Basic Blocks, Flow Graphs, Next-use information

Flow Graphs

- A *flow graph* is a graphical depiction of a sequence of instructions
- A flow graph can be defined at the intermediate code level or target code level

Example

```
     MOV 1,R0
     MOV n,R1
     JMP L2
L1:  MUL 2,R0
     SUB 1,R1
L2:  JMPNZ R1,L1
```

```
     MOV 0,R0
     MOV n,R1
     JMP L2
L1:  MUL 2,R0
     SUB 1,R1
L2:  JMPNZ R1,L1
```

Basic Blocks

- A *basic block* is a sequence of consecutive instructions with exactly one entry point and one exit point (with natural flow or a branch instruction)

Basic blocks and flow graphs

- Graph representation of 3-address statement – flow graph
- Nodes in the graph – Computations
- Edges in the graph – Flow of control
- Useful for optimization, register allocation

Basic Blocks & Control Flow Graph

- A *control flow graph* (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \rightarrow B_j$ iff B_j can be executed immediately after B_i

Partition Algorithm for Basic Blocks

Input: A sequence of three-address statements
Output: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements if basic blocks
   a) The first statement is the leader
   b) Any statement that is the target of a goto is a leader
   c) Any statement that immediately follows a goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Example

leader

```
      MOV 1,R0
      MOV n,R1
      JMP L2
L1:   MUL 2,R0
      SUB 1,R1
L2:   JMPNZ R1,L1
```

leader (L1)

leader (L2)

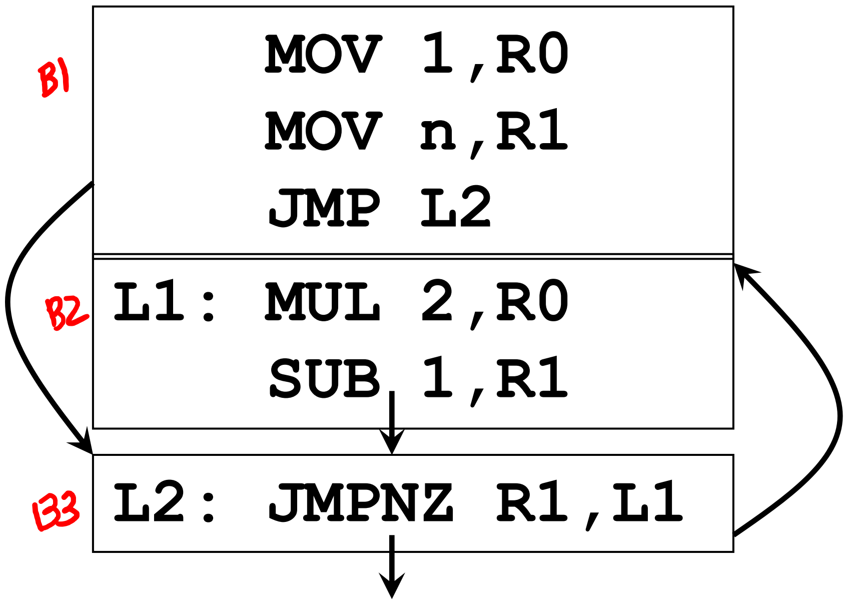
B1
```
      MOV 1,R0
      MOV n,R1
      JMP L2
```

B2
```
L1:   MUL 2,R0
      SUB 1,R1
```

B3
```
L2:   JMPNZ R1,L1
```

Successor and Predecessor Blocks

- Suppose the flow graph has an edge $B_1 \rightarrow B_2$
- B_1 is a *predecessor* of B_2 and B_2 is a *successor* of B_1



Example

```
Begin
        prod := 0
        i := 1;
  do begin
        prod : = prod + a[i] * b[i];
        i = i +1;
        end
  while i < = 20
end
```

```
1.  prod := 0
2.  i := 1
3.  t1 := 4 * i
4.  t2 := a[t1]
5.  t3 := 4 * i
6.  t4 := b [t3]
7.  t5 := t2 *t4
8.  t6 := prod + t5
9.  prod := t6
10. t7 := i+1
11. i := t7
12. If i <= 20 goto (3)
```

Identifying leaders

- (1) is the beginning – hence leader
- (3) is the target of a jump – hence leader
  - Lines (1) and (2) is a basic block
- Statement following (12) is a leader
  - Statements (3) to (12) is another basic block

Control flow graph

B1

1. prod := 0
2. i := 1

B2

3. t1 := 4 * I
4. t2 := a[t1]
5. t3 := 4 * I
6. t4 := b [t3]
7. t5 := t2 *t4
8. t6 := prod + t5
9. prod := t6
10. t7 := i+1
11. i := t7
12. If i <= 20 goto (3)

B1 is the predecessor of B2 and B2 is the successor of B1

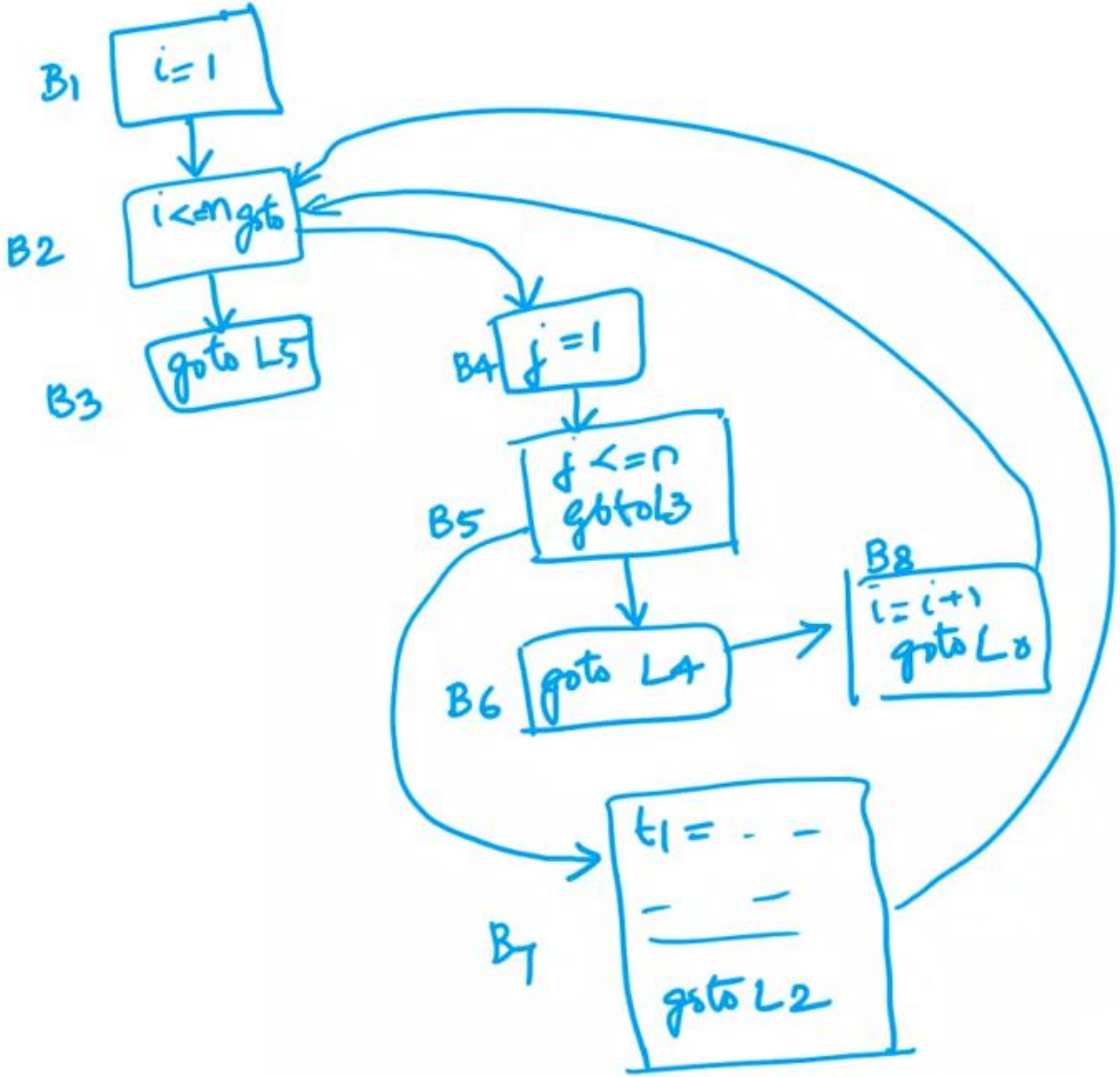
Matrix Addition

```
for i=1 to n
    for j=1 to n
        A[i][j] = B[i][j] + C[i][j]
```

$n_1 \times n_2$

```
i=1
while i<=n {
    j=1
    while j<=n {
        A[i][j] = B[i][j] + C[i][j]
        j=j+1
    }
    i=i+1;
}
```

```
      i=1
L0: if i<=n goto L1
      goto L5
L1: j=1
L2: if j<=n goto L3
      goto L4
L3: t1 = i*n
      t2 = j+t1
      t3 = t2*4
      t4 = i*n
      t5 = j+t1
      t6 = t5*4
      t7 = B[t3]
      t8 = C[t6]
      t9 = t7+t8
      t10 = i*n
      t11 = j+t10
      t12 = t11*4
      A[t12] = t9
      j=j+1
      goto L2
L4: i=i+1
      goto L0
L5:
```



B1
i=1
B2
i<=n goto
B3
goto L5
B4
j=1
B5
j<=n
gotoL3
B6
goto L4
B8
i=i+1
goto L8
B7
t1 = . . .
goto L2

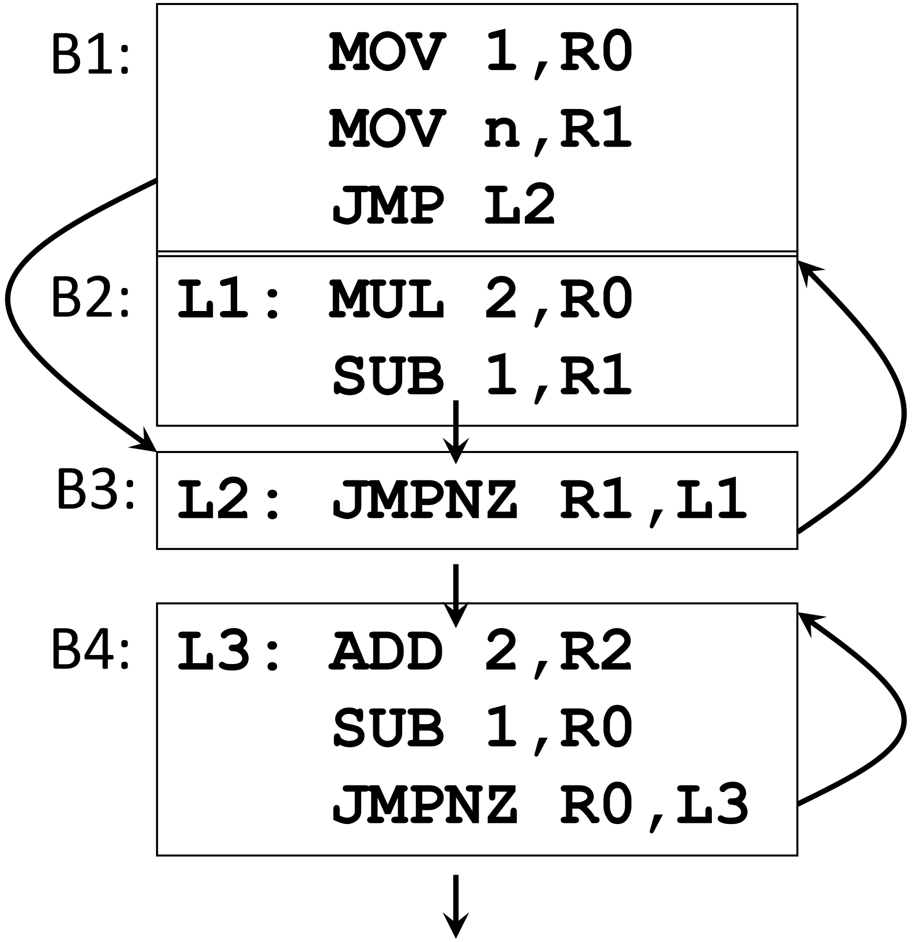
Loops

- A loop is a collection of basic blocks, such that
  - All blocks in the collection are strongly connected – any node to any other there is a path of length one or more within the loop
  - The collection has a unique entry, and the only way to reach a block in the loop is through the entry

Outer & Inner loops

- Loops not containing any other loop is Inner loop
- Loops that has one or more inner loops is outer loop

Loops (Example)



Strongly connected components:

SCC={{B2,B3}, {B4} }

Entries:
B3, B4

Transformations on Basic blocks

- Basic block computes set of expressions
- Values of the variables outside the block is decided by the computation inside the block
- Two basic blocks are equivalent if they compute the same set of expressions

Equivalence of Basic Blocks

- Two basic blocks are *equivalent* if they compute the same set of expressions

```
b  := 0
t1 := a + b
t2 := c * t1
a  := t2
```

```
a := c*a
b := 0
```

```
a  := c * a
b  := 0
```

```
a := c*a
b := 0
```

Transformation on Basic Blocks

- A code-improving transformation is a code optimization to improve speed or reduce code size
- Global transformations are performed across basic blocks
- Local transformations are only performed on single basic blocks
- Transformations must be safe and preserve the meaning of the code
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form

Transformations on Basic blocks

- Structure-Preserving Transformation
  - Syntactic structure of the statements in the basic blocks are not altered
- Algebraic Transformation
  - Mathematical identity based transformation and thus altering the syntactic structure

Transformations on Basic blocks

- Structure-Preserving Transformation
    - Common Sub-expression Elimination
    - Dead-code elimination
    - Renaming of temporary variables
    - Interchange of two independent adjacent statements

Common-Subexpression Elimination

- Remove redundant computations

```
a := b + c
b := a - d
c := b + c
d := a - d
```

```
a := b + c
b := a - d
c := b + c
d := b
```

```
t1 := b * c
t2 := a - t1
t3 := b * c
t4 := t2 + t3
```

```
t1 := b * c
t2 := a - t1
t4 := t2 + t1
```

Dead Code Elimination

- Remove unused statements

```
b := a + 1
a := b + c
…
```

```
b := a + 1
…
```

Assuming **a** is *dead* (not used)

```
if true goto L2
```

```
b := x + y
…
```

Remove unreachable code

Renaming Temporary Variables

- Temporary variables that are dead at the end of a block can be safely renamed

```
t1 := b + c
t2 := a - t1
t1 := t1 * d
d := t2 + t1
```

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

Interchange of Statements

- Independent statements can be reordered

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

```
t1 := b + c
t3 := t1 * d
t2 := a - t1
d := t2 + t3
```

Algebraic Transformations

- Change arithmetic operations to transform blocks to algebraic equivalent forms

```
t1  := a - a
t2  := b + t1
t3  := 2 * t2
x     := y ** 2
```

⇒

```
t1  := 0
t2  := b
t3  := t2 << 1
x   := y * y
```

Next-Use

- Next-use information is needed for dead-code elimination and register assignment
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement

  $$i: x := y \text{ op } z$$

  - Add liveness/next-use info on *x*, *y*, and *z* to statement *i*
  - Set *x* to “not live” and “no next use”
  - Set *y* and *z* to “live” and the next uses of *y* and *z* to *i*

Next-Use (Step 1)

i: **`a := b + c`**

j: **`t := a + b`**

[*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true, *nextuse*(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = none]

Attach current live/next-use information
Because info is empty, assume variables are live

Next-Use (Step 2)

i: **`a := b + c`**

Compute live & next-use information at line *j*

live(**`a`**) = true	*nextuse*(**`a`**) = *j*
live(**`b`**) = true	*nextuse*(**`b`**) = *j*
live(**`t`**) = false	*nextuse*(**`t`**) = none

j: **`t := a + b`**

[*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none,
nextuse(**`t`**) = none]

Next-Use (Step 3)

Attach current live/next-use information to line *i*

i: **`a := b + c`**

[*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = false, *nextuse*(**`a`**) = *j*, *nextuse*(**`b`**) = *j*, *nextuse*(**`t`**) = none]

j: **`t := a + b`**

[*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`c`**) = true, *nextuse*(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`c`**) = none]

Next-Use (Step 4)

Compute live/next-use information at line *i*

live(**a**) = false	*nextuse*(**a**) = none
live(**b**) = true	*nextuse*(**b**) = *i*
live(**c**) = true	*nextuse*(**c**) = *i*
live(**t**) = false	*nextuse*(**t**) = none

i: `a := b + c` [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = false, *nextuse*(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**t**) = none]

j: `t := a + b` [*live*(**a**) = false, *live*(**b**) = false, *live*(**t**) = false, *nextuse*(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Summary

- Converting code to Basic blocks
- Possible transformations in basic blocks
- Next-use computation and its use