CS323: Compilers

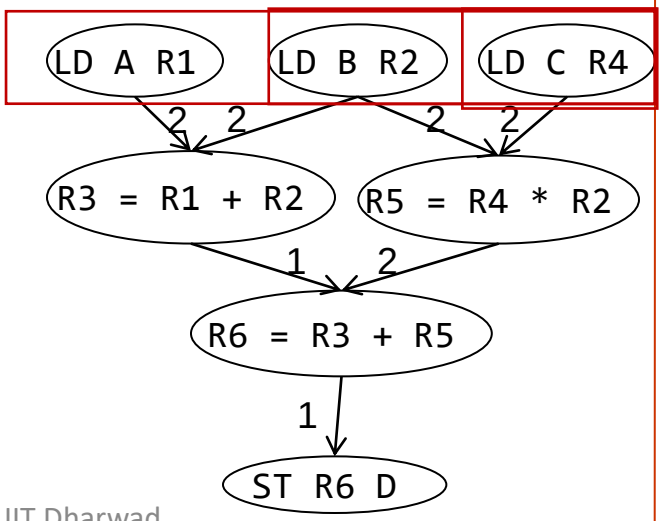
Spring 2023

Week 11: Instruction Scheduling (contd..), Control Flow Graphs

Acknowledgements: Milind Kulkarni

List scheduling - Example

```
1. LD A, R1
2. LD B, R2
3. R3 = R1 + R2
4. LD C, R4
5. R5 = R4 * R2
6. R6 = R3 + R5
7. ST R6, D
```



Cycle #	Available Instruction(s)	Scheduled Instruction(s)	Completed Instruction(s)
0	1, 2, 4	1*	
1	2, 4		
2	2, 4	2*	1
3	4		
4	3,4	3,4	2
5			3
6	5	5	4
7			
8	6	6	5
9	7	7	6
10			7

*an instruction from the list of available instructions is picked at random and scheduled

List scheduling

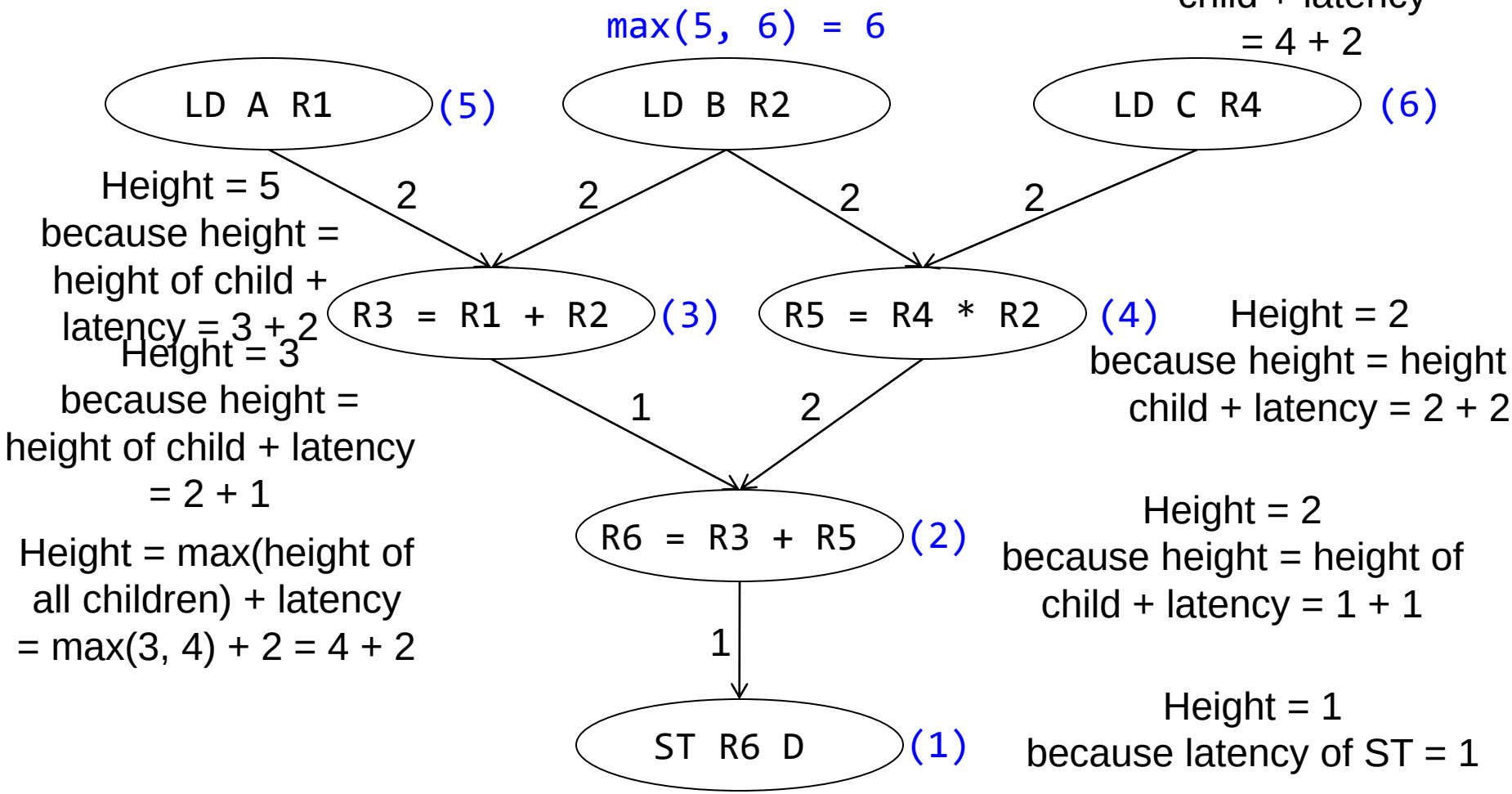
1. LD A, R1
2. LD B, R2
3. R3 = R1 + R2
4. LD C, R4
5. R5 = R4 * R2
6. R6 = R3 + R5
7. ST R6, D

Cycle	ALU0	ALU1	LD/ST
0			1
1			1
2			2
3			2
4	3		4
5			4
6	5		
7			
8	6		
9			7
10			

Height-based scheduling

- Important to prioritize instructions
  - Instructions that have a lot of downstream instructions dependent on them should be scheduled earlier
- Instruction scheduling NP-hard in general, but **height-based scheduling** is effective
- Instruction height = latency from instruction to farthest-away leaf
  - Leaf node height = instruction latency
  - Interior node height = max(heights of children + instruction latency)
- Schedule instructions with highest height first

Computing heights



Height-based list scheduling

1. LD A, R1
2. LD B, R2
3. R3 = R1 + R2
4. LD C, R4
5. R5 = R4 * R2
6. R6 = R3 + R5
7. ST R6, D

Cycle	ALU0	ALU1	LD/ST
0			2
1			2
2			4
3			4
4	5		1
5			1
6	3		
7	6		
8	7		
9			
10			

Instruction Scheduling - Exercise

- 2 ALUs (fully pipelined) and one LD/ST unit (not pipelined) are available.
- Either of the ALUs can execute ADD (1 cycle). Only one of the ALUs can execute MUL (2 cycles).
- LDs take up an ALU for 1 cycle and LD/ST unit for two cycles.
- STs take up an ALU for 1 cycle and LD/ST unit for one cycle.

i) Draw reservation tables, ii)DAG for the code shown iii) schedule using height based list scheduling.

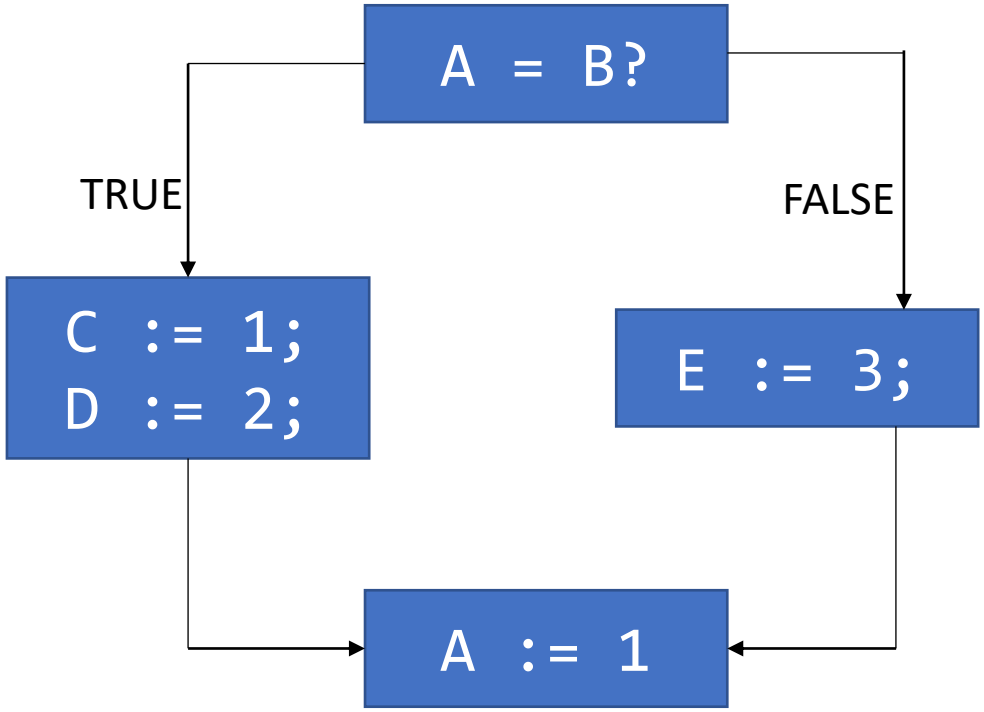
```
1: LD A R1          11: ST R10  E
2: LD B R2          12: ST R7  F
3: LD C R3
4: LD D R4
5: R5 = R1 + R2
6: R6 = R5 * R3
7: R7 = R1 + R6
8: R8 = R6 + R5
9: R9 = R4 + R7
10: R10 = R9 + R8
```

Basic Blocks and Flow Graphs

- Basic Block
  - Maximal sequence of consecutive instructions with the following properties:
    - The first instruction of the basic block is the *only entry point*
    - The last instruction of the basic block is either the halt instruction or the *only exit point*
- Flow Graph
  - Nodes are the basic blocks
  - Directed edge indicates which block follows which block

Basic Blocks and Flow Graphs - Example

```
if A = B then
   C := 1;
   D := 2;
else
   E := 3
fi
A := 1;
```



A data flow graph

Flow Graphs

- Capture how control transfers between basic blocks due to:
  - Conditional constructs
  - Loops
- Are necessary when we want optimize considering larger parts of the program
  - Multiple procedures
  - Whole program

Flow Graphs - Representation

- We need to label and track statements that are jump targets
  - **Explicit targets** – targets mentioned in jump statement
  - **Implicit targets** – targets that follow conditional jump statement
    - Statement that is executed if the branch is not taken
- Implementation
  - Linked lists for Basic Blocks
  - Graph data structures for flow graphs

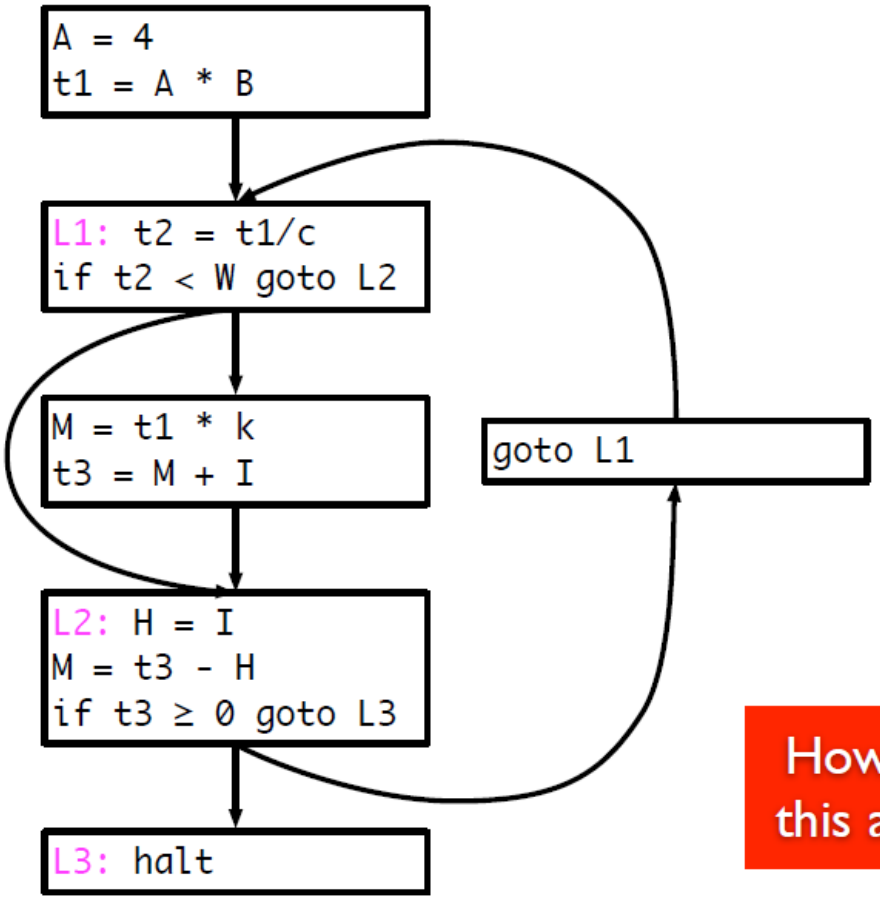
Running example

```
A = 4
t1 = A * B
repeat {
  t2 = t1/C
  if (t2 ≥ W) {
    M = t1 * k
    t3 = M + I
  }
  H = I
  M = t3 - H
} until (T3 ≥ 0)
```

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

CFG for running example



How do we build this automatically?

Constructing a CFG

- To construct a CFG where each node is a basic block
  - Identify *leaders*: first statement of a basic block
  - In program order, construct a block by appending subsequent statements up to, but not including, the next leader
- Identifying leaders
  - First statement in the program
  - Explicit target of any conditional or unconditional branch
  - Implicit target of any branch

Partitioning algorithm

- Input: set of statements, *stat(i)* = i^{th} statement in input
- Output: set of *leaders*, set of basic blocks where *block(x)* is the set of statements in the block with leader *x*
- Algorithm

```
leaders = {1}          //Leaders always includes first statement
for i = 1 to |n|       //|n| = number of statements
    if stat(i) is a branch, then
        leaders = leaders ∪ all potential targets
end for
worklist = leaders
while worklist not empty do
    x = remove earliest statement in worklist
    block(x) = {x}
    for (i = x + 1; i ≤ |n| and i ∉ leaders; i++)
        block(x) = block(x) ∪ {i}
    end for
end while
```

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = ?

Basic blocks = ?

leaders = {1} //Leaders always includes first statement
for i = 1 to |*n*| //|*n*| = number of statements
  if *stat(i)* is a branch, **then**
    leaders = *leaders* ∪ all potential targets
end for
worklist = *leaders*

Running example

```
 1          A = 4
 2          t1 = A * B
 3   L1:    t2 = t1 / C
 4          if t2 < W goto L2
 5          M = t1 * k
 6          t3 = M + I
 7   L2:    H = I
 8          M = t3 - H
 9          if t3 ≥ 0 goto L3
10          goto L1
11   L3:    halt
```

Leaders = {1}
Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1}
Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7}

Basic blocks =

Running example

```
1         A = 4
2         t1 = A * B
3   L1:   t2 = t1 / C
4         if t2 < W goto L2
5         M = t1 * k
6         t3 = M + I
7   L2:   H = I
8         M = t3 - H
9         if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11}

Basic blocks = ?

worklist = *leaders*
while *worklist* not empty **do**
  x = remove earliest statement in *worklist*
  block(*x*) = {*x*}
  for (*i* = *x* + *1*; *i* ≤ |*n*| and *i* ∉ *leaders*; i++)
    block(*x*) = *block*(*x*) ∪ {*i*}
  end for
end while

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11}

Block(1) = ?

Basic blocks =

Running example

```
 1         A = 4
 2         t1 = A * B
 3   L1:   t2 = t1 / C
 4         if t2 < W goto L2
 5         M = t1 * k
 6         t3 = M + I
 7   L2:   H = I
 8         M = t3 - H
 9         if t3 ≥ 0 goto L3
10         goto L1
11   L3:   halt
```

Leaders = {1,3,5,7,10,11}

Basic blocks =

Block(1) = ?

Start from statement 2 and add till either the end or a leader is reached

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11} Block(1) = {1,2}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11} Block(3) = ?

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11} Block(3) = {3,4}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11} Block(5) = ?

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11} Block(5) = {5,6}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11} Block(7) = ?

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11} Block(7) = {7,8,9}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11} Block(10) = ?

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11} Block(10) = {10}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
 5        M = t1 * k
 6        t3 = M + I
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
10        goto L1
11  L3:   halt
```

Leaders = {1,3,5,7,10,11} Block(11) = {11}

Basic blocks =

Running example

```
 1        A = 4
 2        t1 = A * B
-------------------------
 3  L1:   t2 = t1 / C
 4        if t2 < W goto L2
-------------------------
 5        M = t1 * k
 6        t3 = M + I
-------------------------
 7  L2:   H = I
 8        M = t3 - H
 9        if t3 ≥ 0 goto L3
-------------------------
10        goto L1
-------------------------
11  L3:   halt
```

Leaders = {1, 3, 5, 7, 10, 11}

Basic blocks = { {1, 2}, {3, 4}, {5, 6}, {7, 8, 9}, {10}, {11} }

Putting edges in CFG

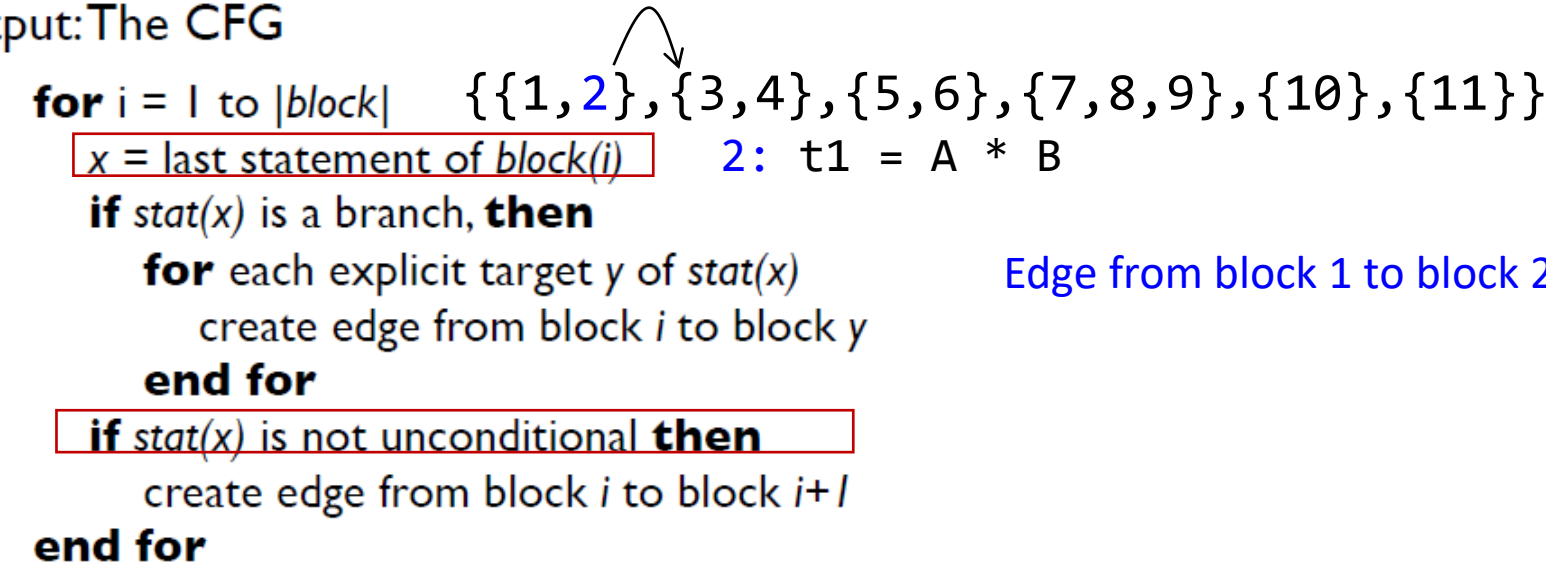
- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG

```
for i = 1 to |block|
    x = last statement of block(i)
    if stat(x) is a branch, then
        for each explicit target y of stat(x)
            create edge from block i to block y
        end for
    if stat(x) is not unconditional then
        create edge from block i to block i+1
end for
```

{{1,2},{3,4},{5,6},{7,8,9},{10},{11}}

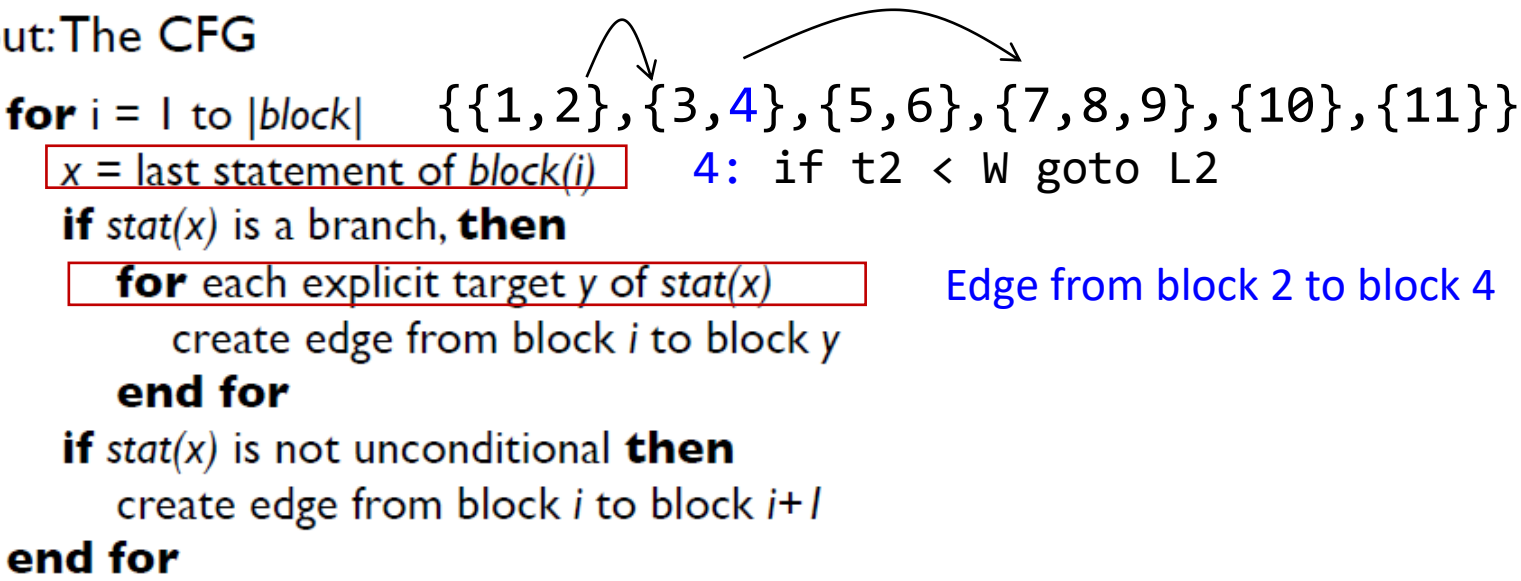
Putting edges in CFG

- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG



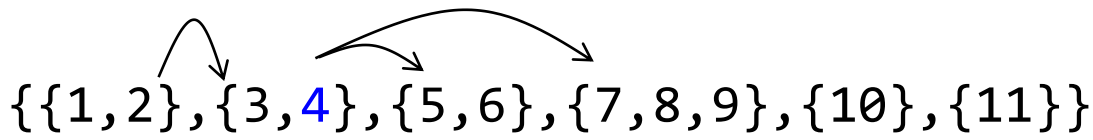
Putting edges in CFG

- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG



Putting edges in CFG

- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG

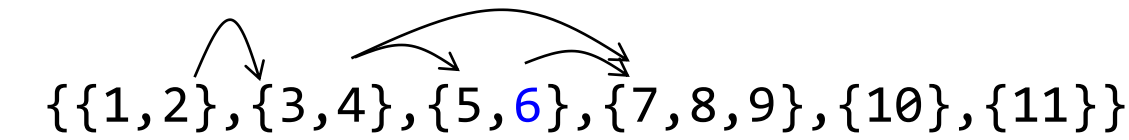



```
for i = 1 to |block|
    x = last statement of block(i)
    if stat(x) is a branch, then
        for each explicit target y of stat(x)
            create edge from block i to block y
        end for
    if stat(x) is not unconditional then
        create edge from block i to block i+1
end for
```

Edge from block 2 to block 3

Putting edges in CFG

- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG

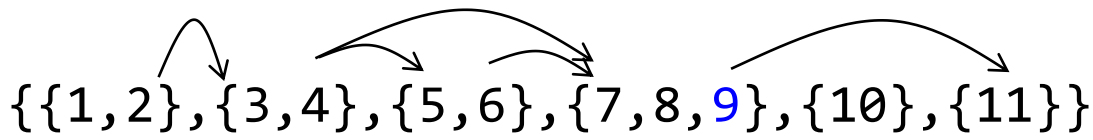



```
for i = 1 to |block|
  x = last statement of block(i)
  if stat(x) is a branch, then
    for each explicit target y of stat(x)
      create edge from block i to block y
    end for
  if stat(x) is not unconditional then
    create edge from block i to block i+1
end for
```

Edge from block 3 to block 4

Putting edges in CFG

- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG

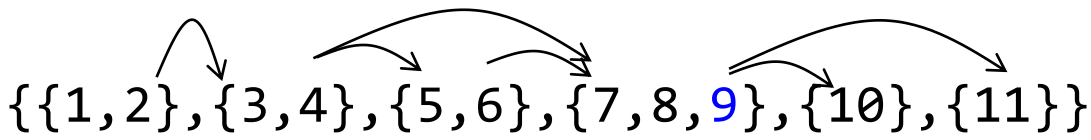



```
for i = 1 to |block|
    x = last statement of block(i)
    if stat(x) is a branch, then
        for each explicit target y of stat(x)
            create edge from block i to block y
        end for
    if stat(x) is not unconditional then
        create edge from block i to block i+1
end for
```

Edge from block 4 to block 6

Putting edges in CFG

- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG

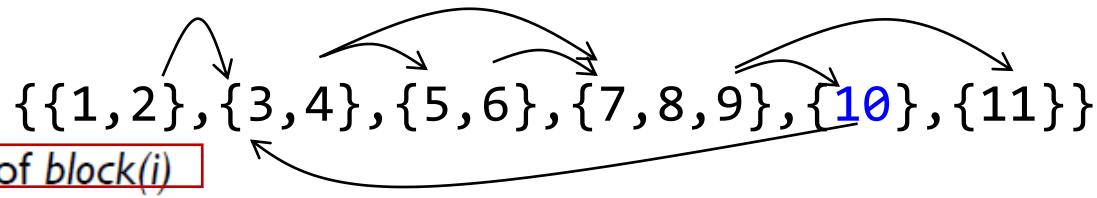



```
for i = 1 to |block|
  x = last statement of block(i)
  if stat(x) is a branch, then
    for each explicit target y of stat(x)
      create edge from block i to block y
    end for
  if stat(x) is not unconditional then
    create edge from block i to block i+1
end for
```

Edge from block 4 to block 5

Putting edges in CFG

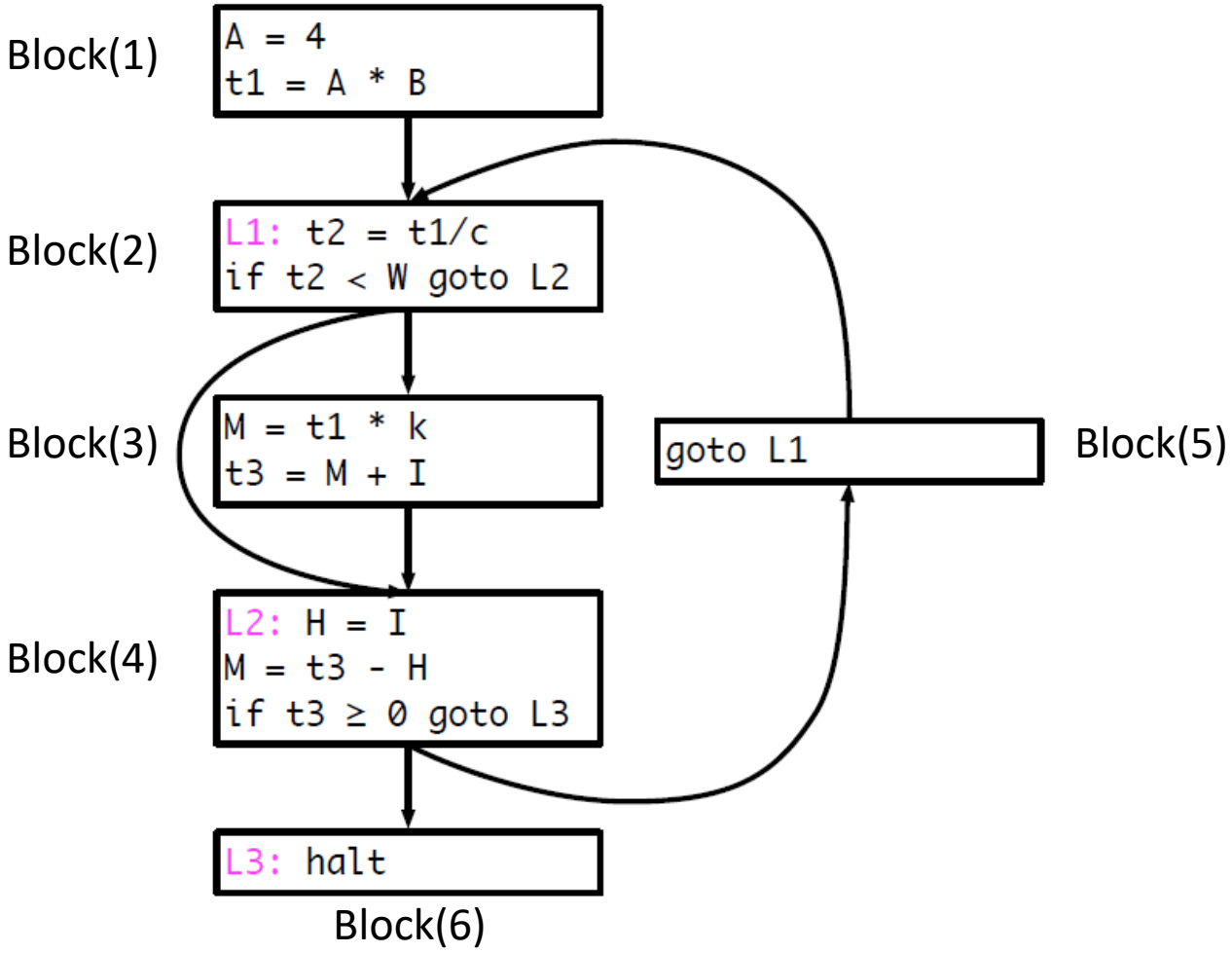
- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG




```
for i = 1 to |block|
    x = last statement of block(i)
    if stat(x) is a branch, then
        for each explicit target y of stat(x)
            create edge from block i to block y
        end for
    if stat(x) is not unconditional then
        create edge from block i to block i+1
end for
```

Edge from block 5 to block 2

Result

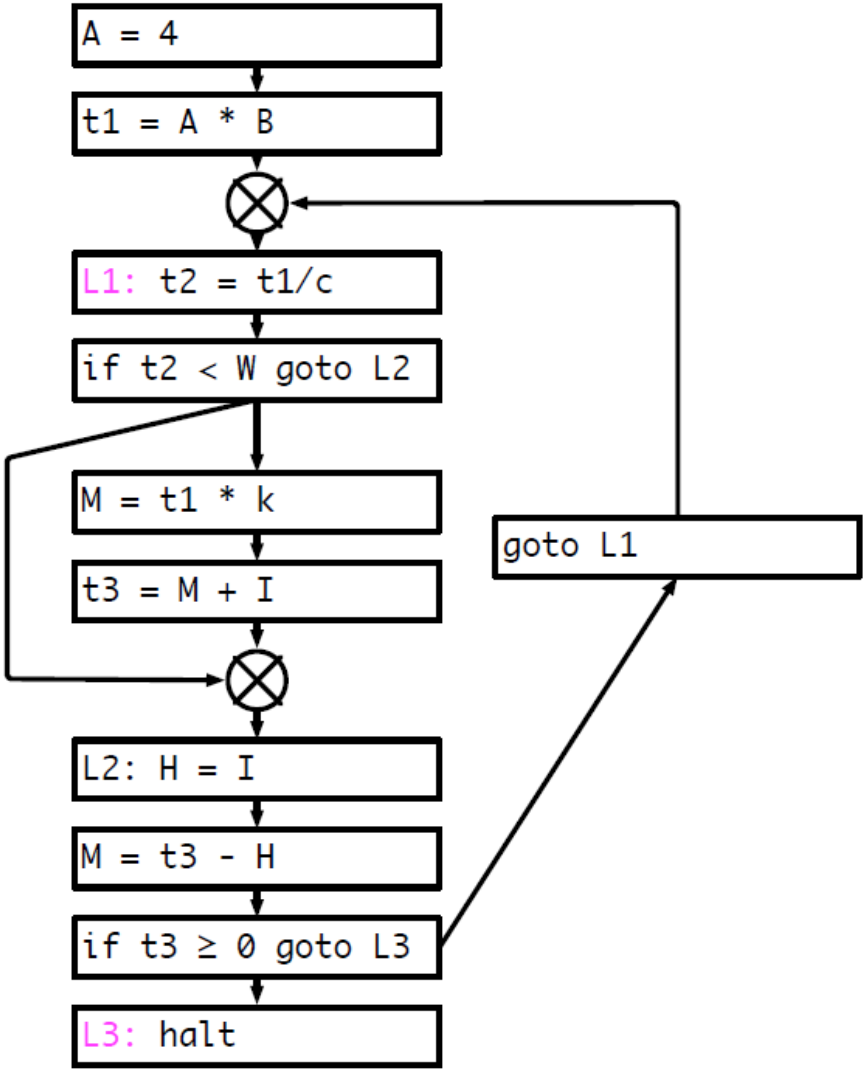

Block(1)
A = 4
t1 = A * B
Block(2)
L1: t2 = t1/c
if t2 < W goto L2
Block(3)
M = t1 * k
t3 = M + I
Block(4)
L2: H = I
M = t3 - H
if t3 ≥ 0 goto L3
goto L1
Block(5)
L3: halt
Block(6)

Discussion

- Some times we will also consider the *statement-level* CFG, where each node is a statement rather than a basic block
  - Either kind of graph is referred to as a CFG
- In statement-level CFG, we often use a node to explicitly represent *merging* of control
  - Control merges when two different CFG nodes point to the same node
- Note: if input language is *structured*, front-end can generate basic block directly
  - "GOTO considered harmful"

Slide Courtesy: Milind Kulkarni

Statement level CFG



Control Flow Graphs - Use

- Why do we need CFGs? - Global Optimization
  - Optimizing compilers do global optimization (i.e. optimize beyond basic blocks)
    - Differentiating aspect of normal and optimizing compilers
  - E.g. loops are the most frequent targets of global optimization (because they are often the “hot-spots” during program execution)

how do we identify loops in CFGs?

Identify Loops in CFGs

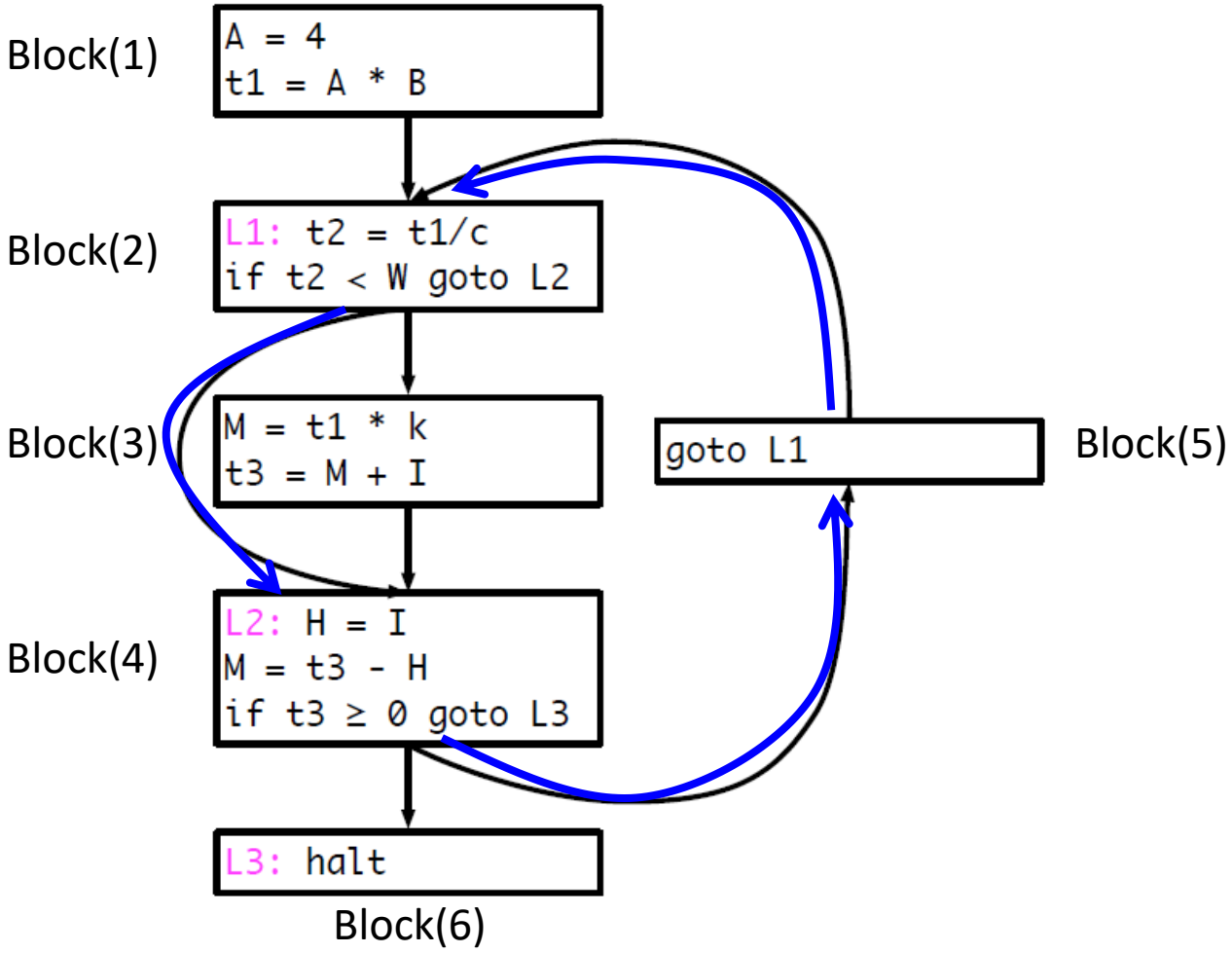
- Loops – **how do we identify loops in CFGs?**

  For a set of nodes, L, that belong to loop:

  1) There is a *loop entry node* with the property that no other node in L has a predecessor outside L. That is, every path from entry of the entire flow graph (*graph entry node*) to any node in L goes through the loop entry node.

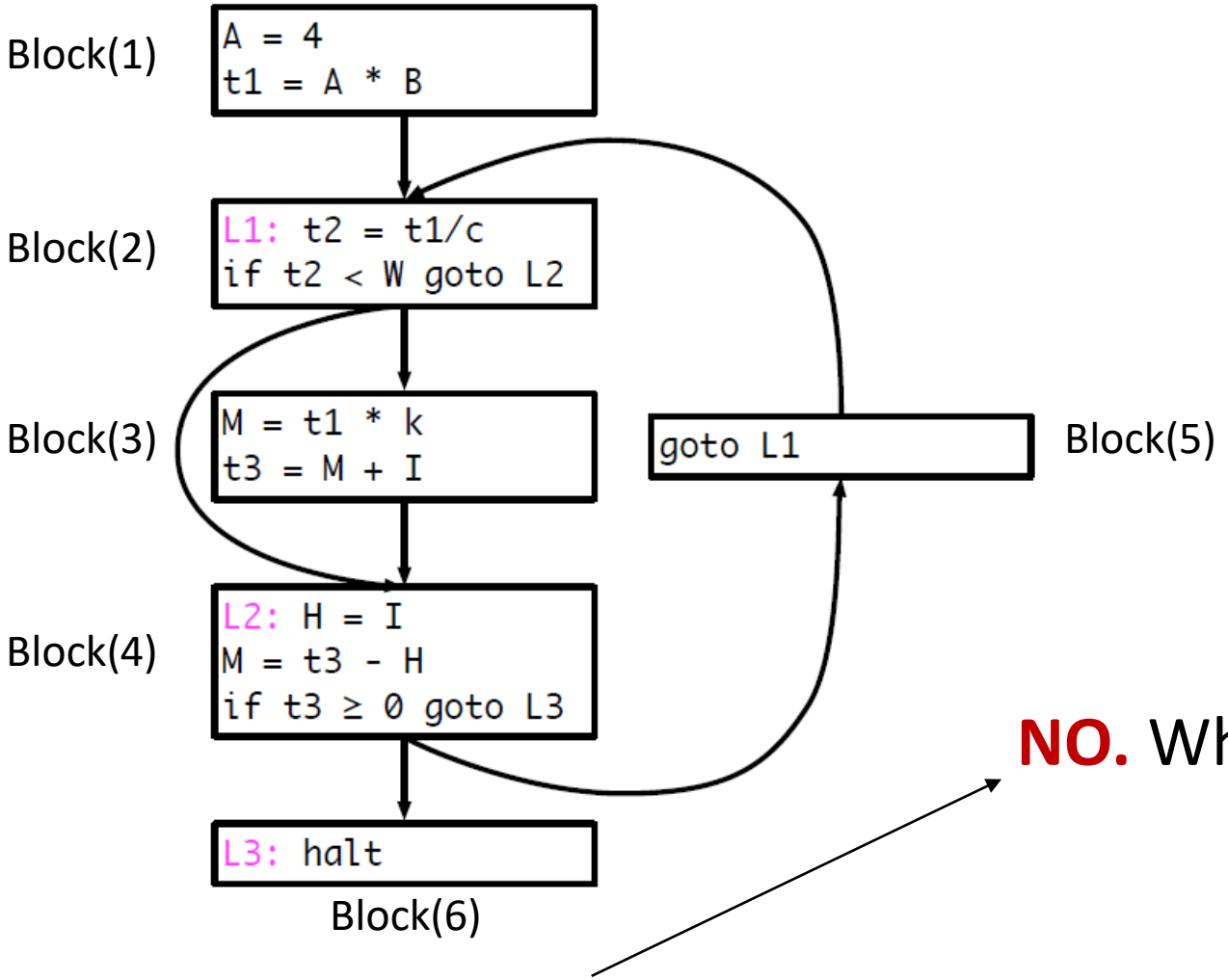
  2) *Every node in L* has a non-empty path, completely within L, to the entry of L.

Identify Loops in CFGs



Consider: {B2, B4, B5}. Is this a loop?, Are there other loops?

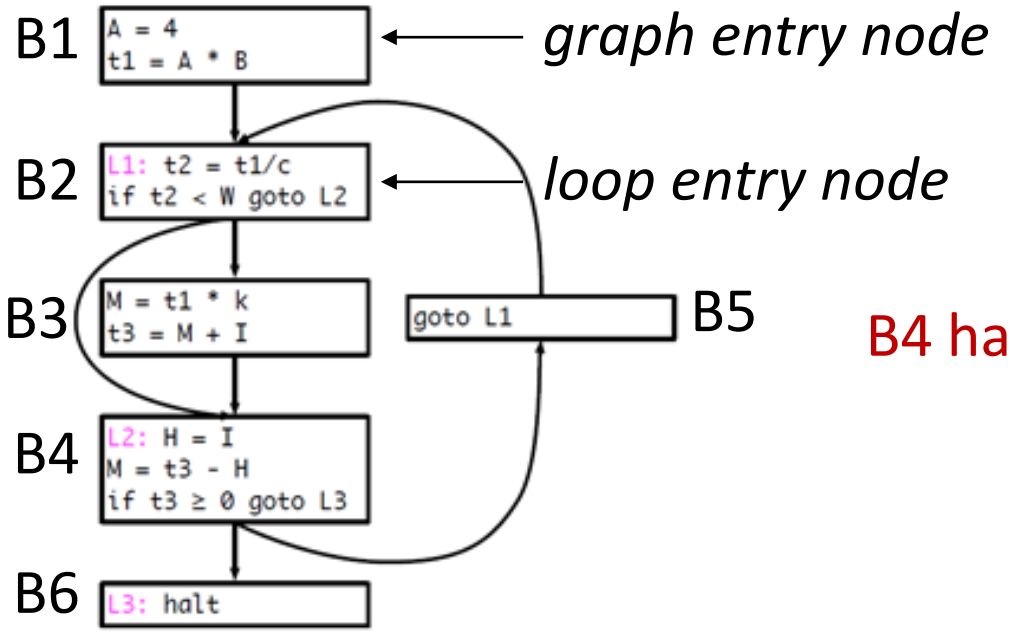
Identify Loops in CFGs



Consider: {B2, B4, B5}. Is this a loop?, Are there other loops?

Identify Loops in CFGs

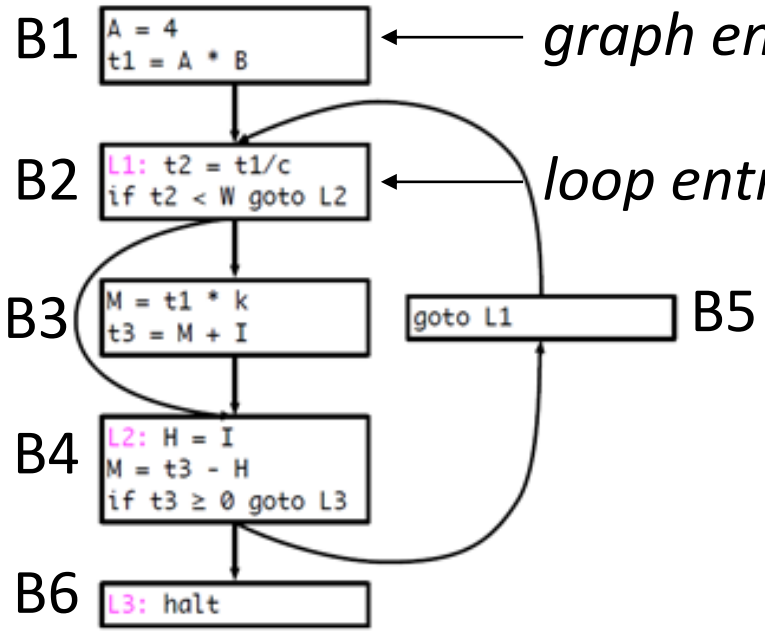
1) Is L={B2, B4, B5} a loop?. No. Consider:
   1) There is a *loop entry node* with the property that no other node in L has a predecessor outside L. That is, every path from entry of the entire flow graph (*graph entry node*) to any node in L goes through the loop entry node.



B4 has a predecessor B3 not in L

Identify Loops in CFGs

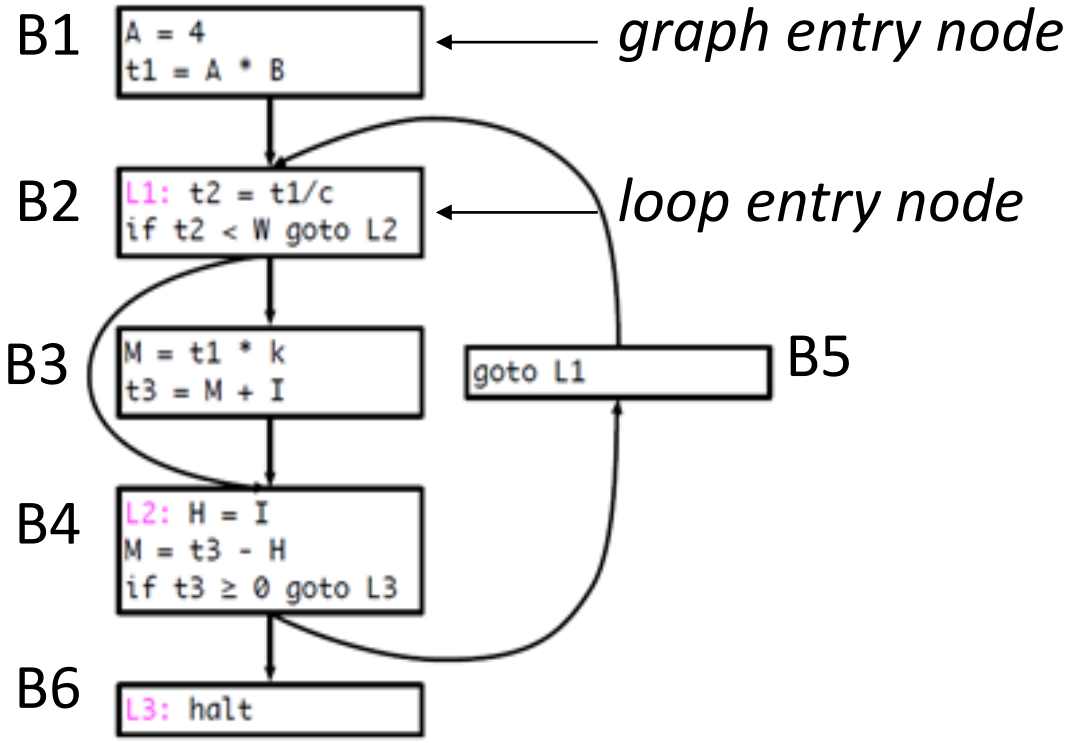
1) Is L={B2, B4, B5} a loop?. No. Consider:
   - *Every node in L* has a non-empty path, completely within L, to the entry of L.



B2 has a path B2->B3->B4->B5->B2 , where B3 is not in L

Identify Loops in CFGs

1) Is L={B2, B3, B4, B5} a loop?.



Optimizing Loops

Optimize Loops

- Example - Code Motion

Should be careful while doing optimization of loops

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

- Optimization: can move 10/I out of loop.

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

- Optimization: can move 10/I out of loop
- What if I = 0?

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

- Optimization: can move 10/I out of loop
- What if I = 0?
- What if I != 0 but loop executes zero times?

Optimization Criteria - Safety and Profitability

- Safety - is the code produced after optimization producing same result?
- Profitability - is the code produced after optimization running faster or uses less memory or triggers lesser number of page faults etc.

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

- E.g. moving I out of the loop introduces exception (when I=0)
- E.g. if the loop is executed zero times, moving `A(j) := 10/I` out is not profitable

Optimize Loops – Code Generation

- The outline of code generation for ‘for’ loops looked like this:

```
for (<init_stmt>;<bool_expr>;<incr_stmt>)
  <stmt_list>
end
```

```
  <init_stmt>
LOOP:
  <bool_expr>
  j<!op> OUT
  <stmt_list>
INCR:
  <incr_stmt>
  jmp LOOP
OUT:
```

```
for (i=0; i<=255;i++) {
    <stmt_list>
}
```

Naïve code generation

```
        code for i=0;
LOOP:   code for i<=255
        jump0 OUT
        code for <stmt_list>
INCR:   code for i++
        jump LOOP
OUT:
```

Question: why naïve is not good?

Optimize Loops – Code Generation

- What happens when ub is set to the maximum possible integer representable by the type of i?

```
for (i=0; i<=255;i++) {
    <stmt_list>
}
```

Better code:

```
    code for i=0;
    code for lb=0, ub=255
    code for lb<=ub
    jump0 OUT
LOOP:  code for <stmt_list>
    code for i=ub
    jump1 OUT
INCR:  code for i++
    jump LOOP
OUT:
```

generalizing:

```
    code for i=0;
    compute lb, ub
    code for lb<=ub
    jump0 OUT
    assign index=lb
    assign limit=ub
LOOP:  code for <stmt_list>
    code for index=limit
    jump1 OUT
INCR:  code for increment index
    jump LOOP
OUT:
```