CS536

Flow Graph and Basic Blocks

A Sahu

CSE, IIT Guwahati

Outline

- Flow Graphs
- Basic Block: detection
- Loop Detection
- Transformation on Basic Block
- Peep Hole Optimization or Window optimization
- Register Allocation

Flow Graphs

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

```
     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)

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

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

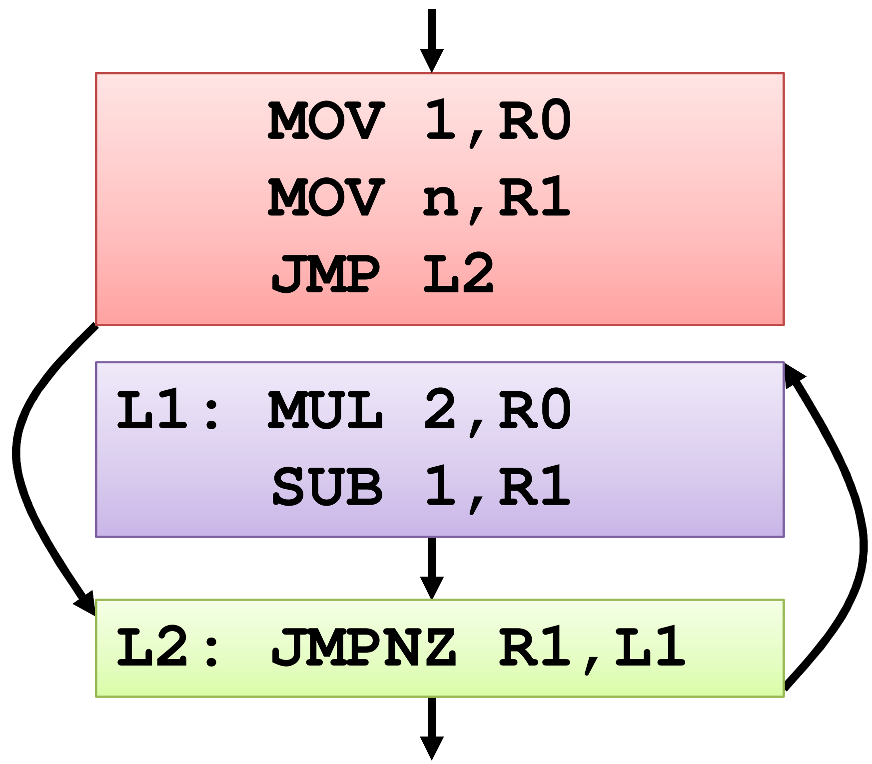
```
L1: MUL 2,R0
    SUB 1,R1
```

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

Basic Blocks and Control Flow Graphs

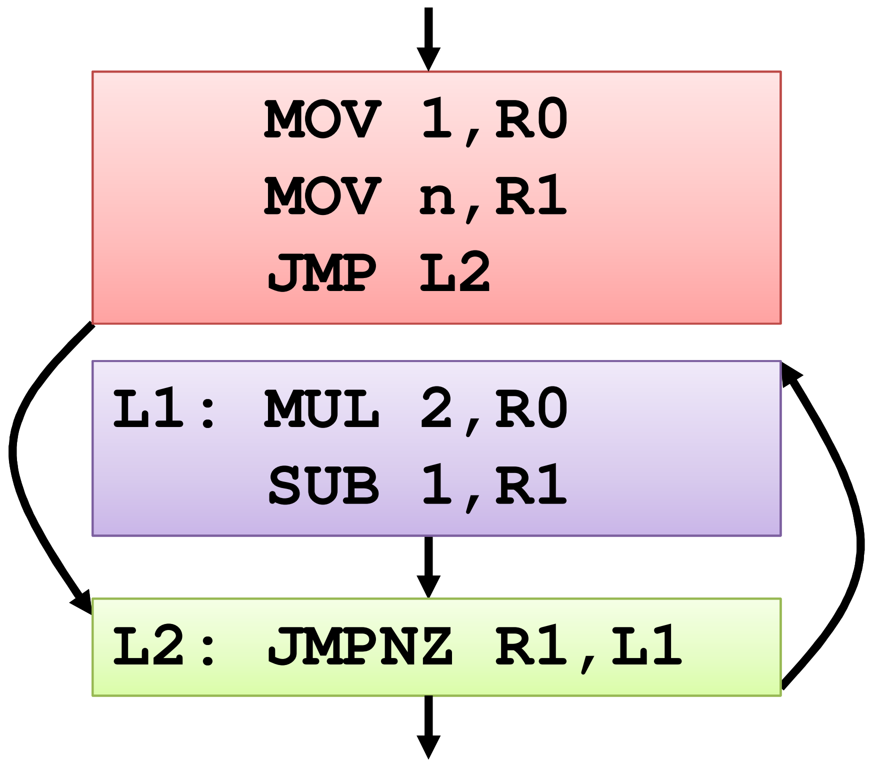
- A *control flow graph* (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \rightarrow B_j$ iff B_j can be executed immediately after B_i

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



Successor and Predecessor Blocks

- Suppose the CFG has an edge $B_1 \rightarrow B_2$
  - Basic block B_1 is a *predecessor* of B_2
  - Basic block B_2 is a *successor* of B_1



Partition Algorithm for Basic Blocks

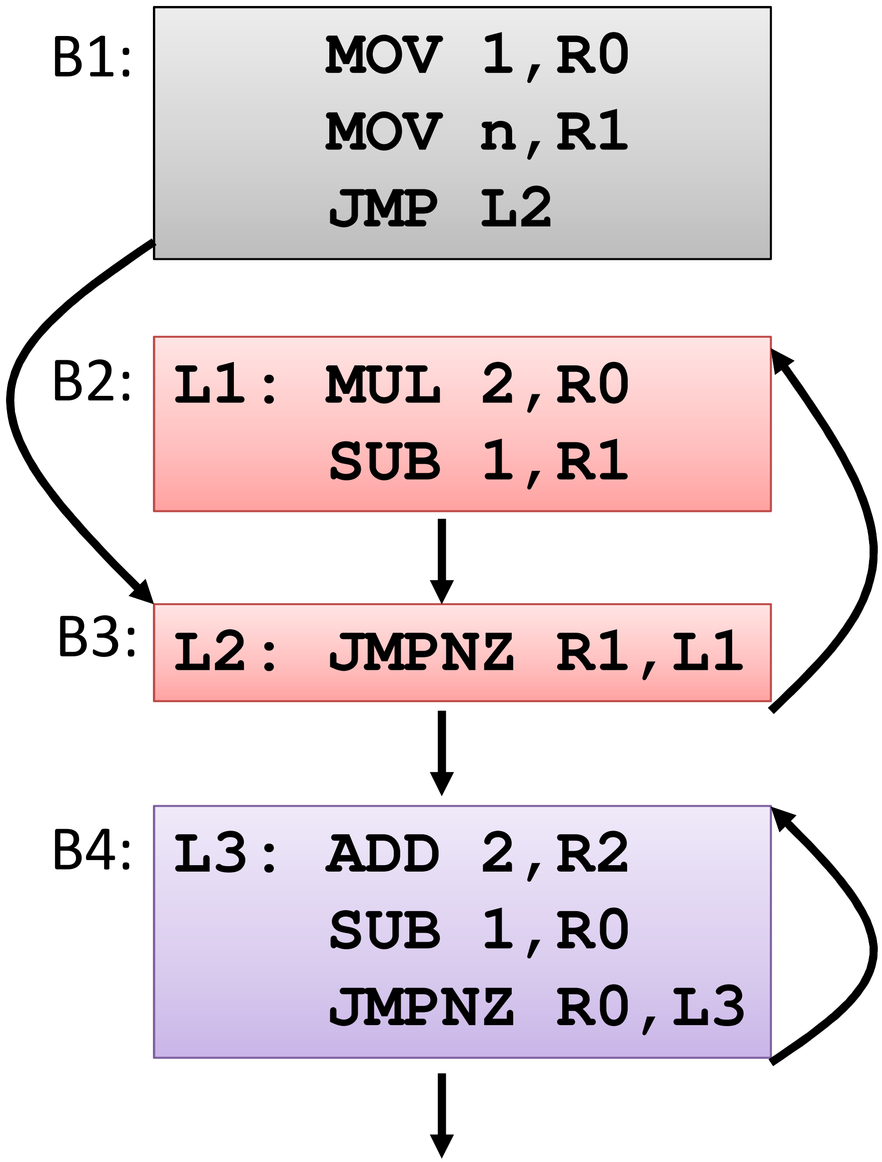
- *Input*: A sequence of three-address statements
- *Output*: A list of basic blocks with each three-address statement in exactly one block

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

Loops

- A *loop* is a collection of basic blocks, such that
  - All blocks in the collection are *strongly connected*
  - The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry

Loops (Example)



Strongly connected components:

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

Unique Entries: **B3**, **B4**

Equivalence of Basic Blocks

- Two basic blocks are (semantically) *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
```

Blocks are equivalent, assuming **`t1`** and **`t2`** are *dead*: no longer used (no longer *live*)

Transformations 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

Common-Subexpression Elimination

- Remove redundant computations

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

```
a1  := b0 + c0
b1  := a1 - d0
c  := b1 + c0
d  := a1 - d0
```

```
a1  := b0 + c0
b1  := a1 - d0
c  := b1 + c0
d  := b1
```

```
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
```

Normal-form block

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
```

Note that normal-form blocks permit all statement interchanges that are possible

Algebraic Transformations

- Change arithmetic operations to transform blocks to algebraic equivalent forms

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

```
t1 := 0
t2 := b
t3 := t2 << 1
```

Thanks