CS323: Compilers

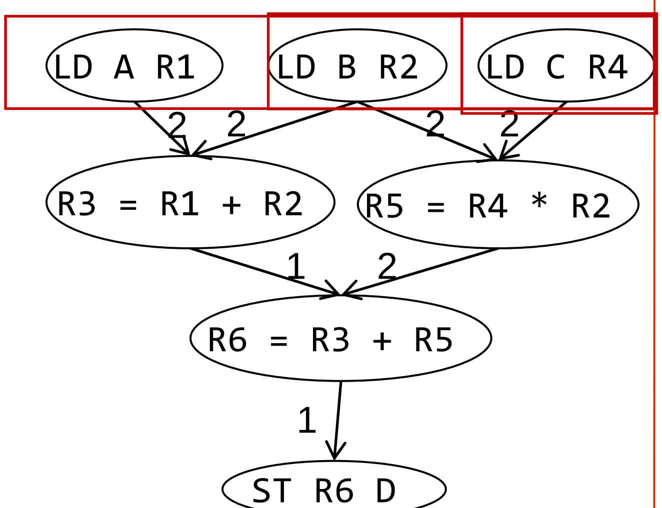
Spring 2023

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			

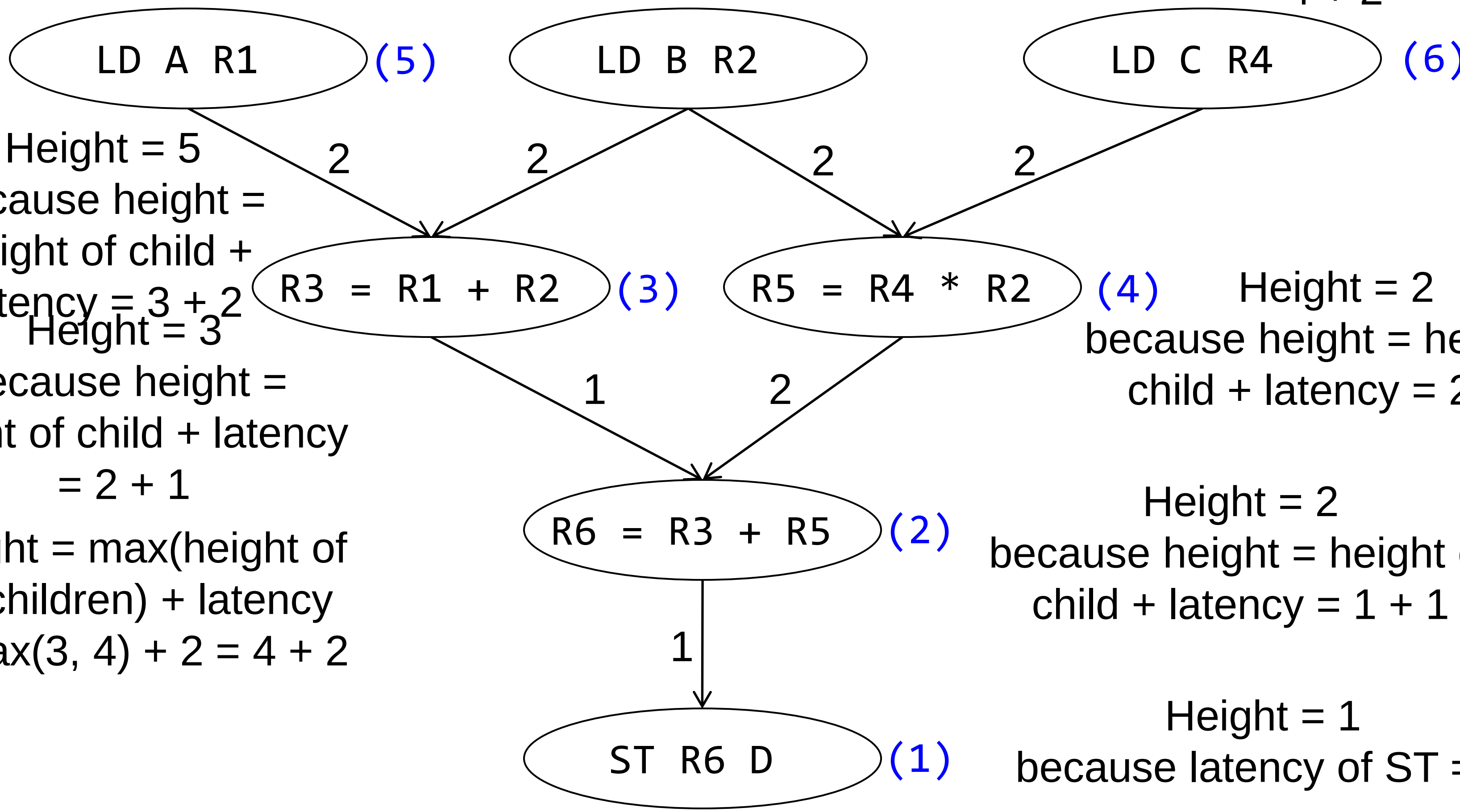
Height-based scheduling

- 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 = height of child + latency = 4 + 2

max(5, 6) = 6



Height = 5 because height = height of child + latency = 3 + 2

Height = 3 because height = height of child + latency = 2 + 1

Height = max(height of all children) + latency = max(3, 4) + 2 = 4 + 2

Height = 2 because height = height of child + latency = 2 + 2

Height = 2 because height = height of child + latency = 1 + 1

Height = 1 because latency of ST = 1

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.

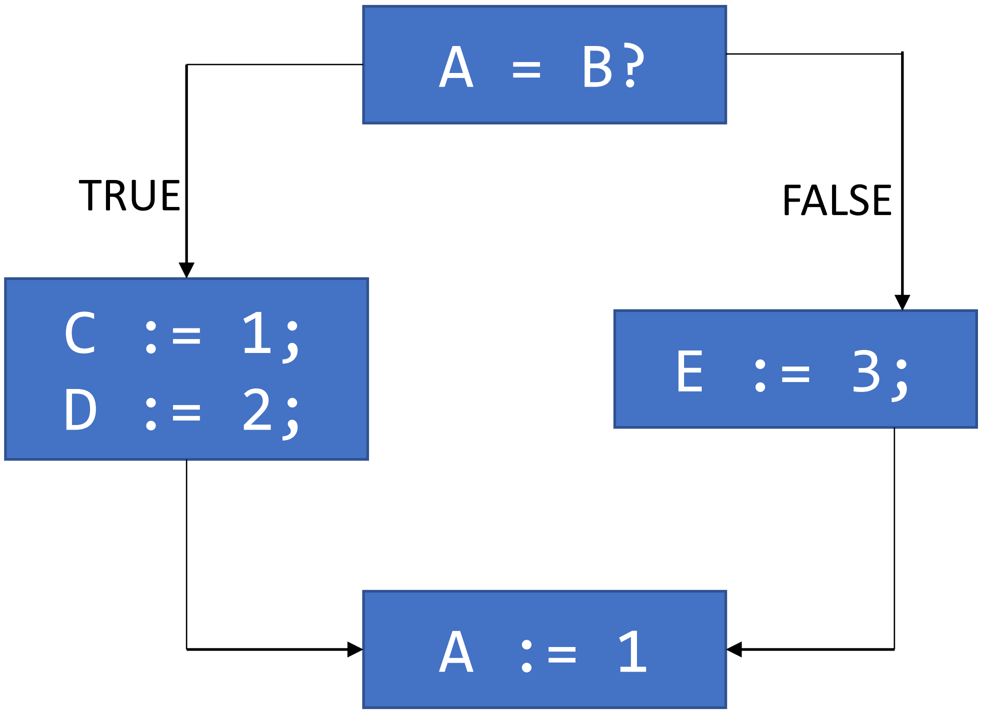
```
1: LD A R1
2: LD B R2
3: LD C R3
4: LD D R4
5: R5 = R1 + R2
6: R6 = R5 * R3
7: R7 = R1 + R6
8: R8 = R6 + R5
9: R9 = R4 + R7
10: R10 = R9 + R8
11: ST R10  E
12: ST R7  F
```

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

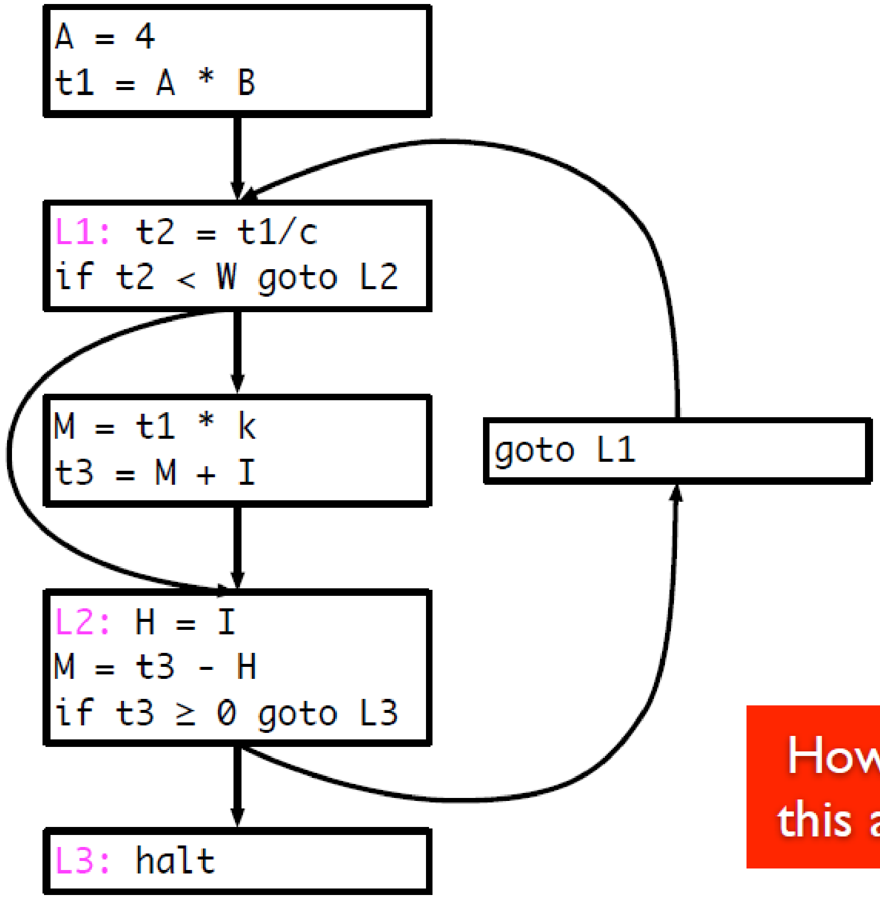
Running example

```
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

```
      A = 4
      t1 = A * B
L1:   t2 = t1 / C
      if t2 < W goto L2
      M = t1 * k
      t3 = M + I
L2:   H = I
      M = t3 - H
      if t3 ≥ 0 goto L3
      goto L1
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 ($i = x + 1; i \leq |n|$ and $i \notin$ *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}

Basic blocks =

Block(1) = ?

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

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

```
{{1,2},{3,4},{5,6},{7,8,9},{10},{11}}
2: t1 = A * B
```

Edge from block 1 to block 2

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

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}}
4: if t2 < W goto L2
```

Edge from block 2 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
```

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

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

```
{{1,2},{3,4},{5,6},{7,8,9},{10},{11}}
6: t3 = M + I
```

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

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

Edge from block 4 to block 6

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

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

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

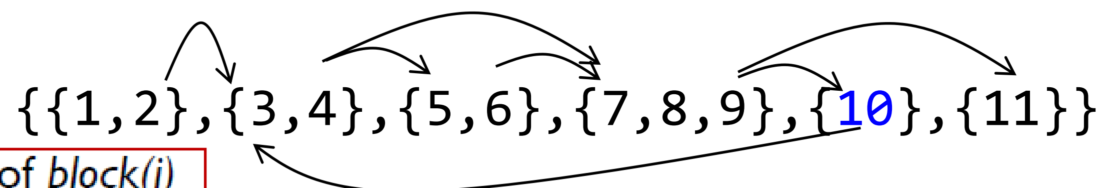
for i = 1 to |*block*|
    x = last statement of *block(i)*
    if *stat(x)* is a branch, **then**
        for each explicit target *y* of *stat(x)*
            create edge from block *i* to block *y*
        end for
    if *stat(x)* is not unconditional **then**
        create edge from block *i* to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

Edge from block 4 to block 5

Putting edges in CFG

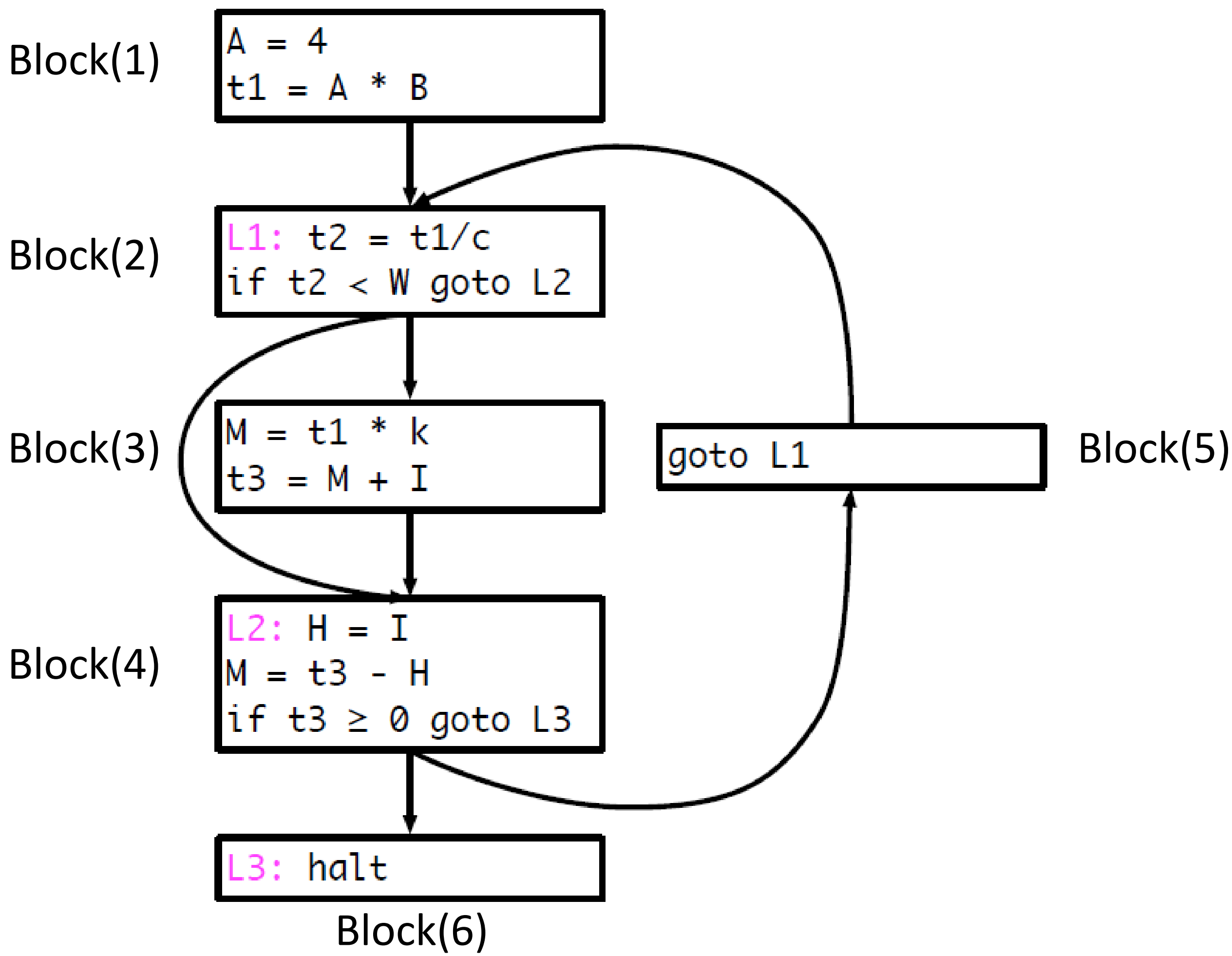
- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG

```
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

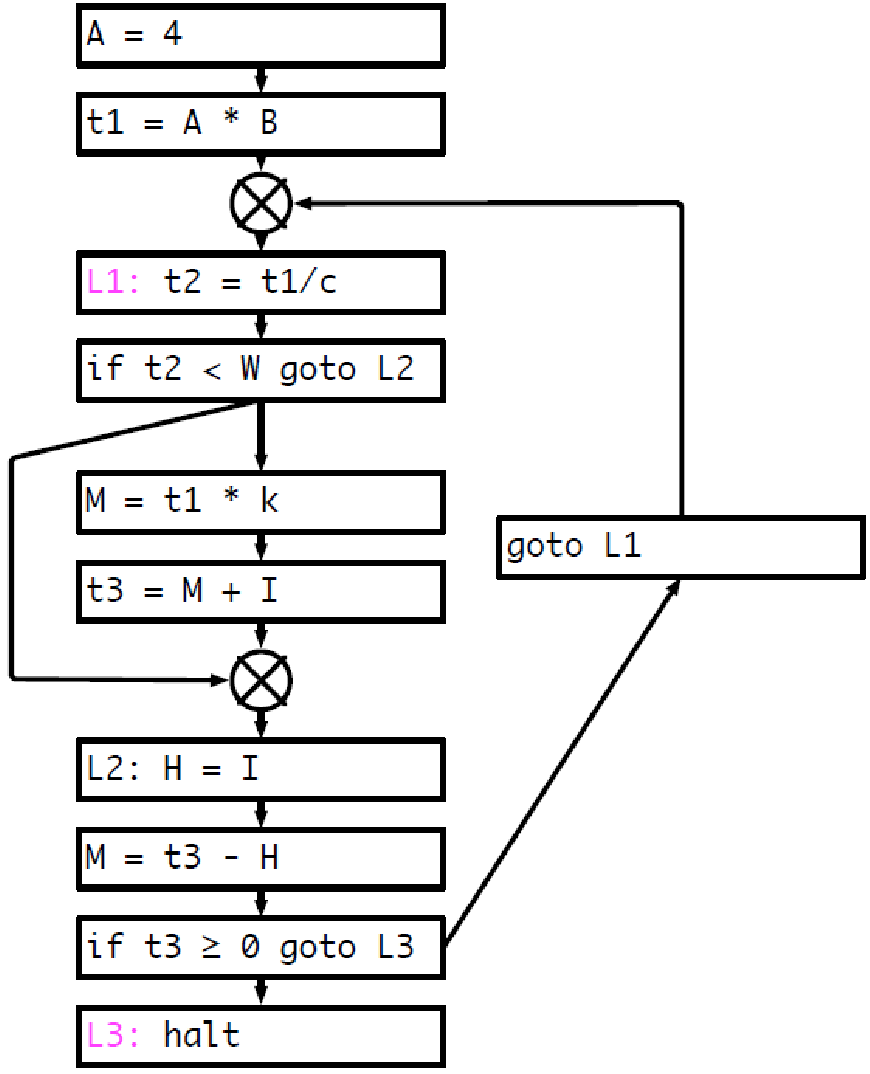
Result


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
goto L1
Block(5)
Block(4)
L2: H = I
M = t3 - H
if t3 ≥ 0 goto L3
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"

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

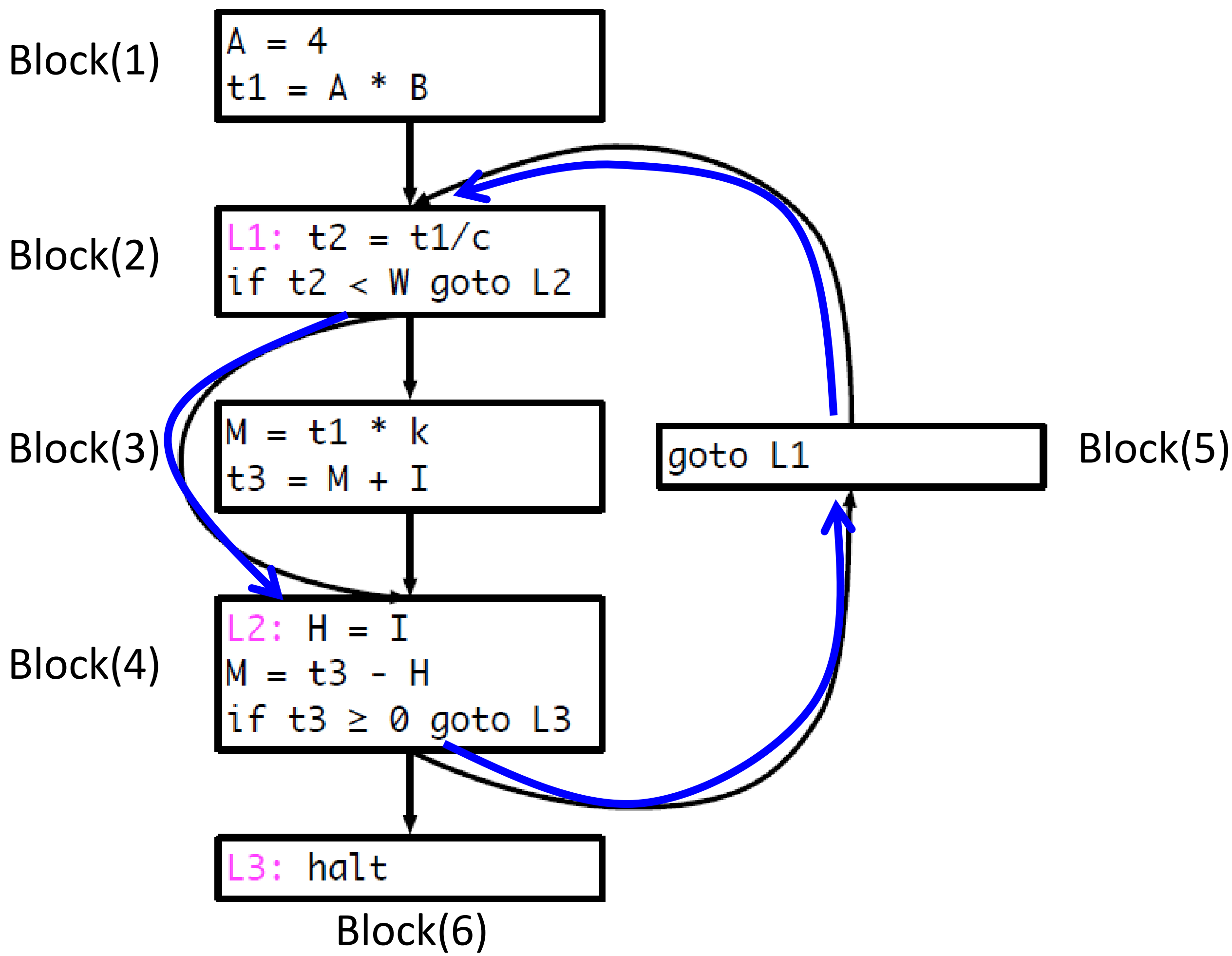
- Loops – **how do we identify loops in CFGs?**

  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.

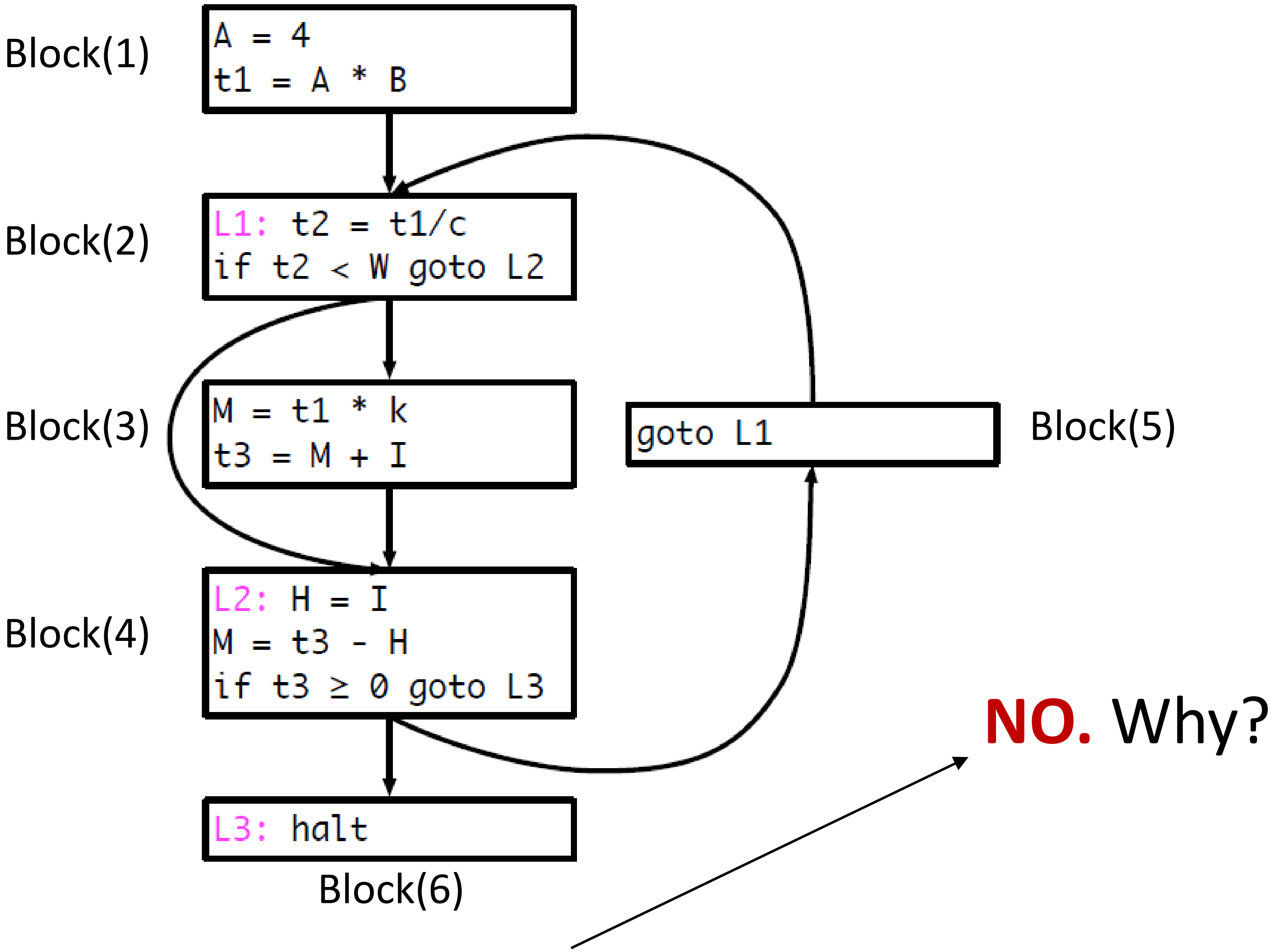
  2) Every node in L has a non-empty path, completely within L, to the entry of L.

Identify Loops in CFGs



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

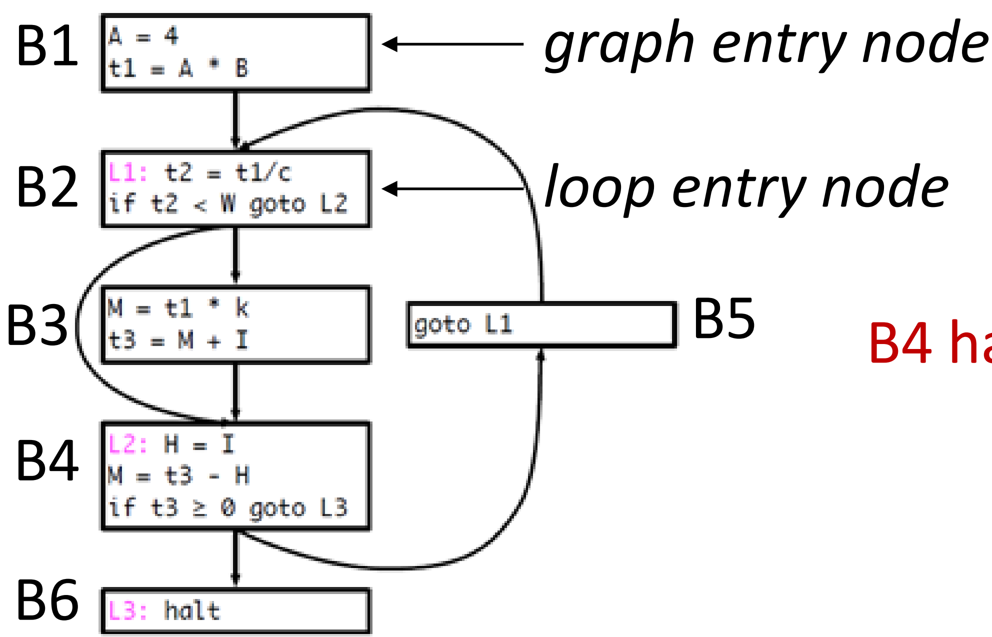
Identify Loops in CFGs



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

Identify Loops in CFGs

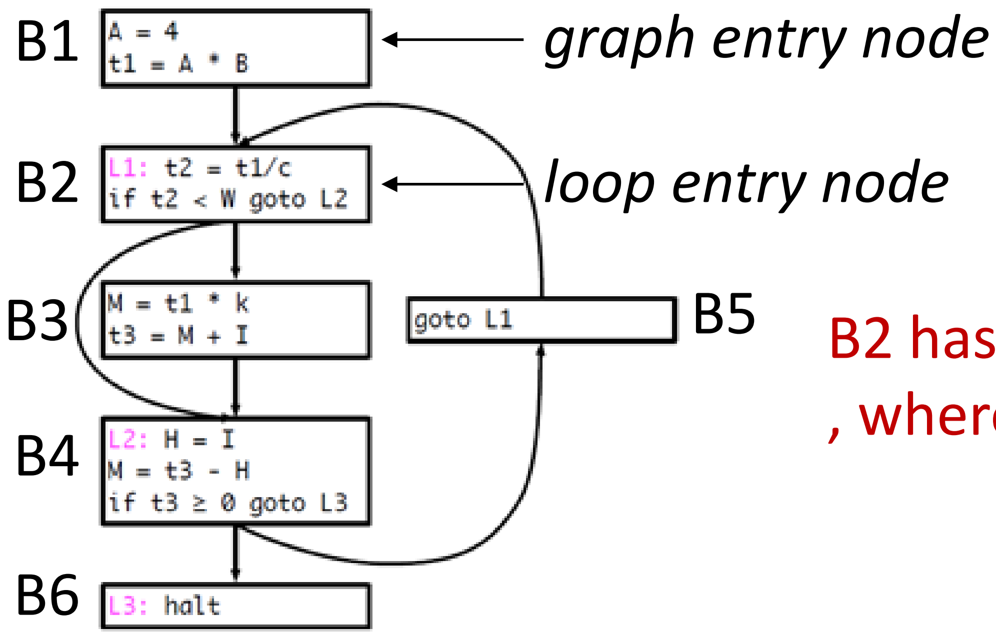
1) Is L={B2, B4, B5} a loop?. No. Consider:
   1) There is a *loop entry node* with the property that no other node in L has a predecessor outside L. That is, every path from entry of the entire flow graph (*graph entry node*) to any node in L goes through the loop entry node.



B4 has a predecessor B3 not in L

Identify Loops in CFGs

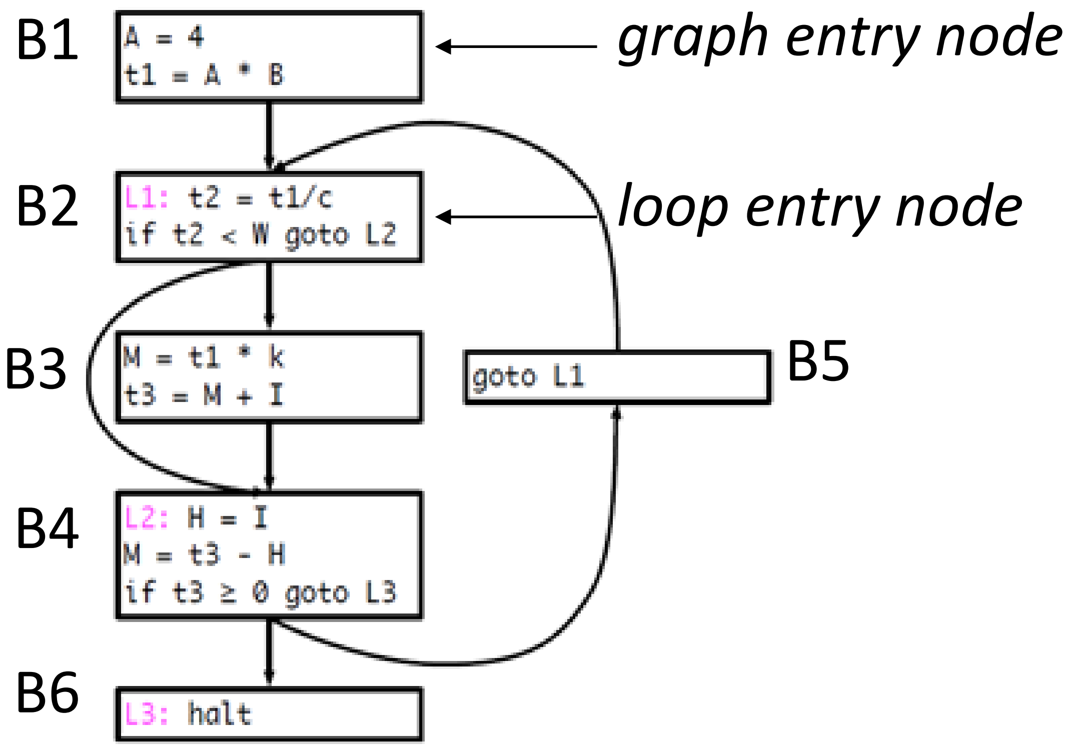
1) Is L={B2, B4, B5} a loop?. No. Consider:
   - *Every node in L* has a non-empty path, completely within L, to the entry of L.



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:
```

Optimize Loops -Identifying Invariant Expressions

- How do we identify expressions that can be moved out of the loop?
  - `LoopDef = {}` set of variables defined (i.e. whose values are overwritten) in the loop body
  - `LoopUse = { }` 'relevant' variables used in computing an expression

```
Mark_Invariants(Loop L) {
   1. Compute LoopDef for L
   2. Mark as invariant all expressions,
      whose relevant variables don't belong
      to LoopDef
}
```

Optimize Loops -Identifying Invariant Expressions

- Example

LoopDef{}

```
for I = 1 to 100         → {A, J, K, I}
  for J = 1 to 100       → {A, J, K}
    for K = 1 to 100     → {A, K}
      A[I][J][K] = (I*J)*K
```

Optimize Loops -Identifying Invariant Expressions

- Example

LoopUse{}

```
for I = 1 to 100            → {}
   for J = 1 to 100         → {I}
      for K = 1 to 100      → {I,J}
         A[I][J][K] = (I*J)*K
```

Optimize Loops -Identifying Invariant Expressions

- Example

Invariant Expressions

```
for I = 1 to 100
   for J = 1 to 100
      for K = 1 to 100          → { I*J, Addr(A[i][j])}
         A[I][J][K] = (I*J)*K
```

For an array access, `A[m] => Addr(A) + m`

For 3D array above*, `Addr(A[I][J][K])` =

`Addr(A)+(I*10000)-10000+(J*100)-100+K-1`

*Assuming row-major ordering of storage

Optimize Loops -Identifying Invariant Expressions

- Example

Invariant Expressions

```
for I = 1 to 100
   for J = 1 to 100        ---->   { Addr(A[i]) }
      for K = 1 to 100
         A[I][J][K] = (I*J)*K
```

For an array access, A[m] => Addr(A) + m

For 3D array above*, Addr(A[I][J][K]) =

Addr(A)+(I*10000)-10000+(J*100)-100+K-1

*Assuming row-major ordering of storage

Optimize Loops -Factoring Invariant Expressions

- Move the invariant expressions identified

```
Factor_Invariants(Loop L) {
   Mark_Invariants(L);
   foreach expression E marked an invariant:
      1. Create a temporary T
      2. Replace each occurrence of E in L with T
      3. Insert T:=E in L's header code
         // If loop is known to execute at least once,
         insert T:=E before LOOP:
}
```

Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
   for J = 1 to 100
      for K = 1 to 100
         A[I][J][K] = (I*J)*K        //Invariant Expressions
```

Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
   for J = 1 to 100
      temp1=A[I][J]
      temp2=I*J
      for K = 1 to 100
         temp1[K] = temp2*K
```

Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
   temp3=A[I]
   for J = 1 to 100
      temp1=temp3[J]
      temp2=I*J
      for K = 1 to 100
         temp1[K] = temp2*K
```

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case I: We can move `t = a op b` if the statement dominates all loop exits where `t` is live

A node bb1 dominates node bb2 if all paths to bb2 must go through bb1

```
for (...) {
   if(*)
      a = 100
}
c=a
```

Cannot move a=100 because it does not dominate c=a i.e. there is one path (when if condition is false) c=a can be executed /'reached' without going to a=100

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

  Case II: We can move `t = a op b` if there is only one definition of `t` in the loop

```
for (...) {
   if(*)
      a = 100
   else
      a = 200
}
```

Multiple definition of a

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case III: We can move `t = a op b` if `t` is not defined before the loop, where the definition reaches t's use after the loop

```
a=5
for (...) {
   a = 4+b
}
c=a
```

Definition of a in a=5 reaches `c=a`, which is defined after the loop

Optimize Loops –Strength Reduction

- Like strength reduction in peephole optimization
  - E.g. replace `a*2` with `a<<1`
- Applies to uses of induction variable in loops
  - Basic induction variable (`I`) – only definition within the loop is of the form `I = I ± S`, (S is loop invariant)

    `I` *usually determines number of iterations*
  - Mutual induction variable (`J`) – defined within the loop, its value is linear function of other induction variable, `I`, such that

    `J = I * C ± D`   (C, D are loop invariants)

Optimize Loops –Strength Reduction

```
strength_reduce(Loop L) {
   Mark_Invariants(L);
   foreach expression E of the form I*C+D where I is
L's loop index and C and D are loop invariants
      1. Create a temporary T
      2. Replace each occurrence of E in L with T
      3. Insert T:=Io*C+D, where Io is the initial value of the
         induction variable, immediately before L
      4. Insert T:=T+S*C, where S is the step size, at the end of
         L's body
      }
```

Optimize Loops –Strength Reduction

- Suppose induction variable `I` takes on values I_o, I_o+S, I_o+2S, I_o+3S... in iterations 1, 2, 3, 4, and so on…
- Then, in consecutive iterations, Expression `I*C+D` takes on values

$$I_o*C+D$$
$$(I_o+S)*C+D = I_o*C+S*C+D$$
$$(I_o+2S)*C+D = I_o*C+2S*C+D$$
$$\dots \qquad \dots$$

- The expression changes by a constant `S*C`
- Therefore, we have replaced a * and + with a +

Optimize Loops – Strength Reduction

- Example (Applying to innermost loop)

```
for I = 1 to 100
   for J = 1 to 100
      for K = 1 to 100
         A[I][J][K] = (I*J)*K
```

```
for I=1 to 100
   temp3=Addr(A[i])
   for J=1 to 100
      temp1=Addr(temp3(J))
      temp2=I*J
      for K=1 to 100
         temp1[K]=temp2*K
```

```
. . .
temp2=I*J
temp4=temp2
for K=1 to 100
   temp1[K]=temp4
   temp4=temp4+temp2
```

```
//S=1
//C=temp2
```

Optimize Loops – Strength Reduction

- Exercise (Apply to intermediate loop)

```
for I=1 to 100
   temp3=Addr(A[i])
   for J=1 to 100
      temp1=Addr(temp3(J))
      temp2=I*J
      for K=1 to 100
         temp1[K]=temp2*K
```

```
. . .
temp2=I*J
temp4=temp2
for K=1 to 100
   temp1[K]=temp4
   temp4=temp4+temp2
```

```
// Induction var = J
// S = 1
// Expression = I * J
```

Optimize Loops – Strength Reduction

- Exercise (Apply to intermediate loop)

```
. . .
temp5=I
for J=1 to 100
    temp1=Addr(temp3(J))
    temp2=temp5
    temp4=temp2
    for K=1 to 100
        temp1[K]=temp4
        temp4=temp4+temp2
    temp5=temp5+I
```

… ➡ …

Optimize Loops – Strength Reduction

- Further strength reduction possible?

```
for I=1 to 100
   temp3=Addr(A[i])
   temp5=I
   for J=1 to 100
      temp1=Addr(temp3(J))
      temp2=temp5
      temp4=temp2
      for K=1 to 100
         temp1[K]=temp4
         temp4=temp4+temp2
      temp5=temp5+I
```

Optimize Loops – Loop Unrolling

- Modifying induction variable in each iteration can be expensive
- Can instead *unroll* loops and perform multiple iterations for each increment of the induction variable
- What are the advantages and disadvantages?

```
for (i = 0; i < N; i++)
  A[i] = ...
```

↓ Unroll by factor of 4

```
for (i = 0; i < N; i += 4)
  A[i] = ...
  A[i+1] = ...
  A[i+2] = ...
  A[i+3] = ...
```

Optimize Loops - Summary

- Low level optimization
  - Moving code around in a single loop
  - Examples: loop invariant code motion, strength reduction, loop unrolling
- High level optimization
  - Restructuring loops, often affects multiple loops
  - Examples: loop fusion, loop interchange, loop tiling