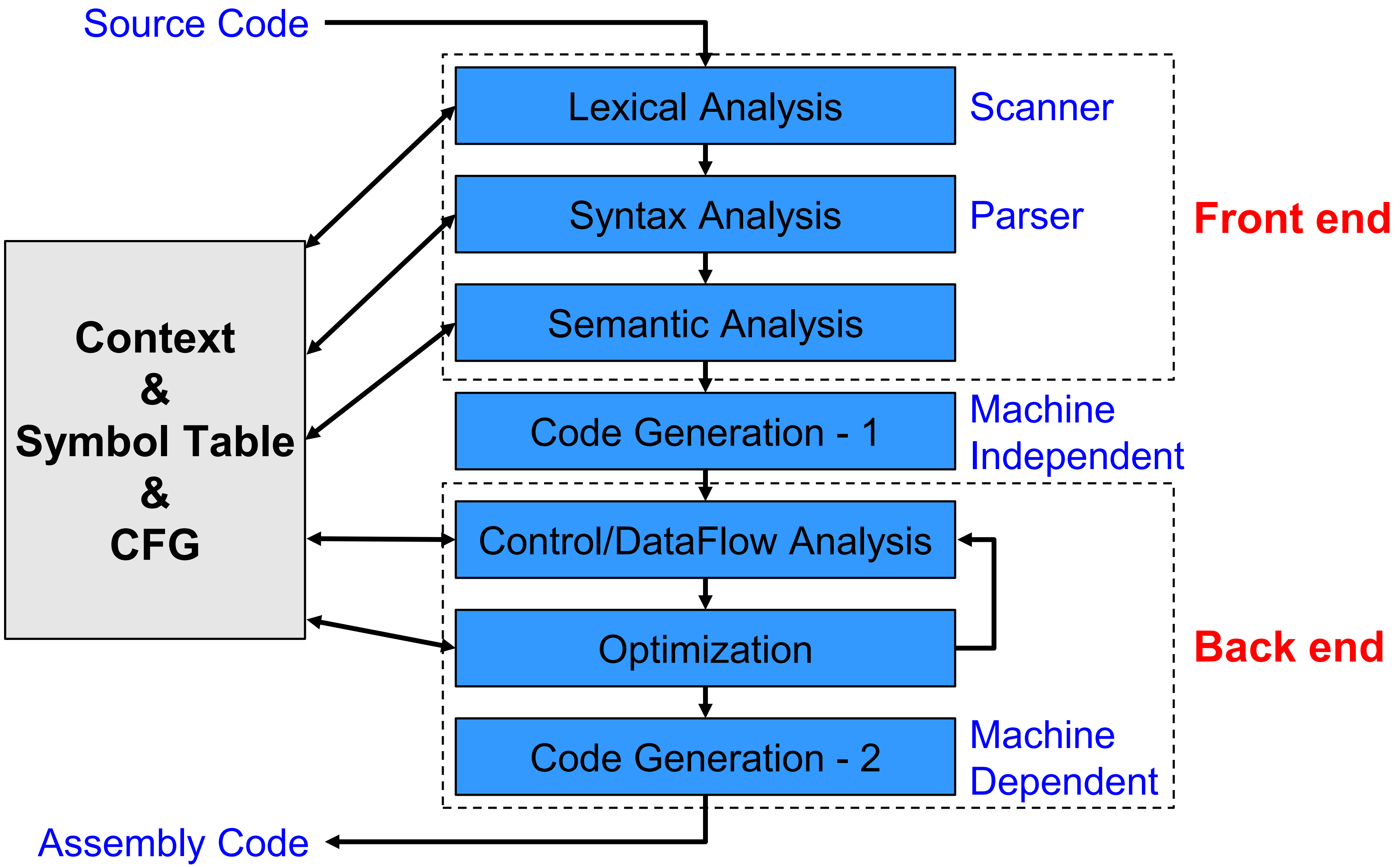
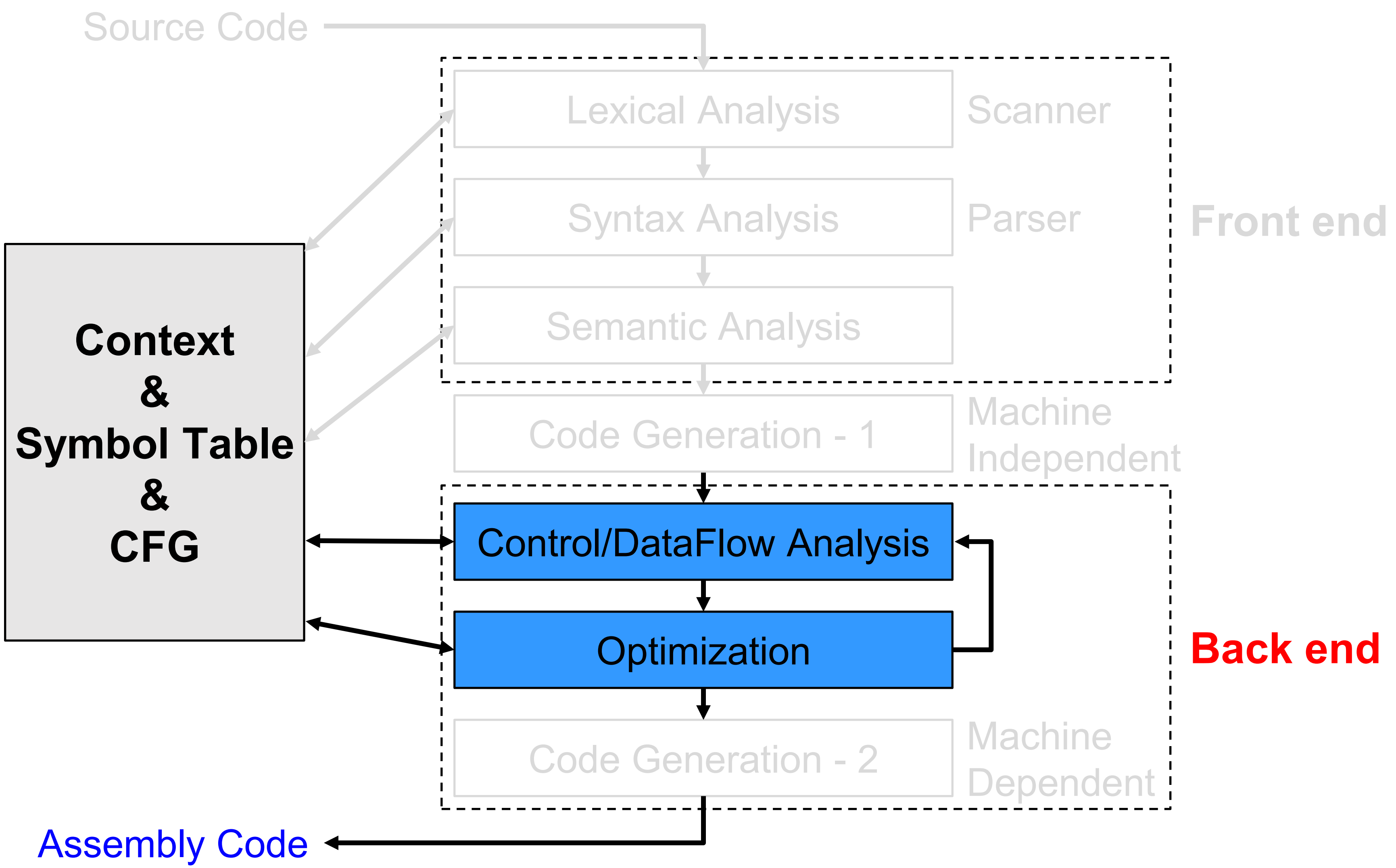
9. Control Flow Analysis

2025 Fall
Hunjun Lee
Hanyang University

Compiler Overview



What You Will Learn



Optimization Types

- **Dataflow optimization**
  - Dataflow is about how a code manipulates the data
  - Can remove redundant computations or simplify computations

- **Control flow optimization**
  - Control flow is about the order of code execution (e.g., branching structure)
  - Can remove unreachable code, change code for reduced computations, …

Control Flow Analysis

- **Determining properties of the program branch structure**

- **The compiler identifies properties that exist regardless of the run-time branch conditions**
  - Run-time optimizations are done at run-time, we focus on static properties at compile time

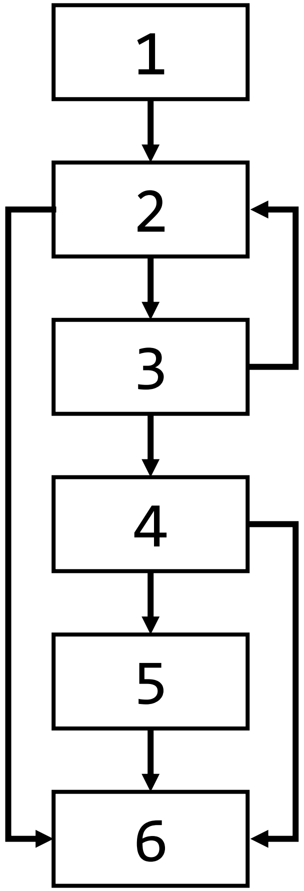
- **Use CFGs and there are rooms to optimize the efficiency of control flow structure**

Recall: CFG

- **Control transfer = branch (taken or fall-through)**
- **Control flow**
  - Branching behavior of an application
  - What sequences of instructions can be executed
- **Execution → Dynamic control flow**
  - Direction of a particular instance of a branch
  - Predict, speculate, etc.
- **Compiler → Static control flow**
  - Not executing the program
  - Input not known, so what could happen, worst case

CFG Exercise

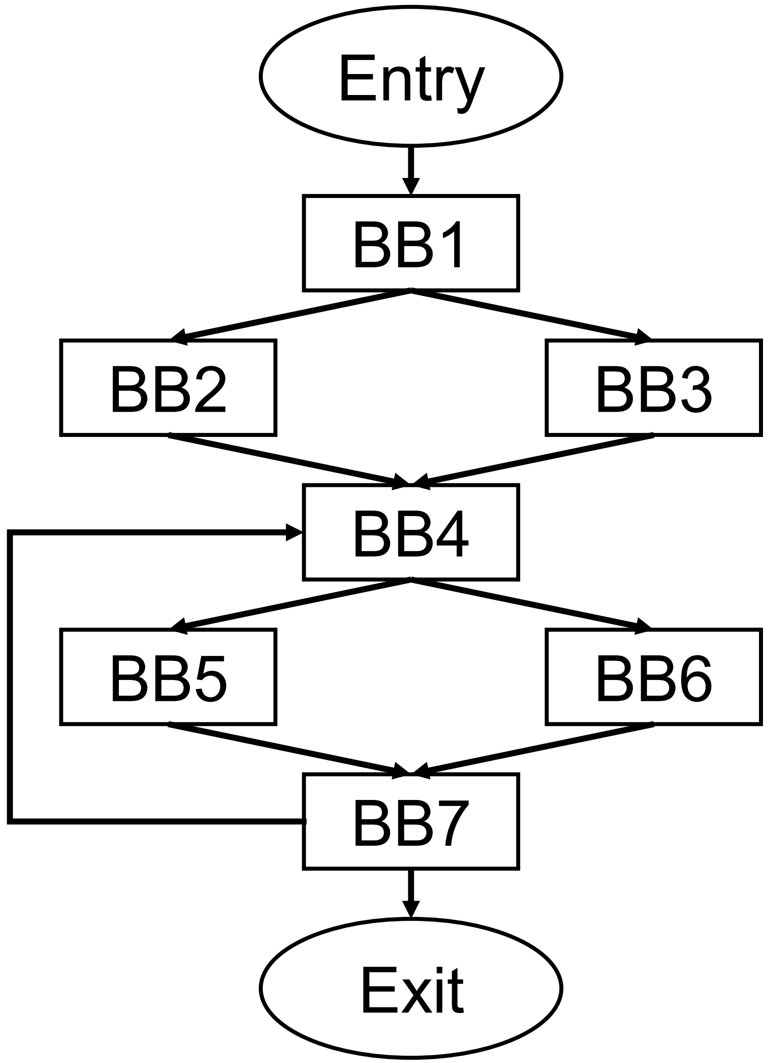
```
------------------------------
1:  r7 = load r8
------------------------------
2:  r1 = r1 + r2
3:  beq r1, 0, L10
------------------------------
4:  r4 = r5 * r6
5:  r1 = r1 + 1
6:  beq r1 100 L2
------------------------------
7:  beq r2 100 L10
------------------------------
8:  r5 = r9 + 1
9:  r7 = r7 & 3
------------------------------
10: r9 = load r3
11: store(r9, r1)
------------------------------
```



Dominator

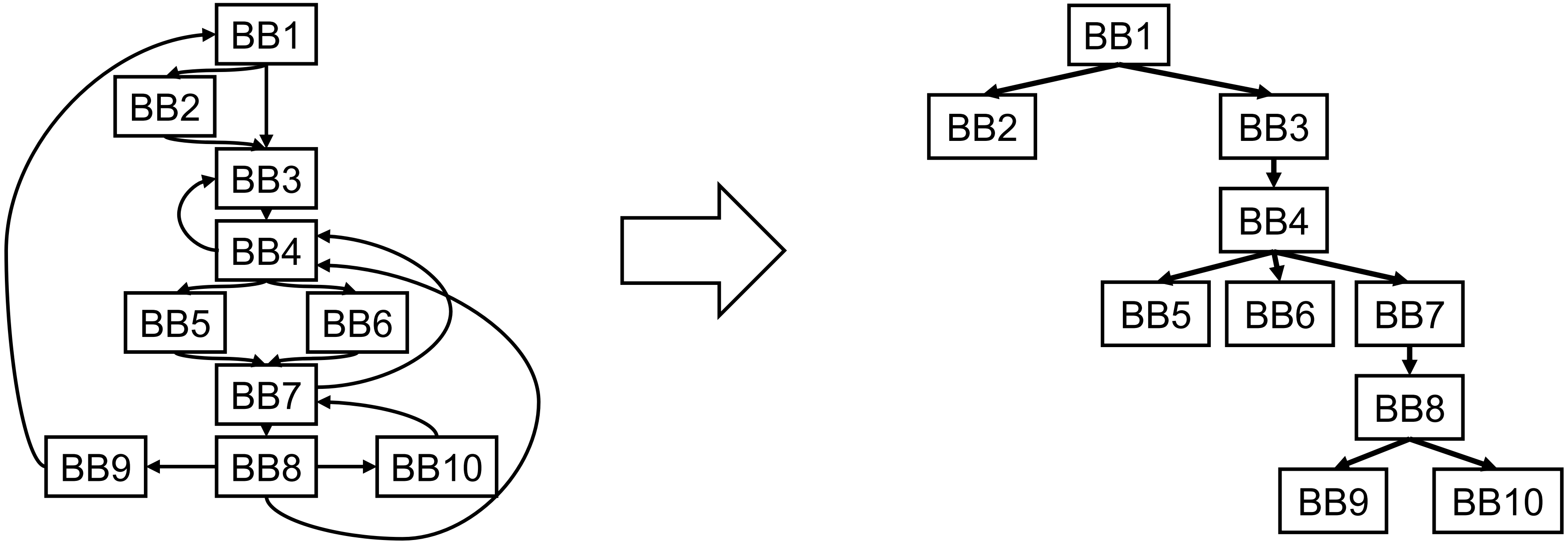
- **Dominator: Given a CFG(V, E, Entry, Exit), a node x dominates a node y, if every path from the Entry block to y contains x**

- **There are three properties of dominators**
  - Each BB dominates itself
  - If x dominates y, and y dominates z, then x dominates z
  - If x and y dominates z, then either x dominates y or y dominates x (* BB2 does not dominate BB4)



Dominator Tree

- **This implies that we can describe the dominator relation in the form of a tree (i.e., dominator tree)**
  - The initial node is a root
  - The parent node dominates all the descendants in the tree

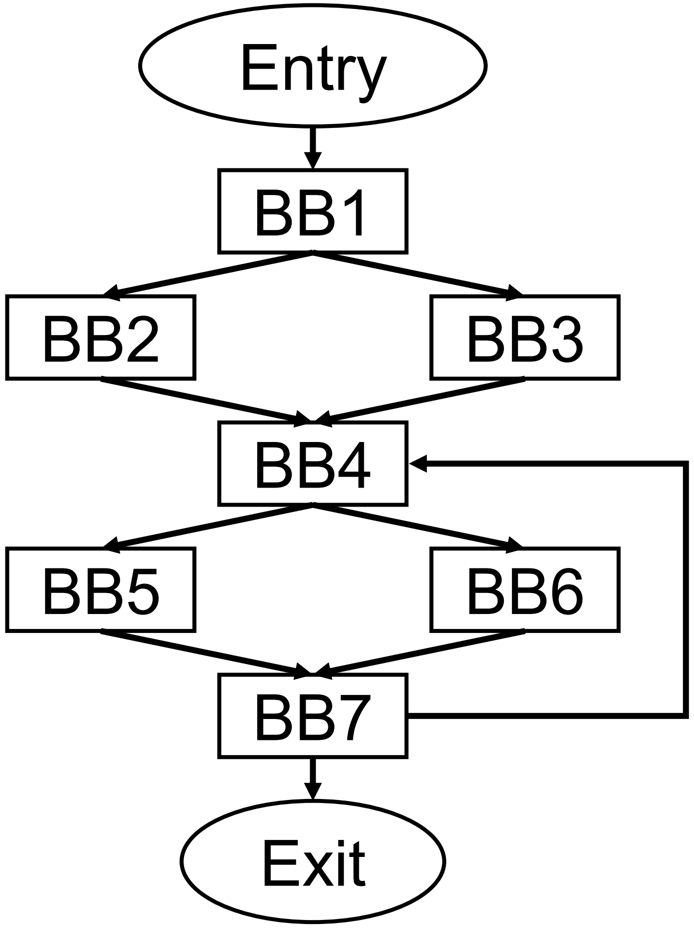


Dominator Analysis

- **dom(BB_i): indicates a set of BBs that dominate BB_i**
- **Initialize (Dom(entry) = entry and Dom(else) = all nodes) and iteratively compute the following**

```
while change {
  change = false;
  for BB in BBs (except entry) {
    tmp(BB) = BB + {Intersection of all predecessors (BBp) of
dom(BBp)}
    if (tmp(BB) != dom(BB)) {
      dom(BB) = tmp(BB);
      change = true; }}}
```

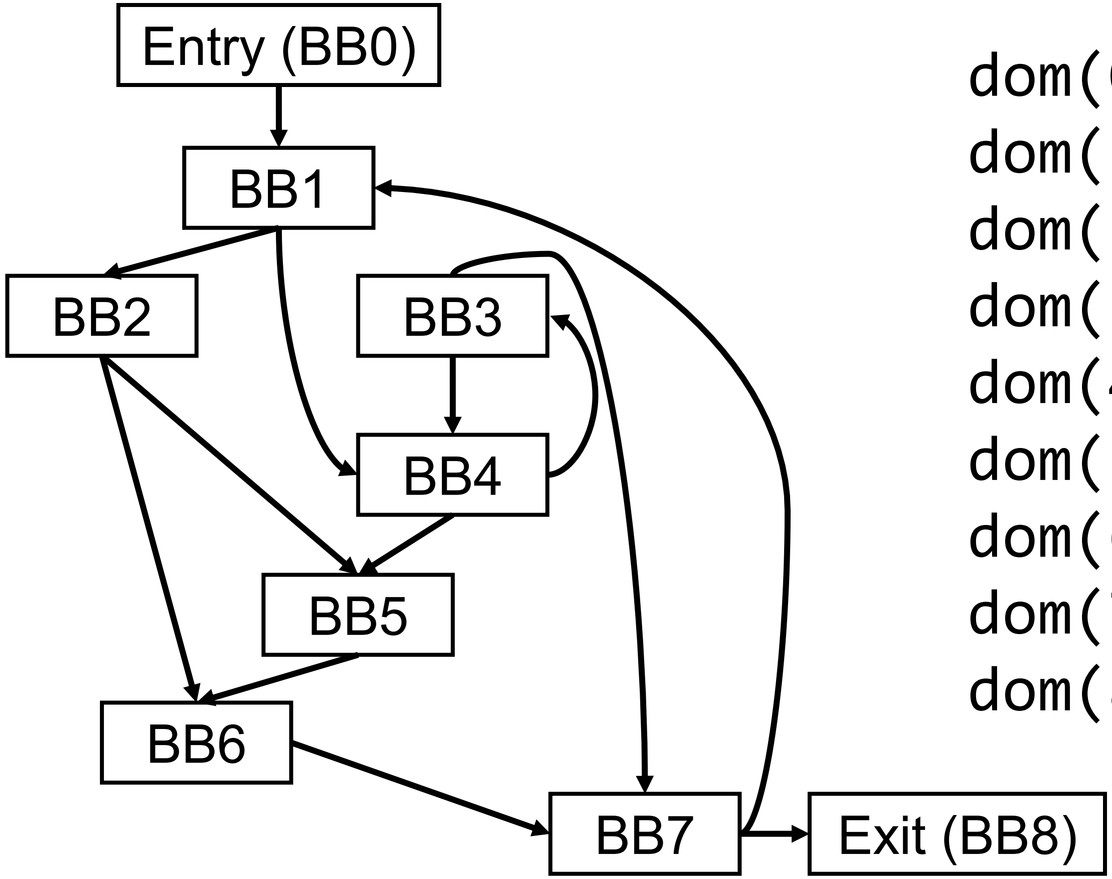
Dominator Analysis Example




```
dom(0): {0}      → {0}
dom(1): {0~8}    → {0,1}
dom(2): {0~8}    → {0,1,2}
dom(3): {0~8}    → {0,1,3}
dom(4): {0~8}    → {0,1,4}
dom(5): {0~8}    → {0,1,4,5}
dom(6): {0~8}    → {0,1,4,6}
dom(7): {0~8}    → {0,1,4,7}
dom(8): {0~8}    → {0,1,4,7,8}
```

Class Exercise

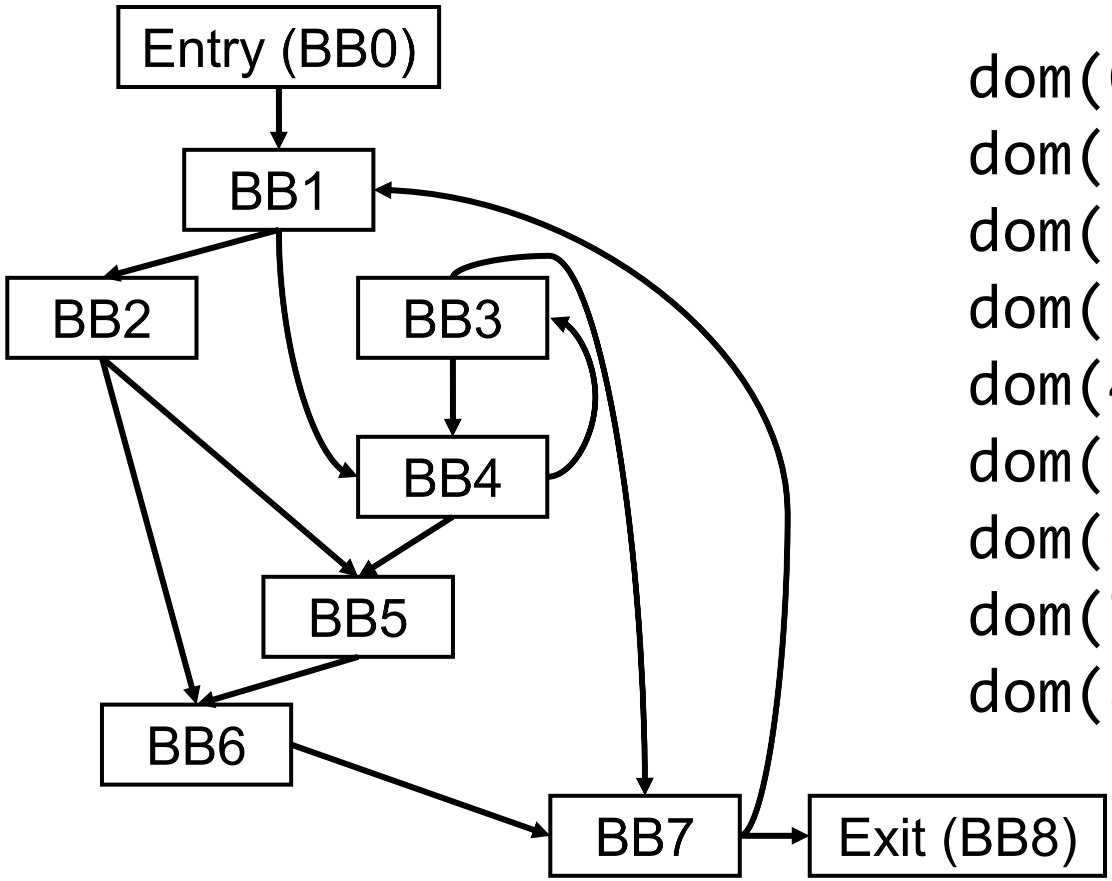
- **Identify the dom(BBi) of all BBs and draw dominator tree**



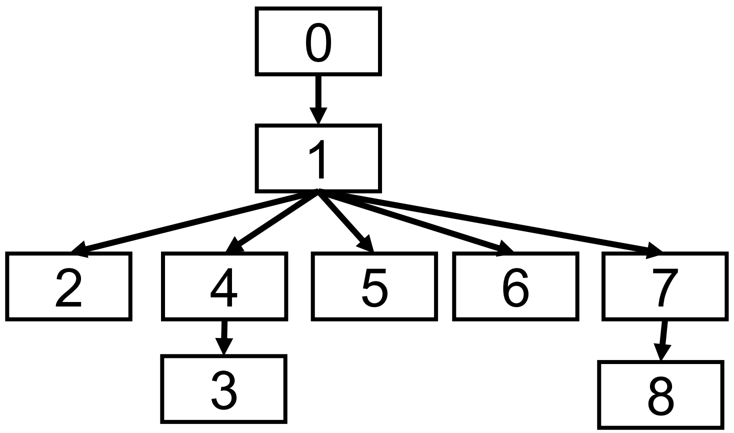

```
dom(0): {0}   → {0}           → {0}
dom(1): {0~8} → {0,1}         → {0,1}
dom(2): {0~8} → {0,1,2}       → {0,1,2}
dom(3): {0~8} → {0~8}         → {0,1,3,4}
dom(4): {0~8} → {0,1,4}       → {0,1,4}
dom(5): {0~8} → {0,1,5}       → {0,1,5}
dom(6): {0~8} → {0,1,6}       → {0,1,6}
dom(7): {0~8} → {0,1,6,7}     → {0,1,7}
dom(8): {0~8} → {0,1,6,7,8}   → {0,1,7,8}
```

Class Exercise

- **Identify the dom(BBi) of all BBs and draw dominator tree**




```
dom(0): {0}
dom(1): {0,1}
dom(2): {0,1,2}
dom(3): {0,1,3,4}
dom(4): {0,1,4}
dom(5): {0,1,5}
dom(6): {0,1,6}
dom(7): {0,1,7}
dom(8): {0,1,7,8}
```



Natural Loops

- **There are two properties of a natural loop**
  - There exists a single-entry point called the header which **dominates all blocks in the loop**
  - A backedge is an edge whose target dominates source: a back edge must be a part of at least one loop
  - The natural loop of a backedge is: smallest set of nodes that includes the target and source of the backedge, and has no predecessors outside the set except for the predecessors of the header

Algorithms to Find Natural Loops

- **Step #1) Find the dominator relationship in a CFG**

- **Step #2) Identify the back edges**

- **Step #3) Find the natural loop associated with each back edge**

Step #1) Dominator Relationship

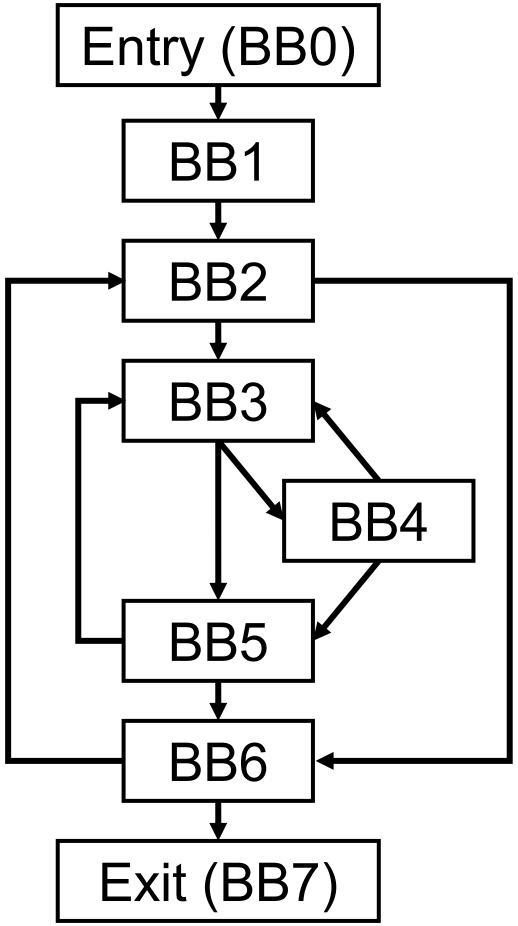
- **We have covered it before!**

Step #2) Finding Back Edges

- **Perform a depth-first search over the CFG and identify backward edges**
  - backward edges indicate the edges from the descendants to ancestors

- **For each backward edge, check if the destination is the dominator of the list**

Back Edge Example



dom(1) = {0,1}
dom(2) = {0,1,2}
dom(3) = {0,1,2,3}
dom(4) = {0,1,2,3,4}
dom(5) = {0,1,2,3,5}
dom(6) = {0,1,2,6}

BE = target dominates source

E → 1 : No
1 → 2 : No
2 → 3 : No
2 → 6 : No
3 → 4 : No
3 → 5 : No
4 → 3 : Yes
4 → 5 : No
5 → 3 : Yes
5 → 6 : No
6 → 2 : Yes
6 → 7 : No

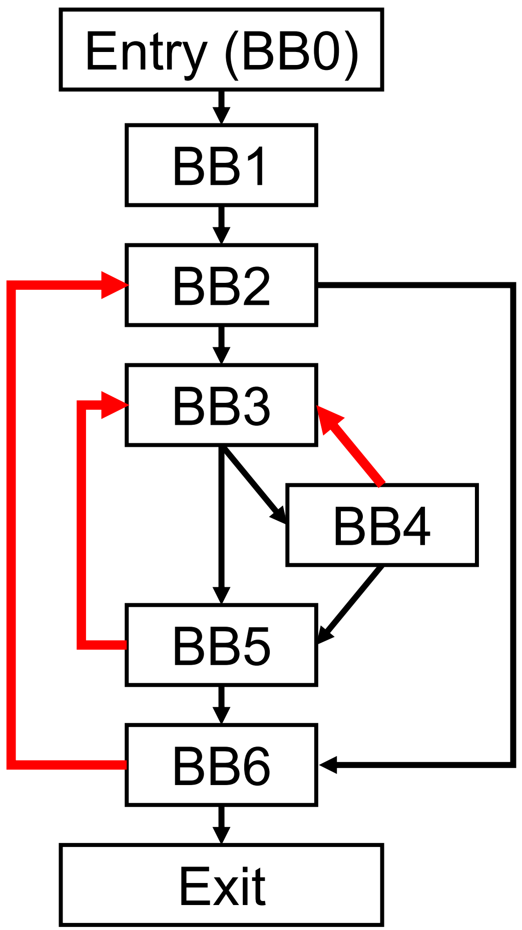
Step #3) Find Natural Loops

- **We should identify the loops for the identifies back edges (x → y)**
  - The destination node becomes the loop header (y)
  - Delete the destination node and its in/out edges
  - Find nodes that reach the source node
  - These nodes plus the source and destination nodes comprise the natural loop (Loop BB)

- **For optimization, merge loops with the same loop header**
  - Loop Backedge = Loop Backedge1 + Loop Backedge2
  - Loop BB = Loop BB1 + Loop BB2

- **Important property is that the header dominates all Loop BB**

Loop Detection Example



dom(1) = {0,1}
dom(2) = {0,1,2}
dom(3) = {0,1,2,3}
dom(4) = {0,1,2,3,4}
dom(5) = {0,1,2,3,5}
dom(6) = {0,1,2,6}

Loop1:
- Loop BB = {**2**,3,4,5,6}
- Loop Backedge = 6 → 2

Loop2:
- Loop BB = {**3**,4}
- Loop Backedge = 4 → 3

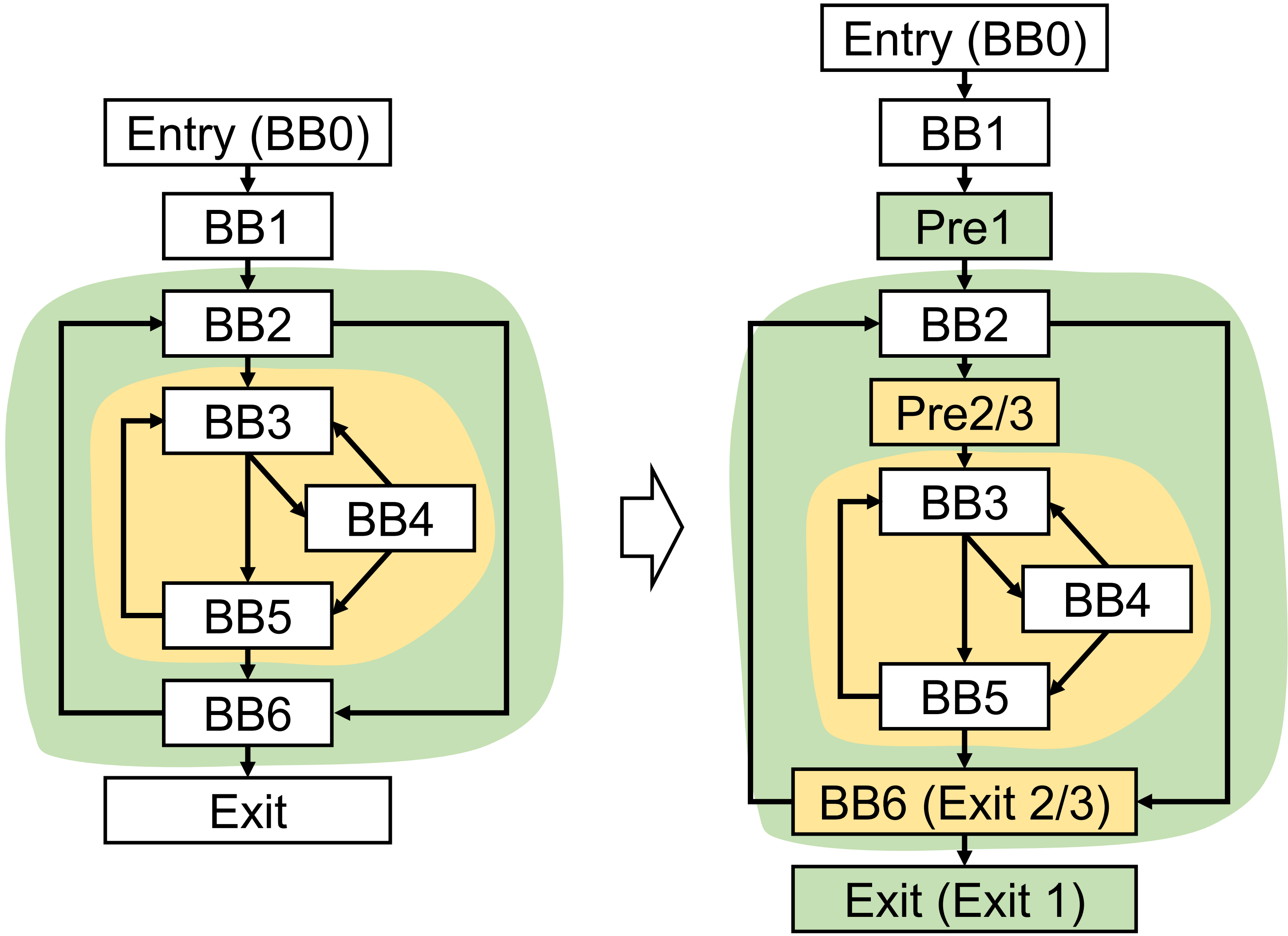
Loop3:
- Loop BB = {**3**,4,5}
- Loop Backedge = 5 → 3

Loop2/3:
- Loop BB = {**3**,4,5}
- Loop Backedge = 4 → 3, 5 → 3

Important Concepts in a Loop

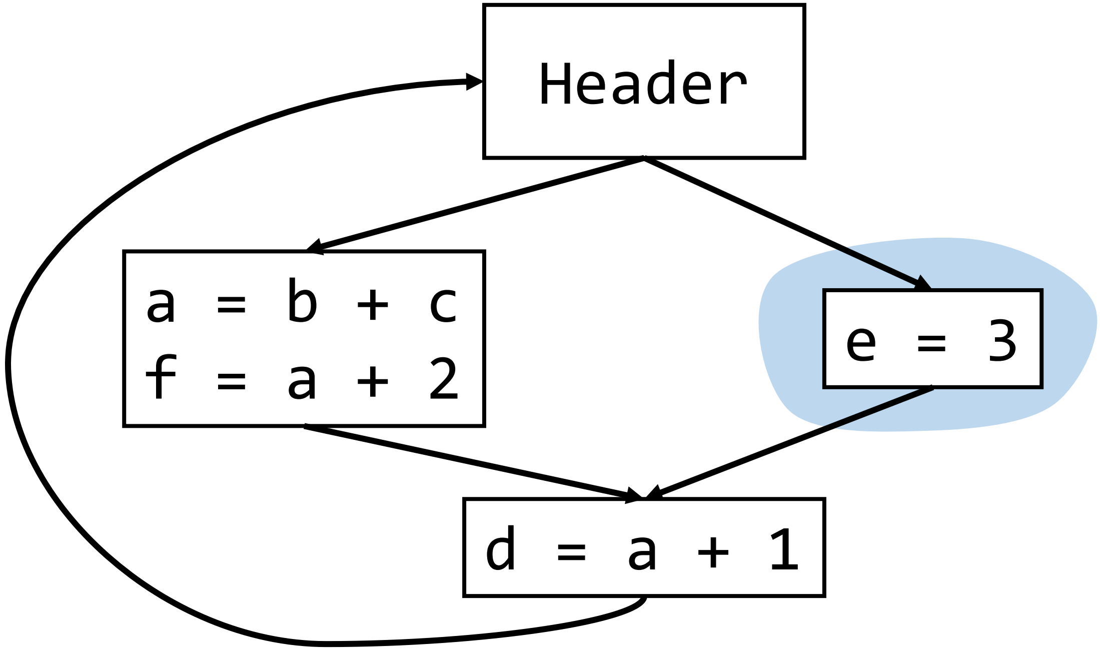
- **Header and Loop BB**
- **Back Edges**
- **Exit Edges**
  - For each Loop BB, examine each outgoing edge
  - If the destination node of the edge is not in Loop BB, then it is an exit edge
- **Preheader (Preloop)**
  - generate a new block before the header (it falls through to header)
  - Whenever a loop is executed, preheader executed
  - However, it is not executed during the iteration
  - All edges entering header (except back edge) retarget to preheader

Exit BB and Preheader Example



Loop-Invariant Computation

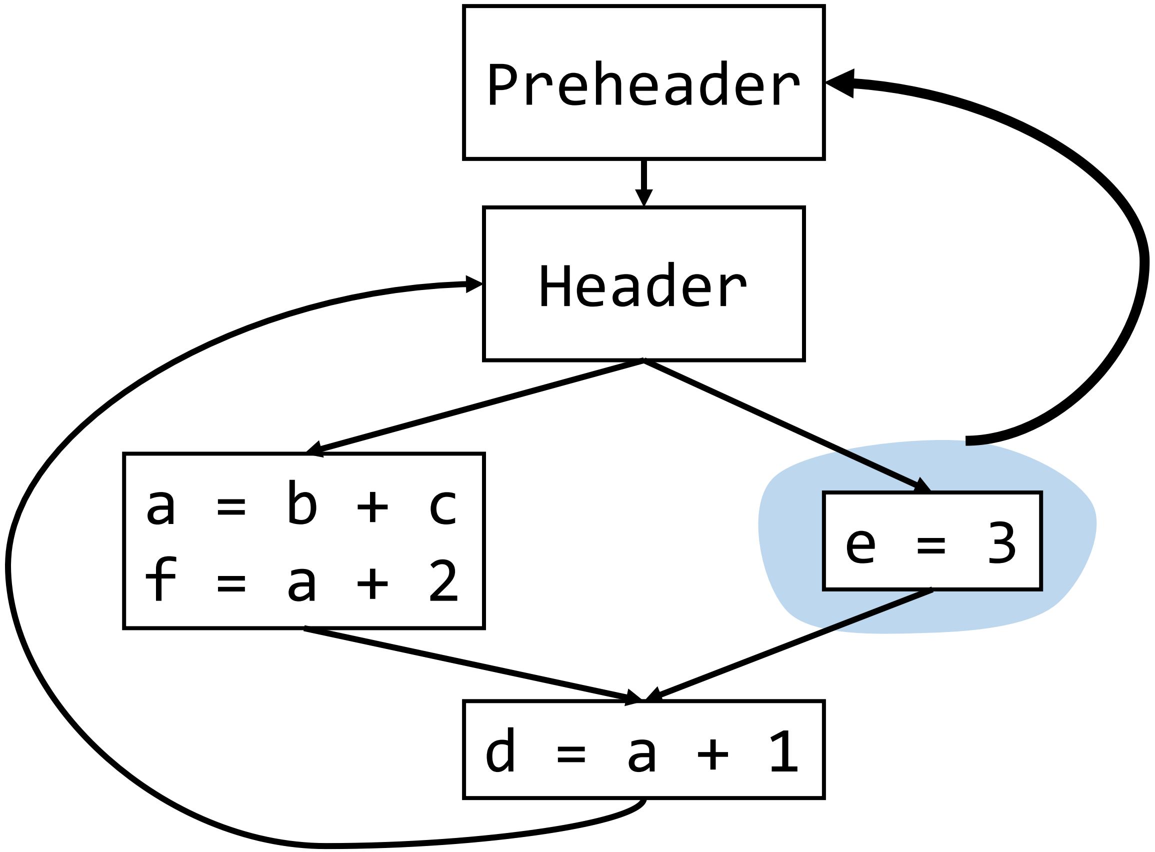
- There are set of computations whose value do not change as long as the control stays within the loop



e = 3 for the entire loop

Loop-Invariant Code Motion (LICM)

- **We can move the loop invariant code to the preheader**



We can move the invariant code to the preheader

LICM Formulation

- **The LICM should solve two problems**
  - Identifying loop-invariant codes
  - Identifying which loop-invariant codes can move to the pre-header?

- **Observations:**
  - Loop invariant: all operands are defined outside loop or are defined by loop invariants (or by constants)
  - Code motion: not all invariant statements can be moved to the preheader

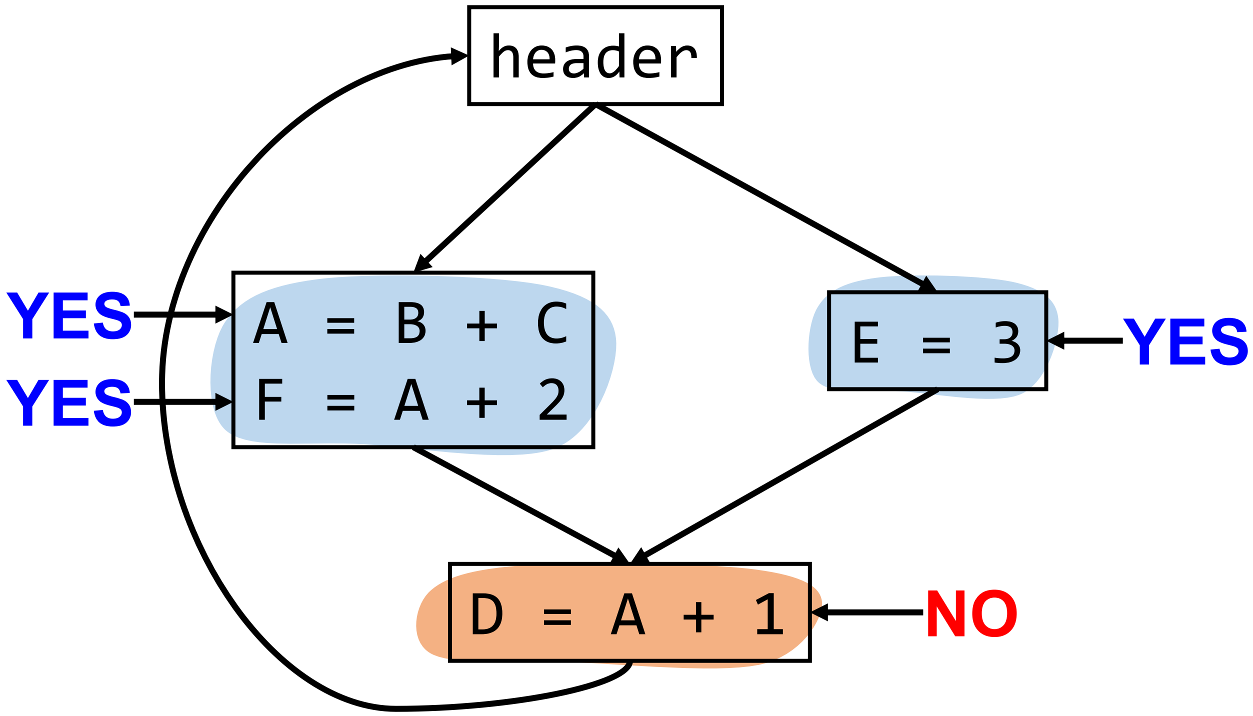
Detecting Loop Invariant

- **Mark A = B + C as invariant if**
  - All reaching definitions of B are outside of the loop, or there is exactly “one reaching definition” for B and it is from a loop-invariant statement inside the loop
  - Check similarly for C

- **Repeat until there is no change in the set of loop-invariant statements**

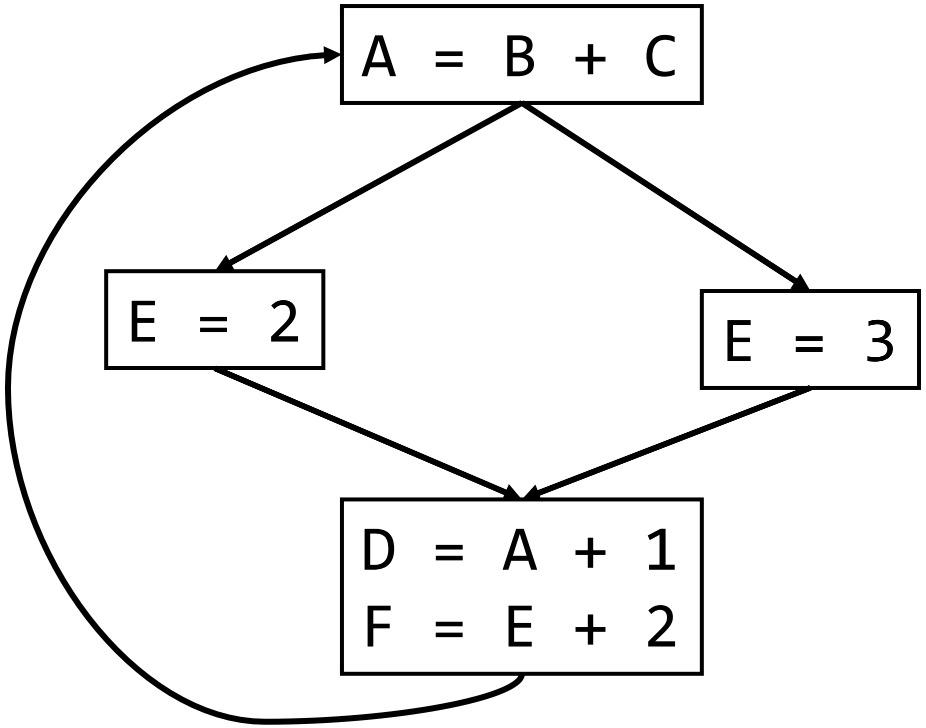
Loop Invariant Example - 1

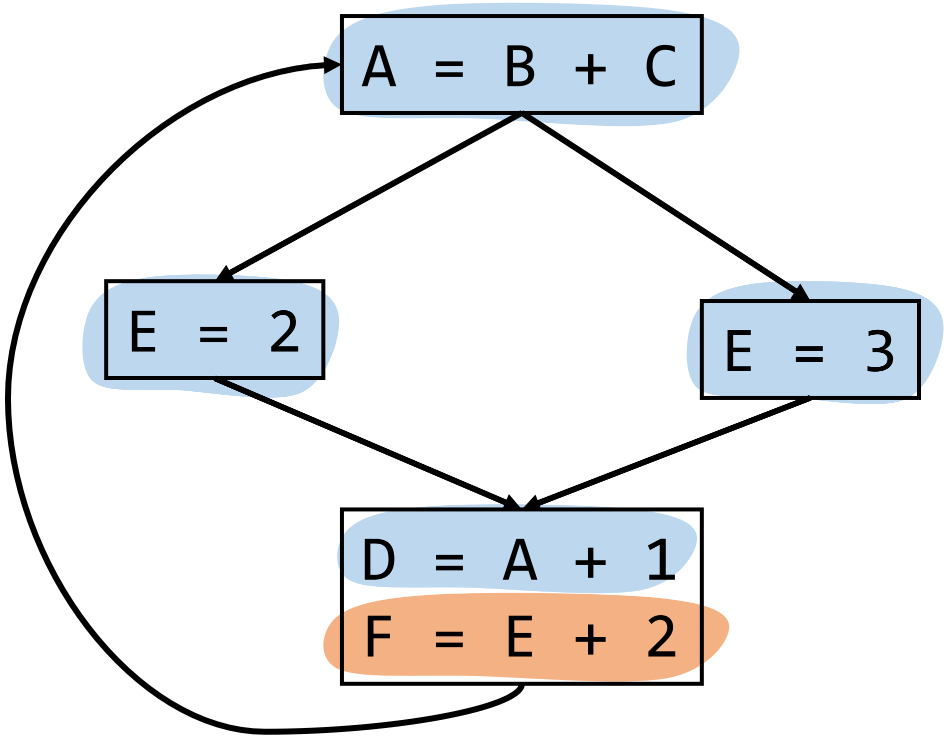
- **Identify loop invariant statements**



Loop Invariant Exercise

- Identify loop invariant statements





Conditions for Code Motion - 1

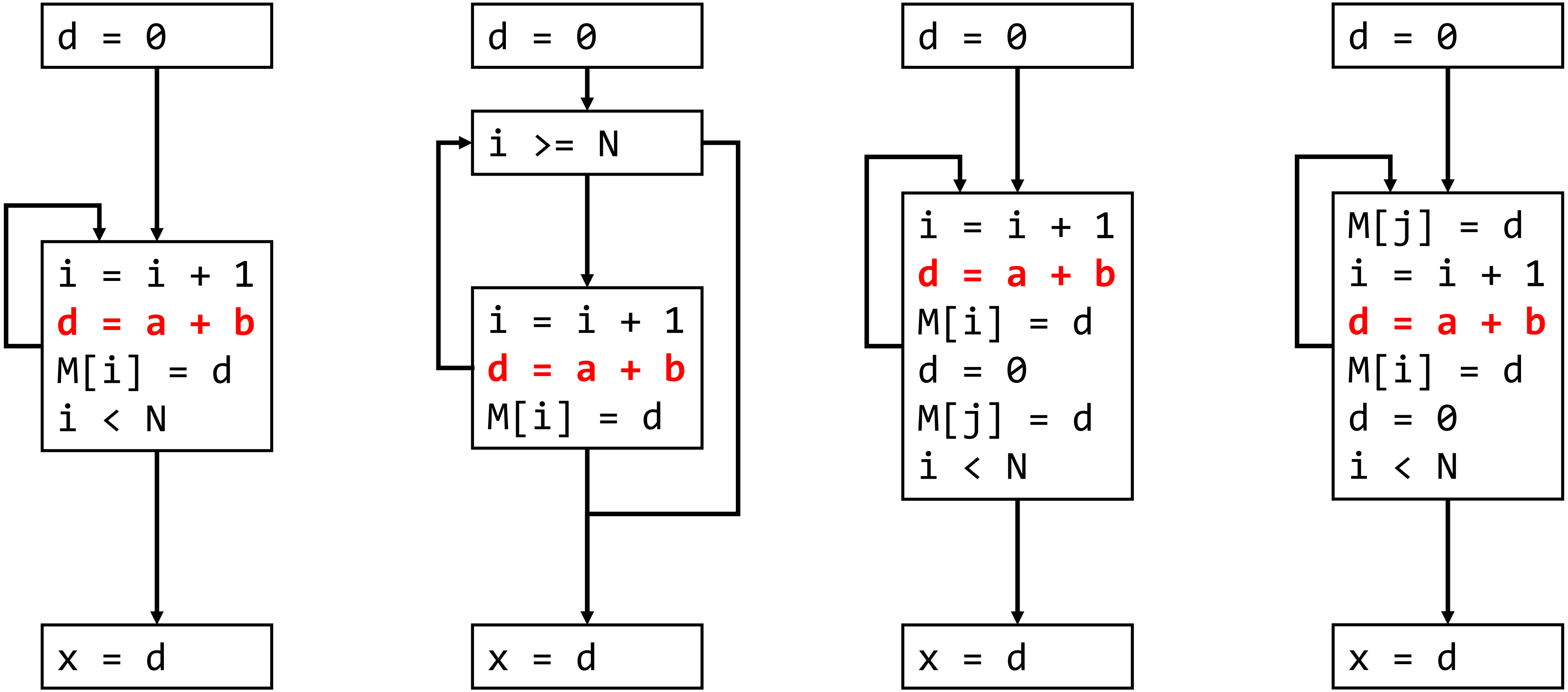
- **We do not move all the invariant codes!**

- **In a naïve form, there are three basic conditions for code motion (when moving the following code)** `d = a + b`
  - a, b are numerical constants
  - a, b are defined outside the loop
  - a, b are loop invariants

- **Is this the only condition?**

Conditions for Code Motion - 2

- Can we move code `d = a + b` in the examples below?

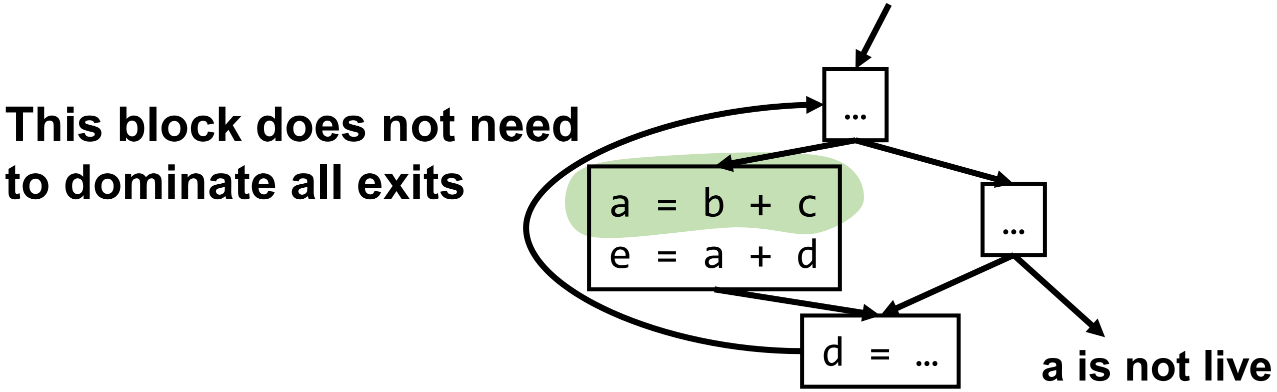


Conditions for Code Motion - 3

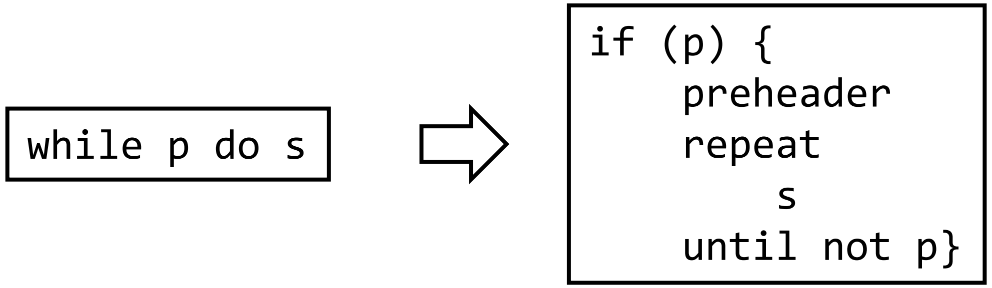
- **There are additional conditions for moving** `d = a + b`
  - The computation dominates all loop exits (second example)
  - d is defined once in the loop (third example)
  - d is not live before the loop (fourth example)

More Aggressive Optimizations

- **Liveness-aware relaxation (Example 2)**
  - We can relax the dominance relation to the exits as long as the variable is not live after we exit the loop

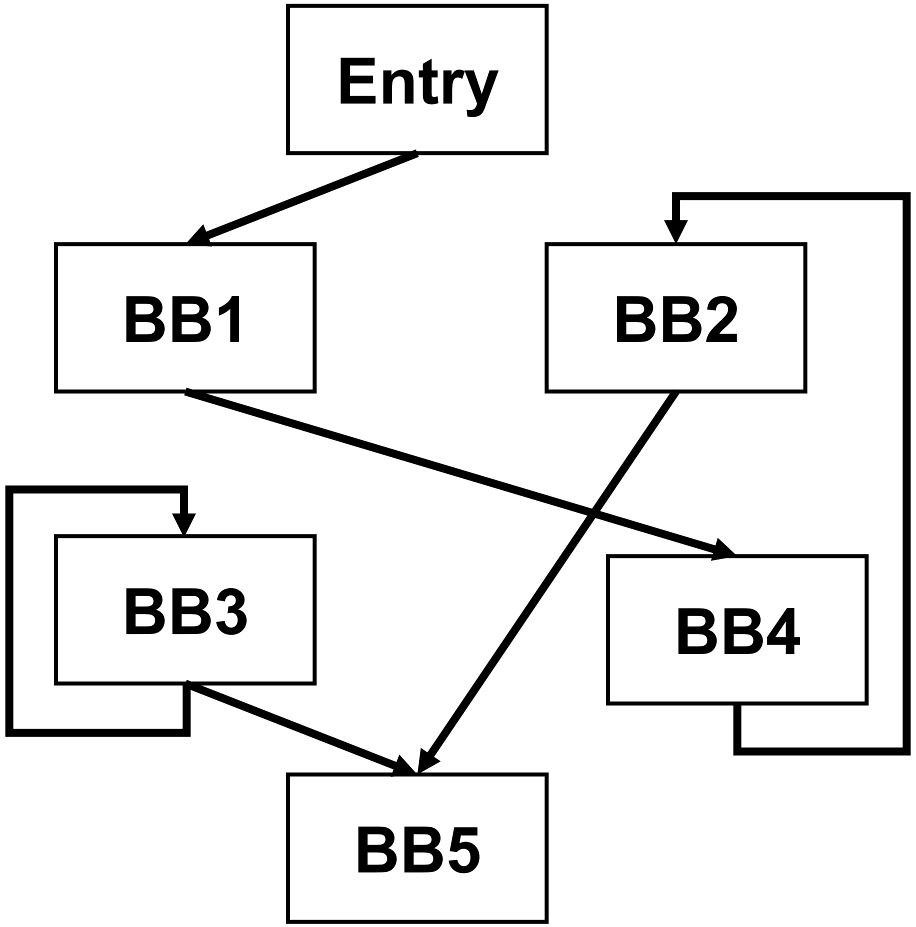


- **Landing Pads (Example 2)**
  - We can maximize code motion opportunities by setting a condition before entering the loop



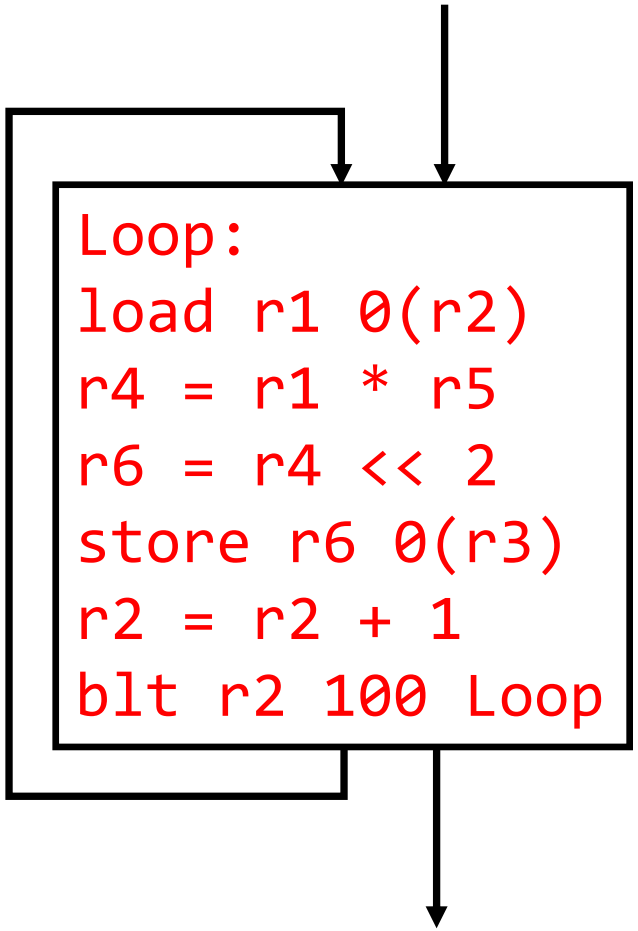
Unreachable Code Elimination

```
//Mark entry BB visited
entry.visited = True
visit = {entry}
while not visit.empty():
      curr = visit.pop()
      for BB in curr.successor:
            if BB.visited == False:
                  BB.visited = True
                  visit += BB
}
```



Loop Unrolling

- **The most renowned control flow optimization**
- **Unrolling potentially increases instruction level parallelism**
  - Minimize branch instructions
  - Enable highly parallel execution
  - Reduced dependency
- **Three variants exist**
  - **Type 1:** Unroll multiple of known trip count
  - **Type 2:** Unroll with remainder loop
  - **Type 3:** While loop unroll



Loop Unroll – Type 1

- **Case when we know everything @ compile time**
  - Loop variable, increment, initial value, final count

```
Loop:
load r1 0(r2)
r4 = r1 * r5
r6 = r4 << 2
store r6 0(r3)
r2 = r2 + 1
blt r2 100 Loop
```

```
Loop:
load r1 0(r2)
r4 = r1 * r5
r6 = r4 << 2
store r6 0(r3)
r2 = r2 + 1
load r1 0(r2)
r4 = r1 * r5
r6 = r4 << 2
store r6 0(r3)
r2 = r2 + 1
blt r2 100 Loop
```

```
Loop:
load r1 0(r2)
r4 = r1 * r5
r6 = r4 << 2
store r6 0(r3)
load r1 1(r2)
r4 = r1 * r5
r6 = r4 << 2
store r6 0(r3)
r2 = r2 + 2
blt r2 100 Loop
```

Loop Unroll – Type 2

- **Case when we know some @ compile time**
  - Loop variable?, increment?, initial value?, final count?

```
Loop:
load r1 0(r2)
r4 = r1 * r5
r6 = r4 << 2
store r6 0(r3)
r2 = r2 + x
blt r2 y Loop
```

```
tc = final - initial
tc = tc / increment
rem = tc % N
fin = rem * increment
```

```
RemLoop:
load r1 0(r2)
r4 = r1 * r5
r6 = r4 << 2
store r6 0(r3)
r2 = r2 + X
blt r2 fin RemLoop
```

```
Loop:
load r1 0(r2)
r4 = r1 * r5
r6 = r4 << 2
store r6 0(r3)
load r1 x(r2)
r4 = r1 * r5
r6 = r4 << 2
store r6 0(r3)
r2 = r2 + (2*X)
blt r2 Y Loop
```

Loop Unroll – Type 3

- **Non counted loops**
  - Pointer chasing, …

```
add r0 r0 1
load r1 0(r0)
add r1 r1 1
store r1 0(r0)
bne r1 0 Loop
```

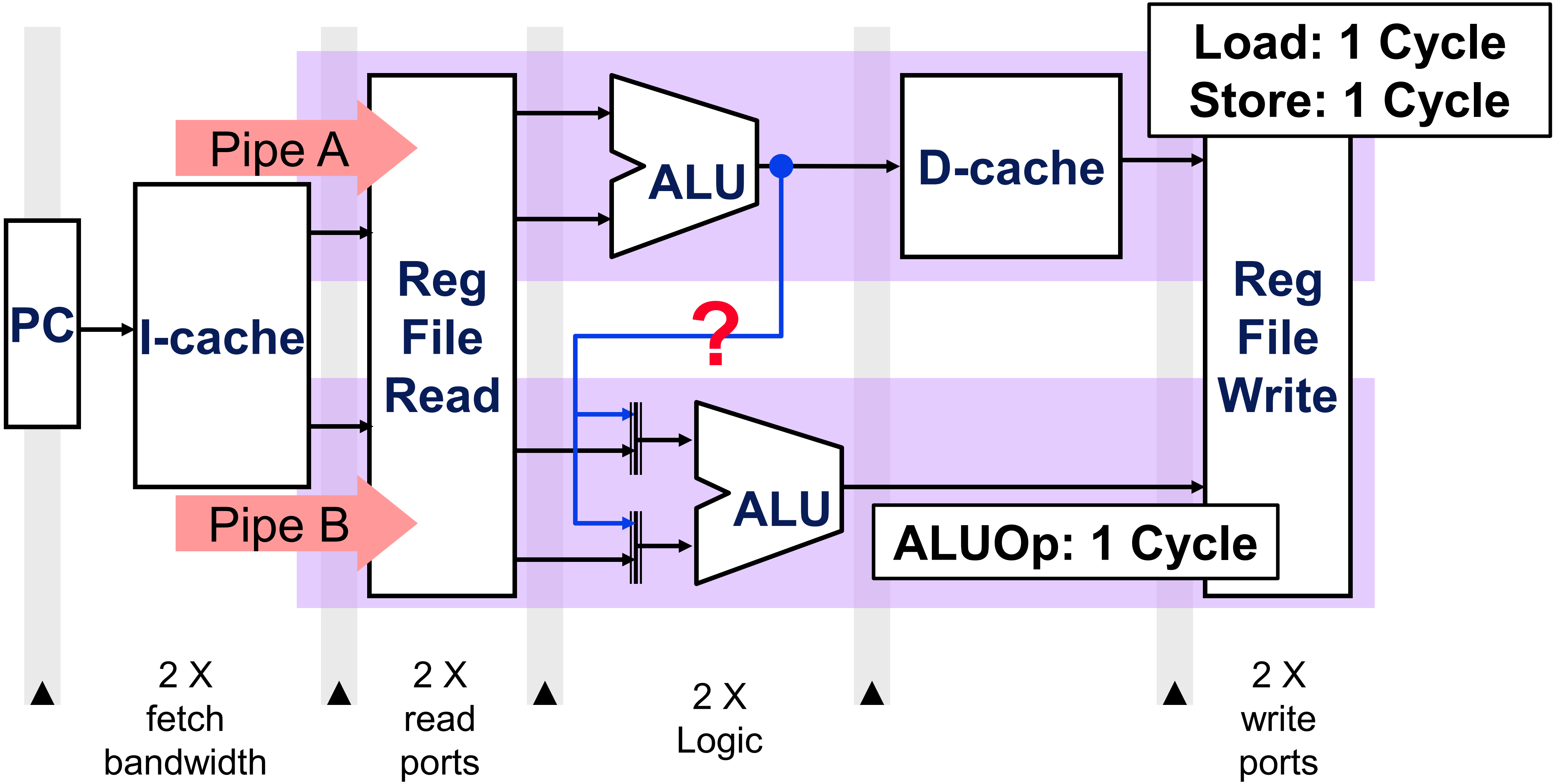
We cannot remove the branches

```
Loop:
add r0 r0 1
load r1 0(r0)
add r1 r1 1
store r1 0(r0)
beq r1 0 Exit
add r0 r0 1
load r1 0(r0)
add r1 r1 1
store r1 0(r0)
bne r1 0 Loop
Exit:
```

Reduce Data Dependency

```
Loop:
add r0 r0 1
load r1 0(r0)
add r1 r1 1
store r1 0(r0)
beq r1 0 Exit
add r2 r0 1
load r3 0(r2)
add r3 r3 1
store r3 0(r2)
add r0 r2 0
bne r1 0 Loop
Exit:
```

Superscalar CPU



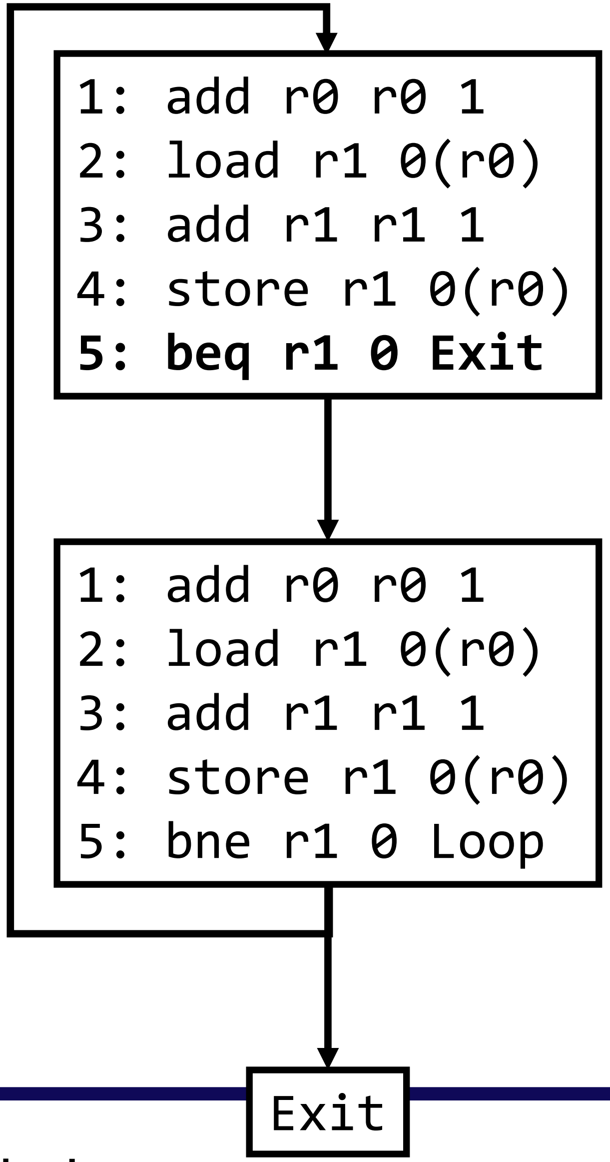
Unrolling and Instruction Scheduling - 1

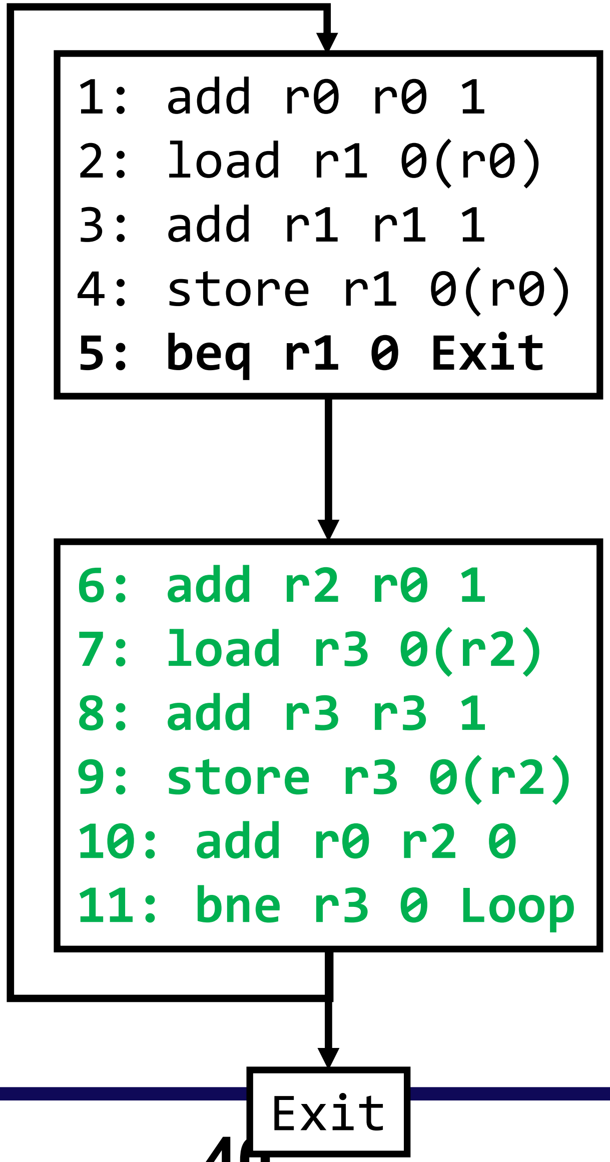
- **Assumption:**
  - **two memory pipeline** + **two ALU (+branch) pipeline**
  - **(copy takes one cycle & others take two cycles)**
- **Unrolling enables a new scheduling opportunities**

```
1: add r0 r0 1
2: load r1 0(r0)
3: add r1 r1 1
4: store r1 0(r0)
5: bne r1 0 Loop
```

MEM0	MEM1	ALU0	ALU1
		1	
2			
		3	
4		5	

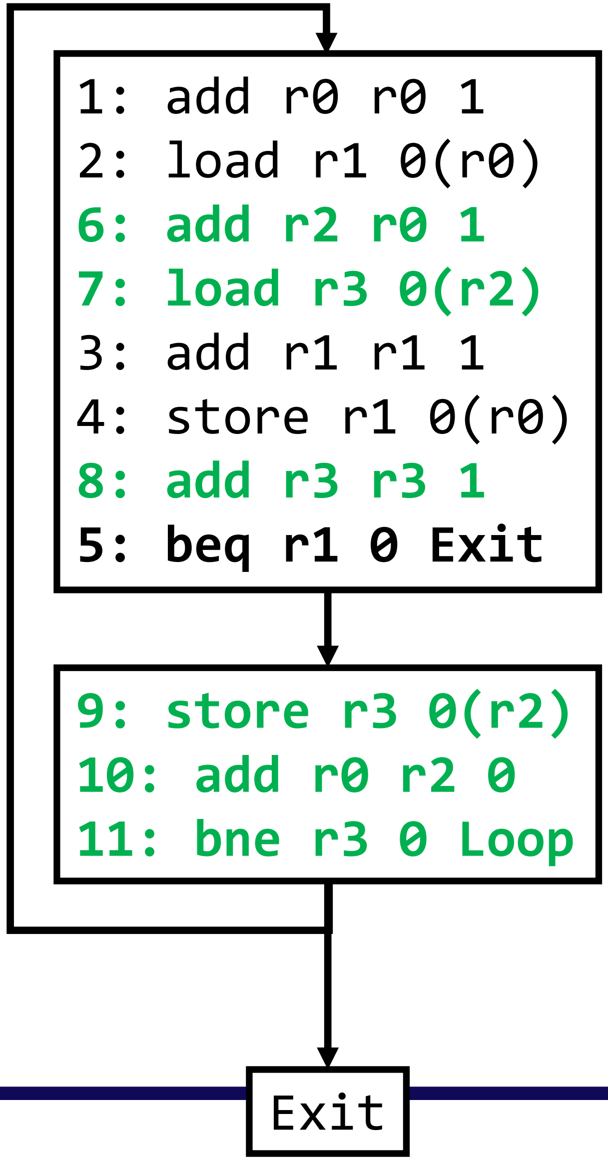
Unrolling and Instruction Scheduling - 2





MEM0	MEM1	ALU0	ALU1
		1	
2		6	
7		3	
4		5	8
9		10	11

Unrolling and Instruction Scheduling - 2



Assumption!: registers are dead after the loop

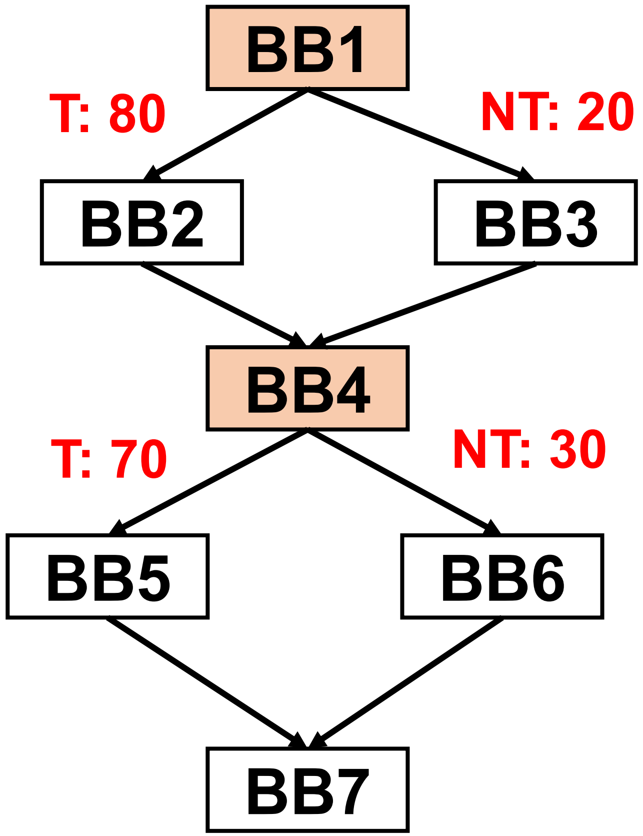
MEM0	MEM1	ALU0	ALU1
		1	
2		6	
7		3	
4		5	8
9		10	11

Loop Unroll Summary

- **Type 1 is the most effective**
  - All intermediate branches removed, least code expansion
  - Only applicable to a small fraction of loops
- **Type 2 is almost as effective**
  - Remainder loop is required since trip count not known at compile time
  - Need to make sure don’t spend much time in rem loop
- **Type 3 can be effective**
  - No branches eliminated
  - But iteration overlap still possible
  - Always applicable (most loops fall into this category!)
  - Use average trip count to guide unroll amount (Maybe profiling?)

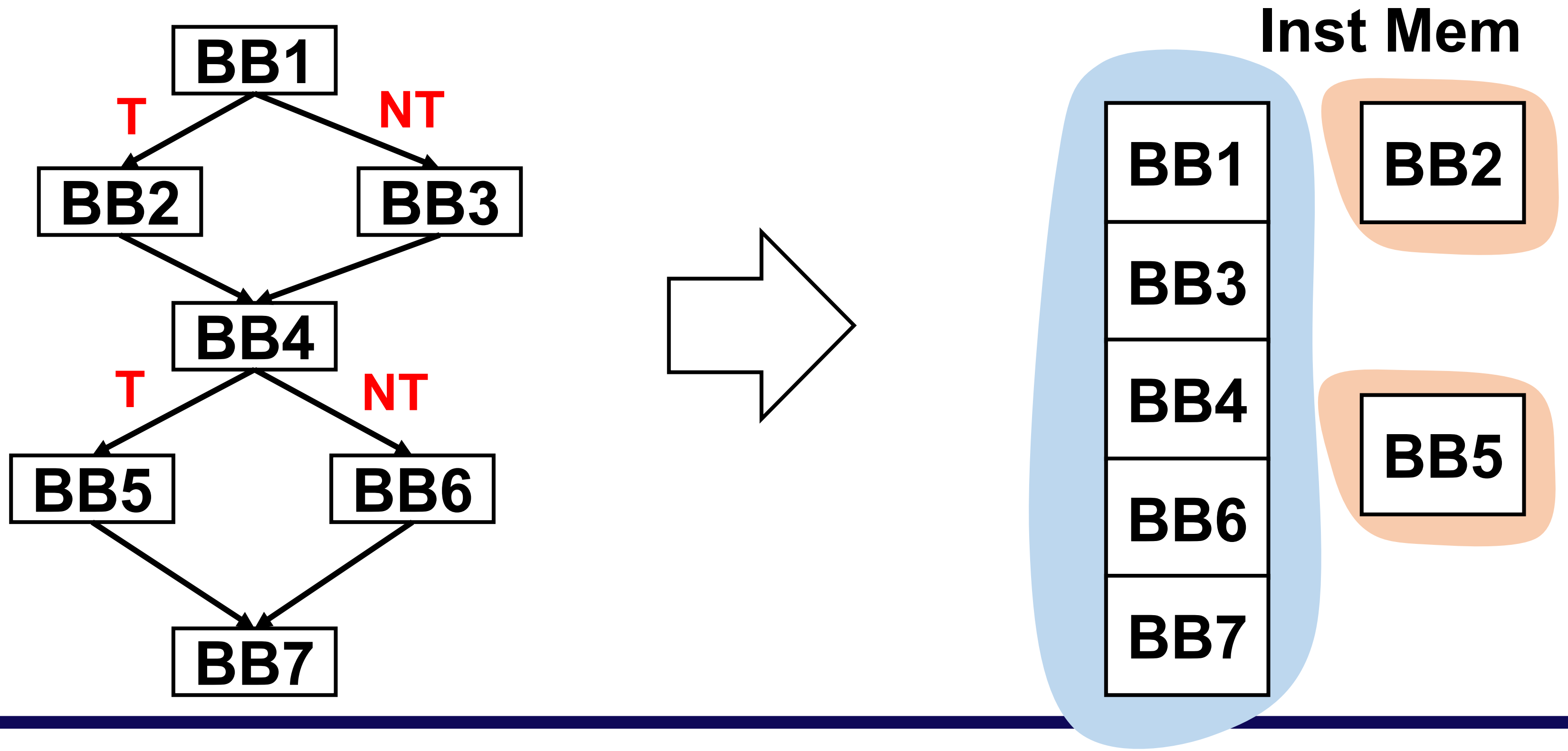
Profile-Based Optimization - 1

- We can profile the number of taken and non-taken branches at compile time (assume that it will be similar at runtime)



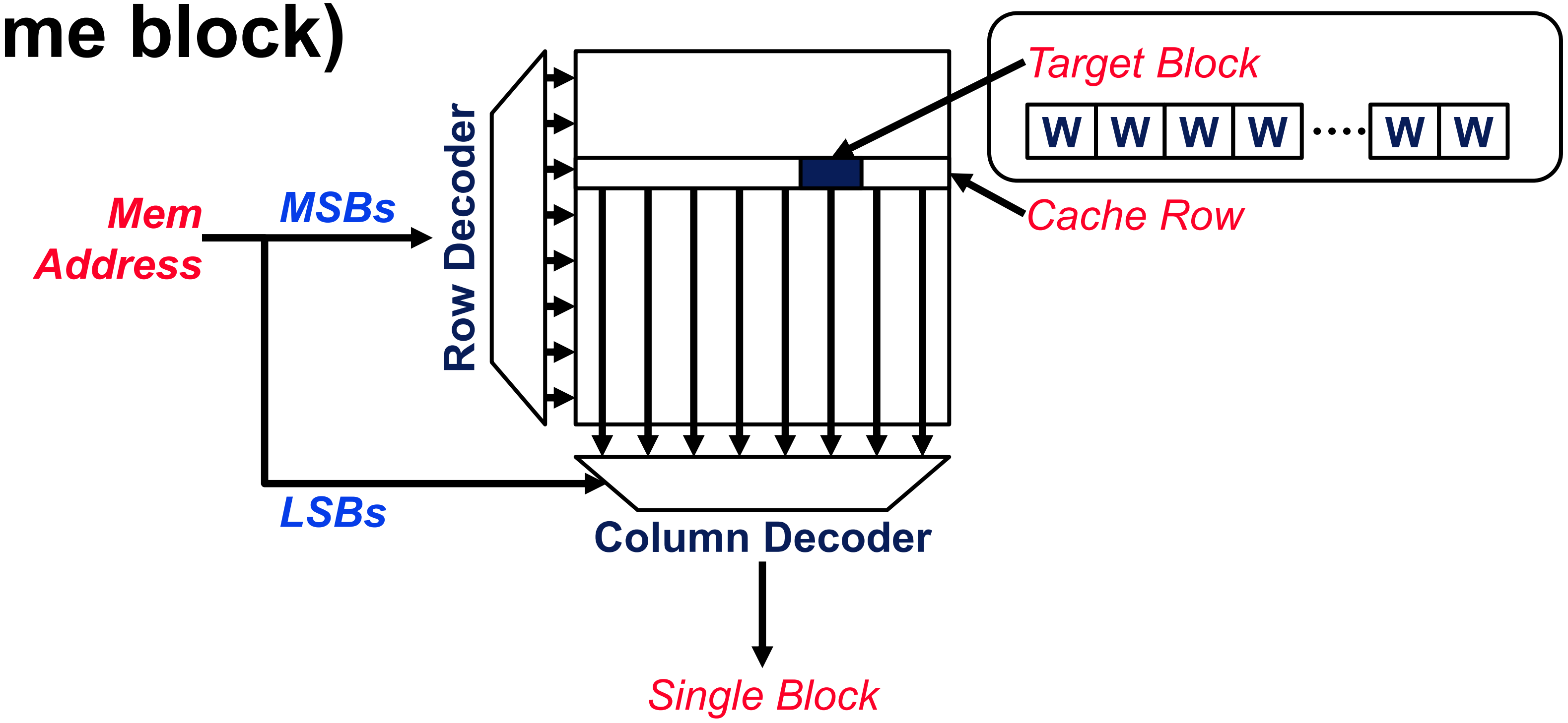
Profile-Based Optimization - 2

- **Branch taken:**
  - **Taken:** placed at non-contiguous memory region
  - **Not-Taken:** placed at contiguous memory region



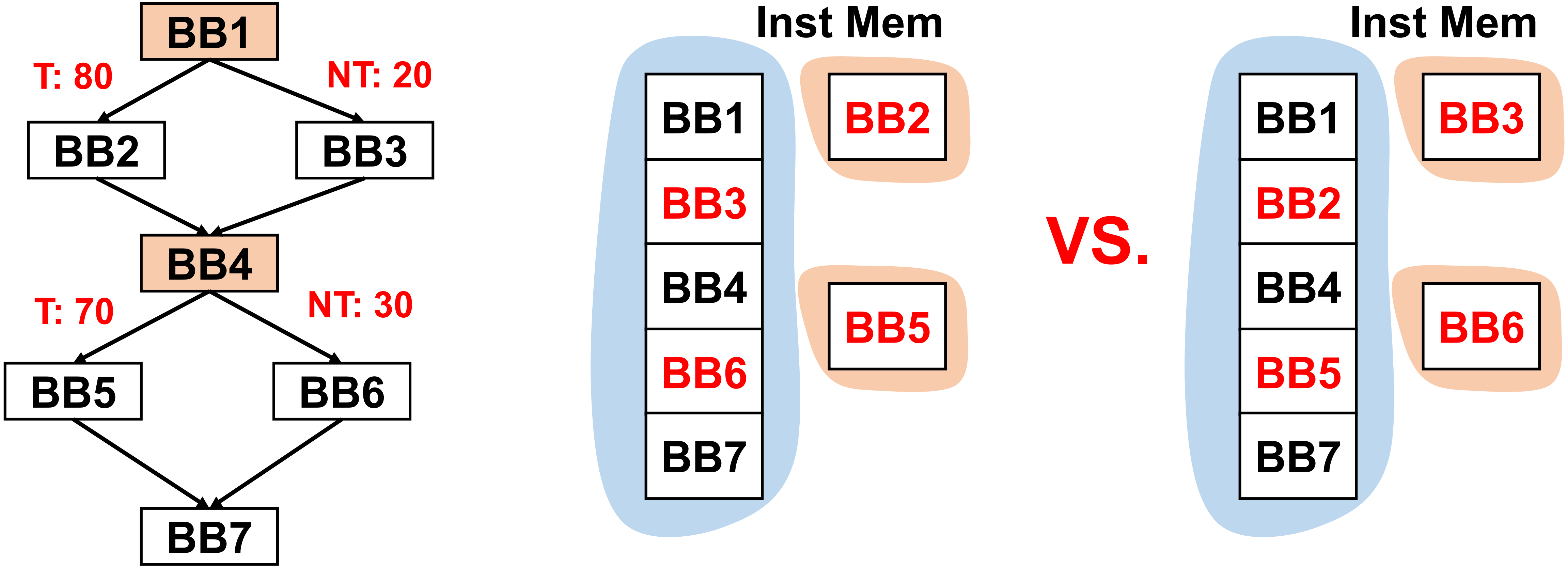
Review on the Cache

- Cache is designed to read multiple words within a single cache block (typically 4~8 words)
- The CPU can access 4 ~ 8 instructions in parallel (only if they are within the same block)



Profile-Based Optimization - 3

- It is better to frequently access the contiguous data in the memory



Linearizing a Trace

- We can modify the program block to enable spatially local trace