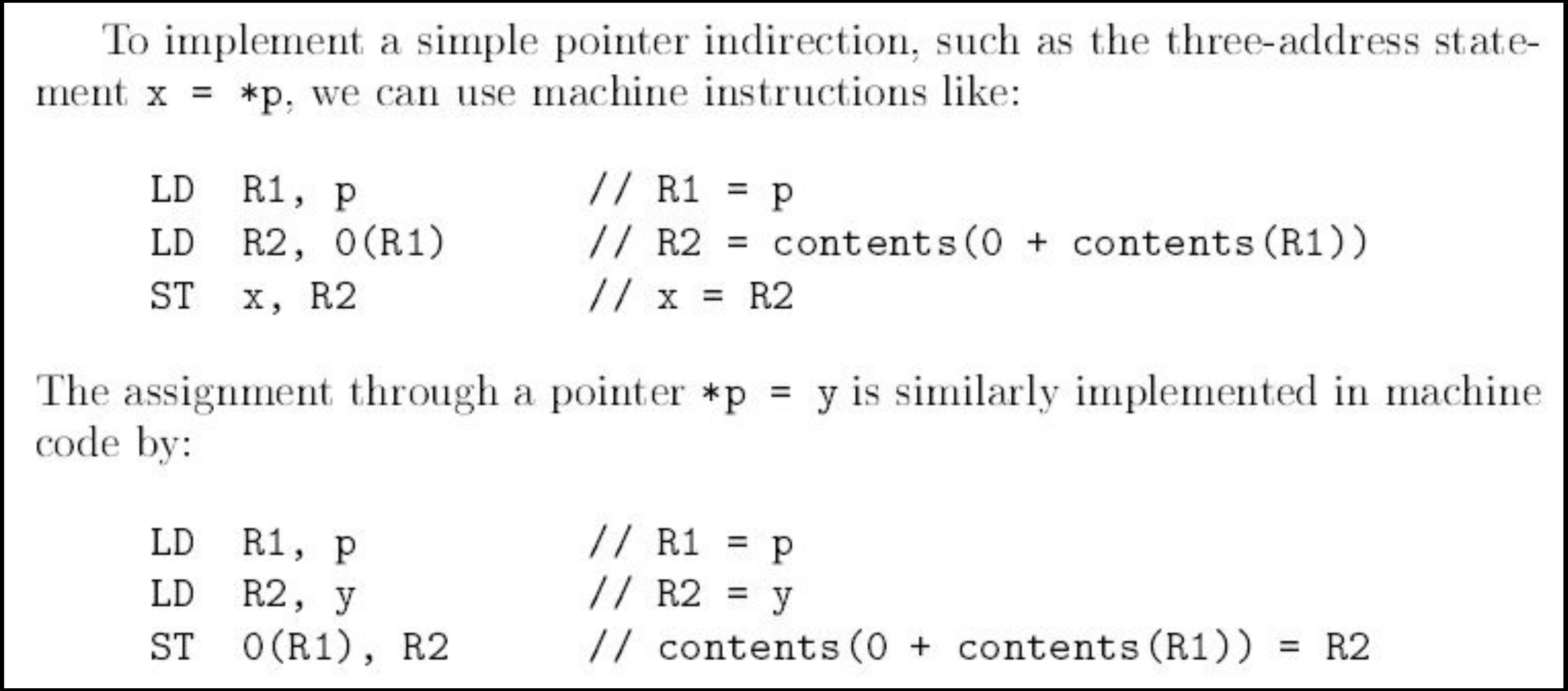
Assume elements of a are indexed starting at 0

Machine Instructions for `a[j] = c`



```
LD  R1, c          // R1 = c
LD  R2, j          // R2 = j
MUL R2, R2, 8      // R2 = R2 * 8
ST  a(R2), R1      // contents(a +  contents(R2)) = R1
```

Generate Code for the three address statements



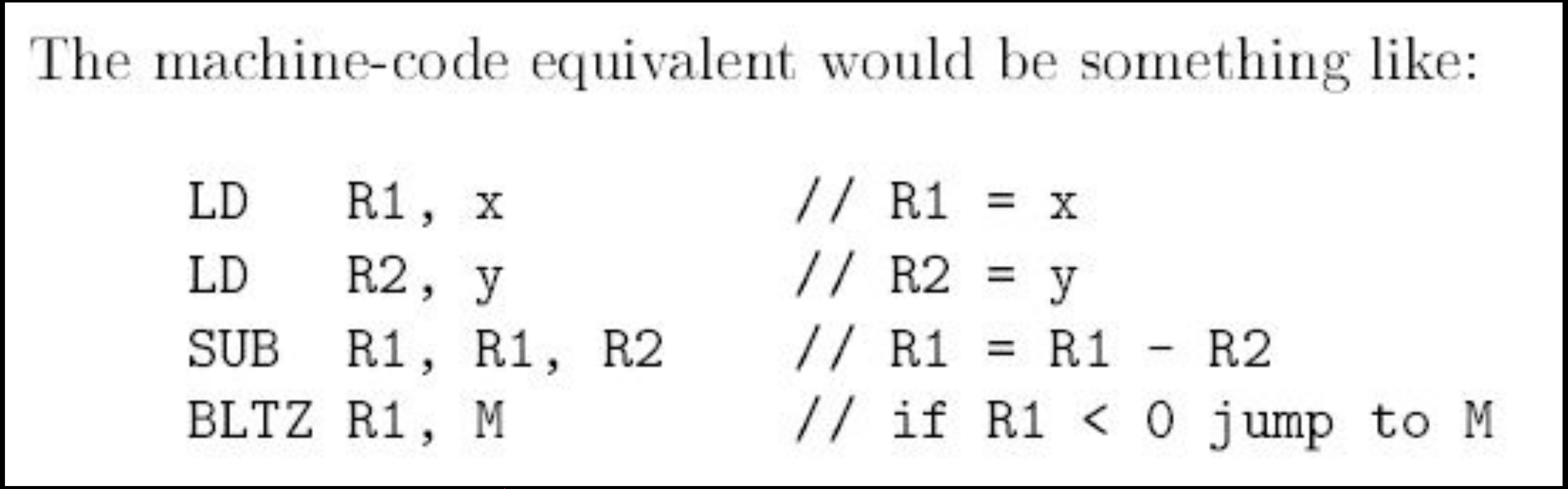
To implement a simple pointer indirection, such as the three-address statement `x = *p`, we can use machine instructions like:

```
LD  R1, p          // R1 = p
LD  R2, 0(R1)      // R2 = contents(0 + contents(R1))
ST  x, R2          // x = R2
```

The assignment through a pointer `*p = y` is similarly implemented in machine code by:

```
LD  R1, p          // R1 = p
LD  R2, y          // R2 = y
ST  0(R1), R2      // contents(0 + contents(R1)) = R2
```

Machine code for a conditional jump instruction like `if x < y goto L`

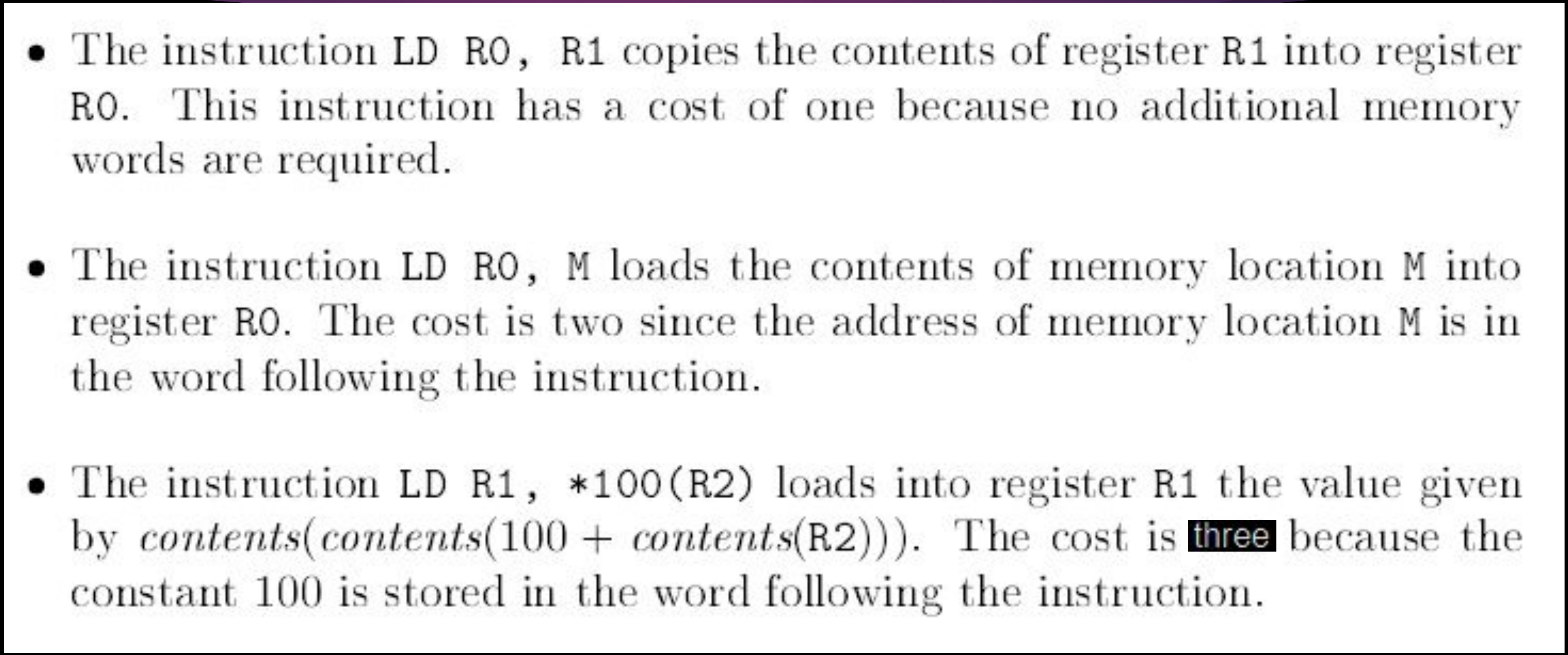


The machine-code equivalent would be something like:

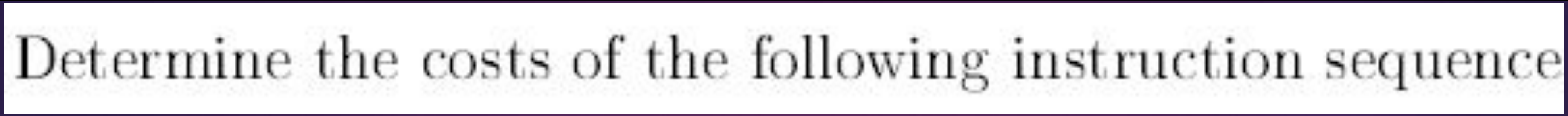
```
LD   R1, x          // R1 = x
LD   R2, y          // R2 = y
SUB  R1, R1, R2     // R1 = R1 - R2
BLTZ R1, M          // if R1 < 0 jump to M
```

Here M means the label L

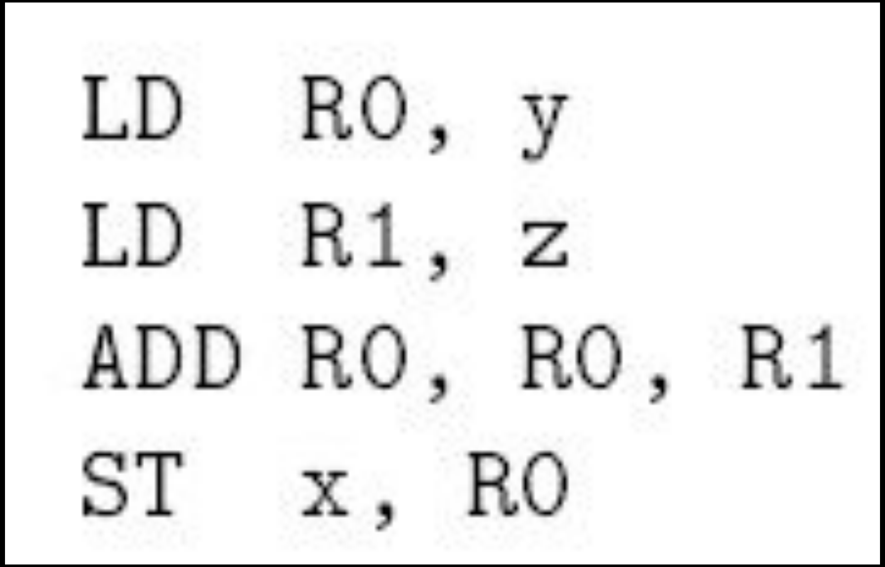
Program and Instruction Cost



- The instruction `LD R0, R1` copies the contents of register `R1` into register `R0`. This instruction has a cost of one because no additional memory words are required.
- The instruction `LD R0, M` loads the contents of memory location `M` into register `R0`. The cost is two since the address of memory location `M` is in the word following the instruction.
- The instruction `LD R1, *100(R2)` loads into register `R1` the value given by *contents*(*contents*(100 + *contents*(`R2`))). The cost is three because the constant 100 is stored in the word following the instruction.



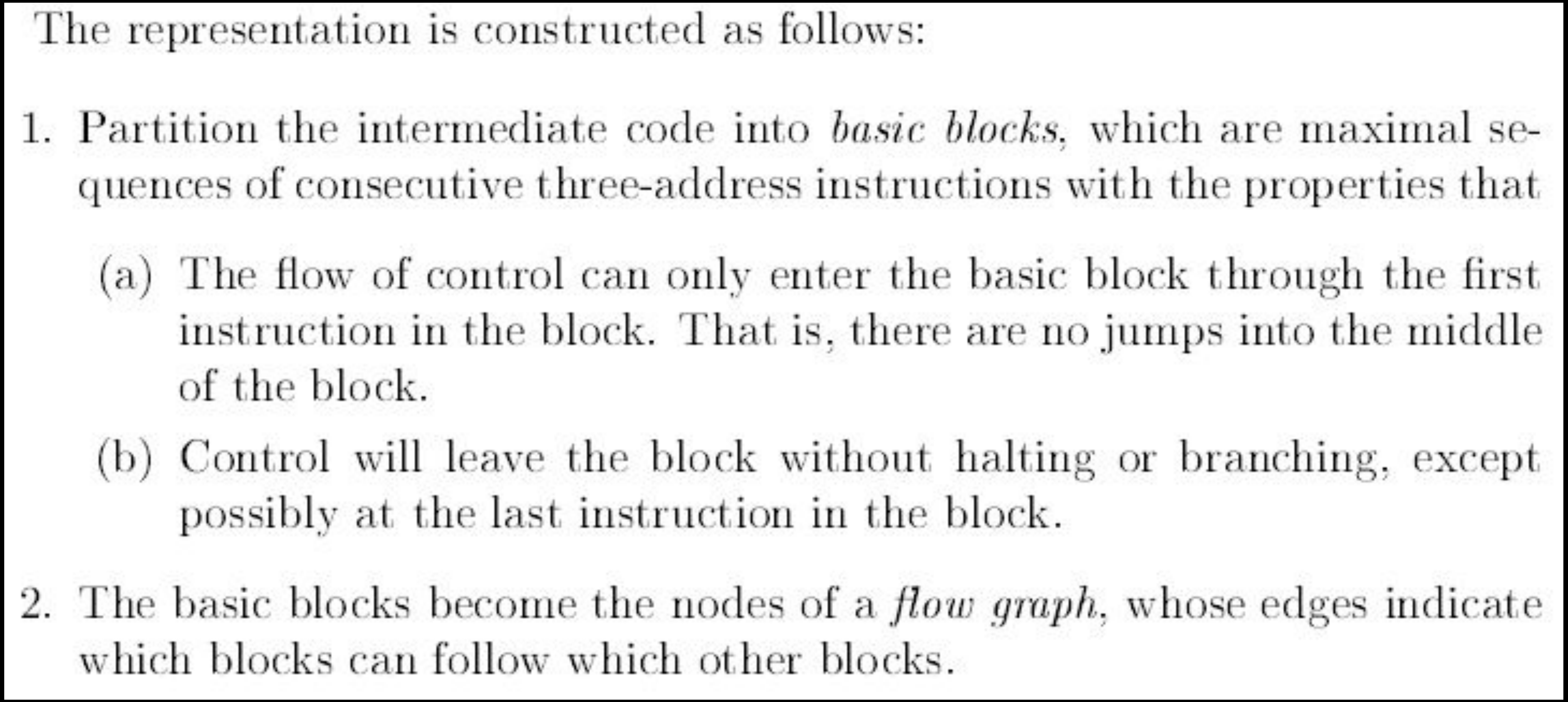
Determine the costs of the following instruction sequence



```
LD  R0, y
LD  R1, z
ADD R0, R0, R1
ST  x, R0
```

Cost :
2+2+1+2 = 7

Basic Blocks and 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

Algorithm : Partitioning three-address instructions into basic blocks.

INPUT: A sequence of three-address instructions.

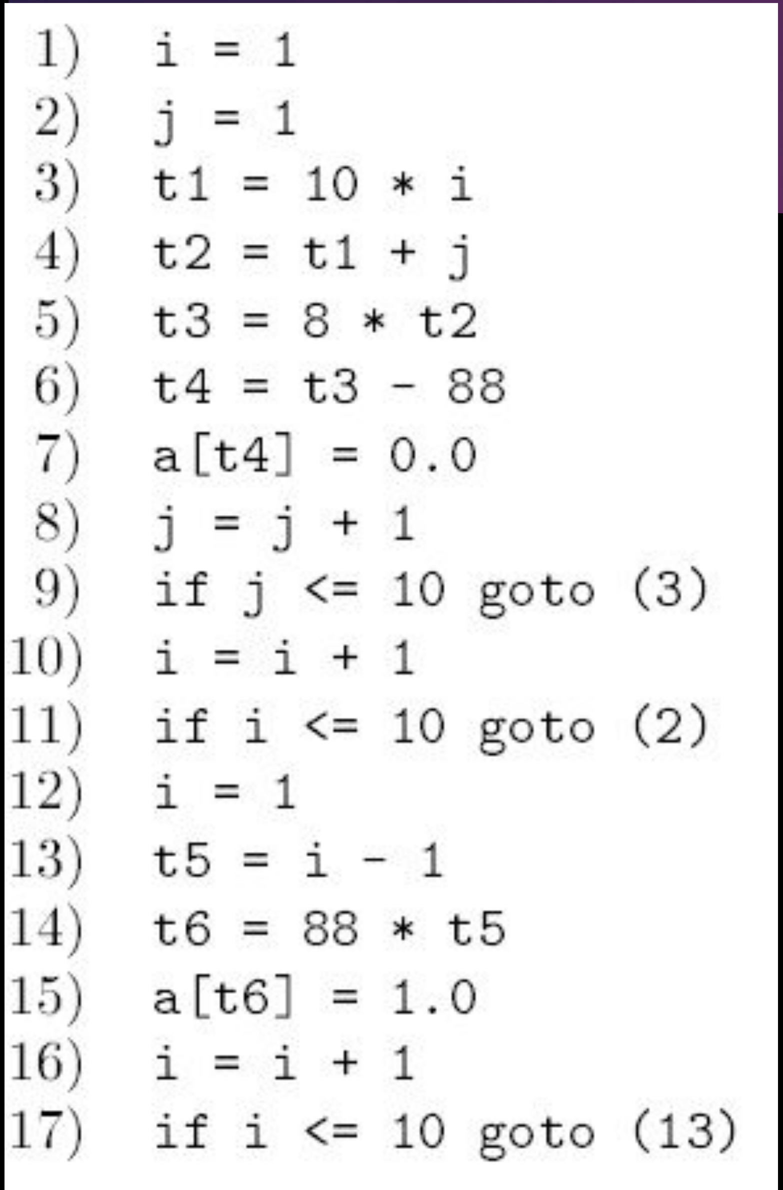
OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are *leaders*, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

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.

Finding Leaders

Intermediate code to set a 10 × 10 matrix to an identity matrix



```
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:

•First, instruction 1 is a leader by rule (1) of Algorithm.

• To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11, and 17.

•By rule (2), the targets of these jumps are leaders; they are instructions 3, 2, and 13, respectively.

•Then, by rule (3), each instruction following a jump is a leader; those are instructions 10 and 12.

•No instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

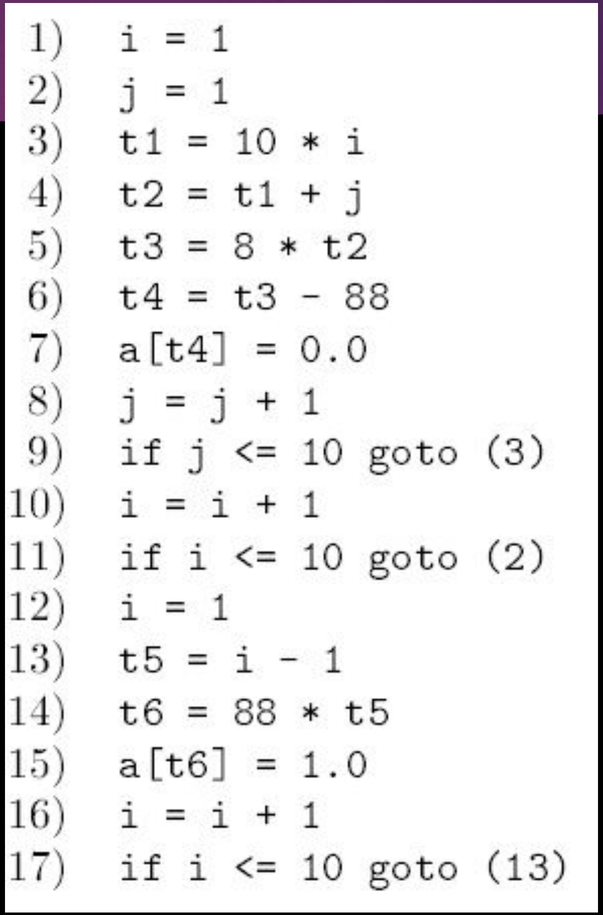
Leaders are instructions 1, 2, 3, 10, 12, and 13.

Partitioning into Basic Blocks

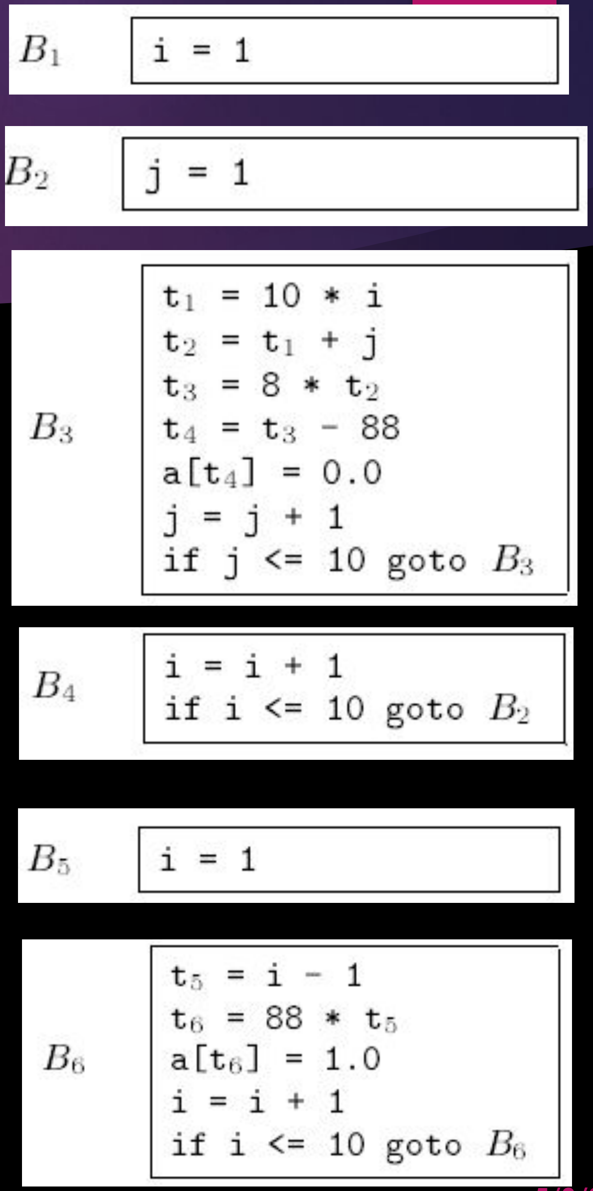
Leaders:

- First, instruction 1 is a leader by rule (1) of Algorithm.
- To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11, and 17.
- By rule (2), the targets of these jumps are leaders; they are instructions 3, 2, and 13, respectively.
- By rule (3), each instruction following a jump is a leader; those are instructions 10 and 12.
- No instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

Leaders are instructions 1, 2, 3, 10, 12, and 13.



```
 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)
```



B_1	i = 1
B_2	j = 1
B_3	t_1 = 10 * i t_2 = t_1 + j t_3 = 8 * t_2 t_4 = t_3 - 88 a[t_4] = 0.0 j = j + 1 if j <= 10 goto B_3

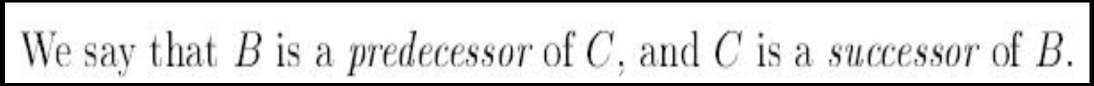
B_4	i = i + 1
	if i <= 10 goto B_2

B_5	i = 1

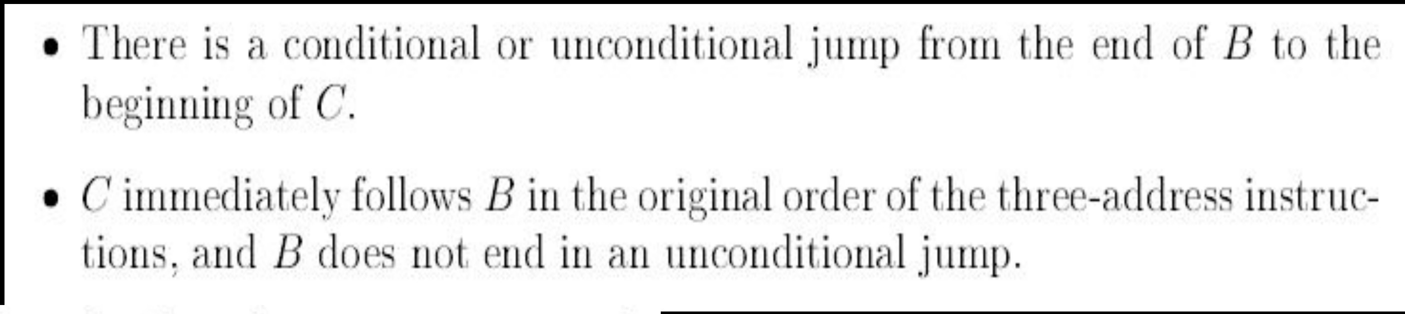
B_6	t_5 = i - 1
	t_6 = 88 * t_5
	a[t_6] = 1.0
	i = i + 1
	if i <= 10 goto B_6

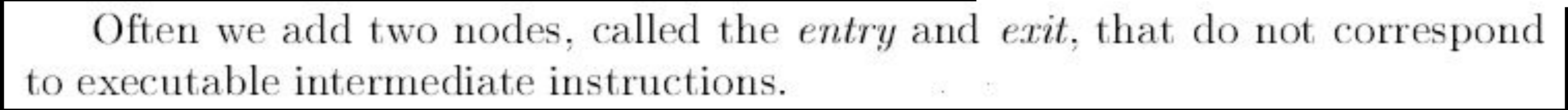
Flow Graphs

Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. **The nodes of the flow graph are the basic blocks**.


We say that B is a *predecessor* of C, and C is a *successor* of B.

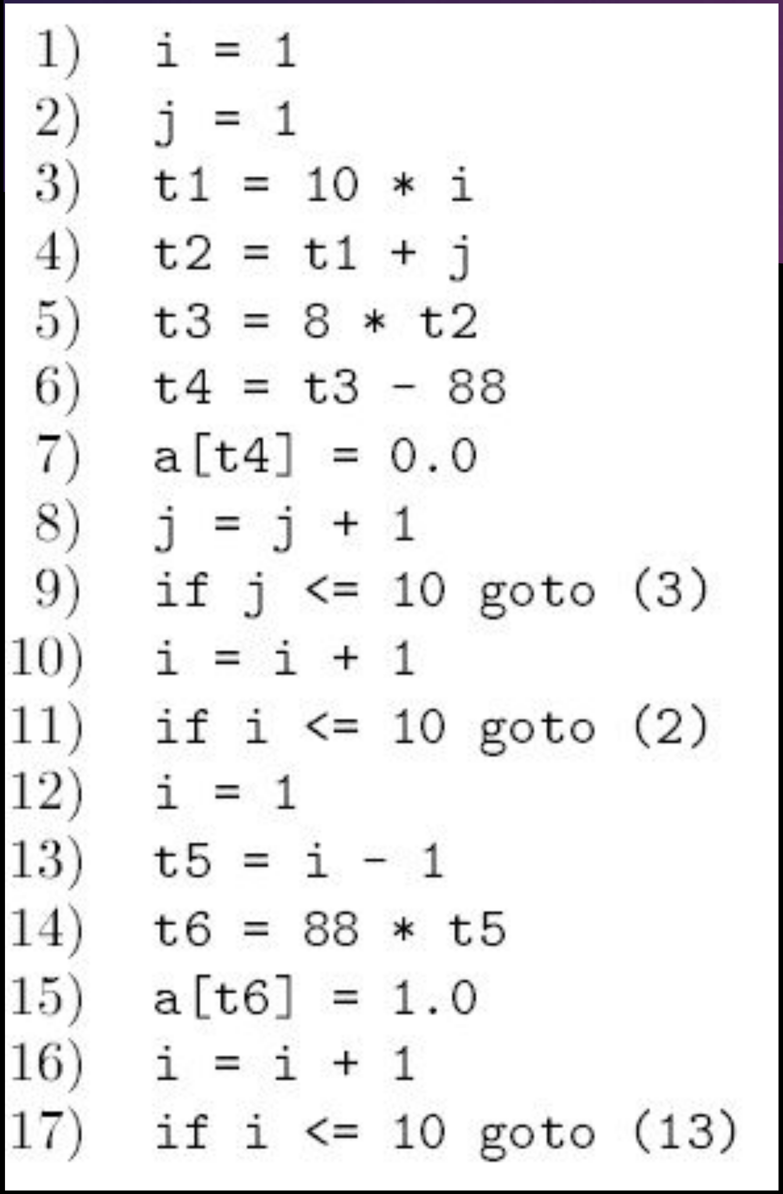
There are two ways that such an edge could be justified:


- There is a conditional or unconditional jump from the end of B to the beginning of C.
- C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

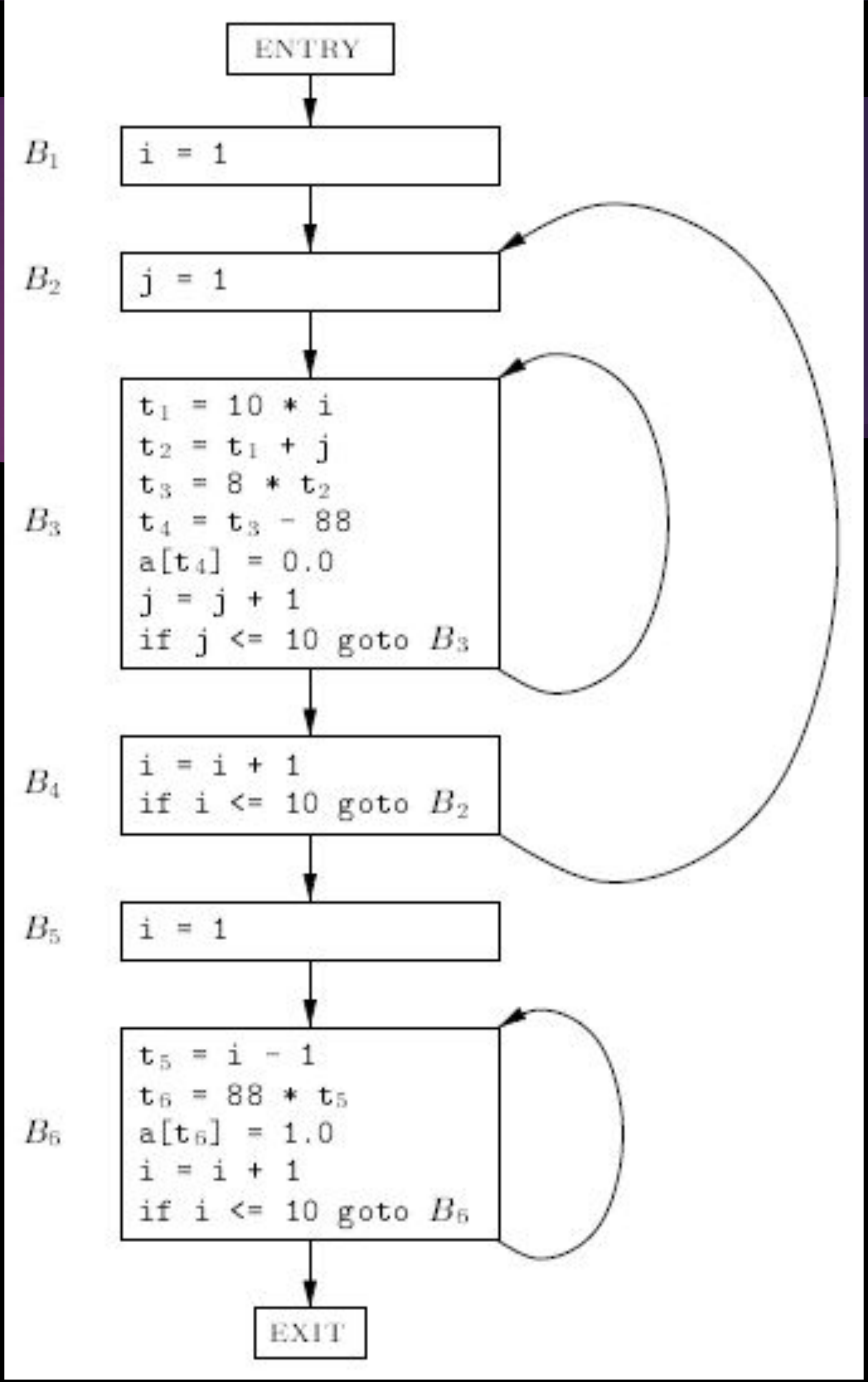

Often we add two nodes, called the *entry* and *exit*, that do not correspond to executable intermediate instructions.

- There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code.
- There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program.
- If the final instruction of the program is not an unconditional jump, then the block containing the final instruction of the program is one predecessor of the exit.

Flow Graph



```
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)
```



```
B1: i = 1
B2: j = 1
B3: t1 = 10 * i
    t2 = t1 + j
    t3 = 8 * t2
    t4 = t3 - 88
    a[t4] = 0.0
    j = j + 1
    if j <= 10 goto B3
B4: i = i + 1
    if i <= 10 goto B2
B5: i = 1
B6: t5 = i - 1
    t6 = 88 * t5
    a[t6] = 1.0
    i = i + 1
    if i <= 10 goto B6
```