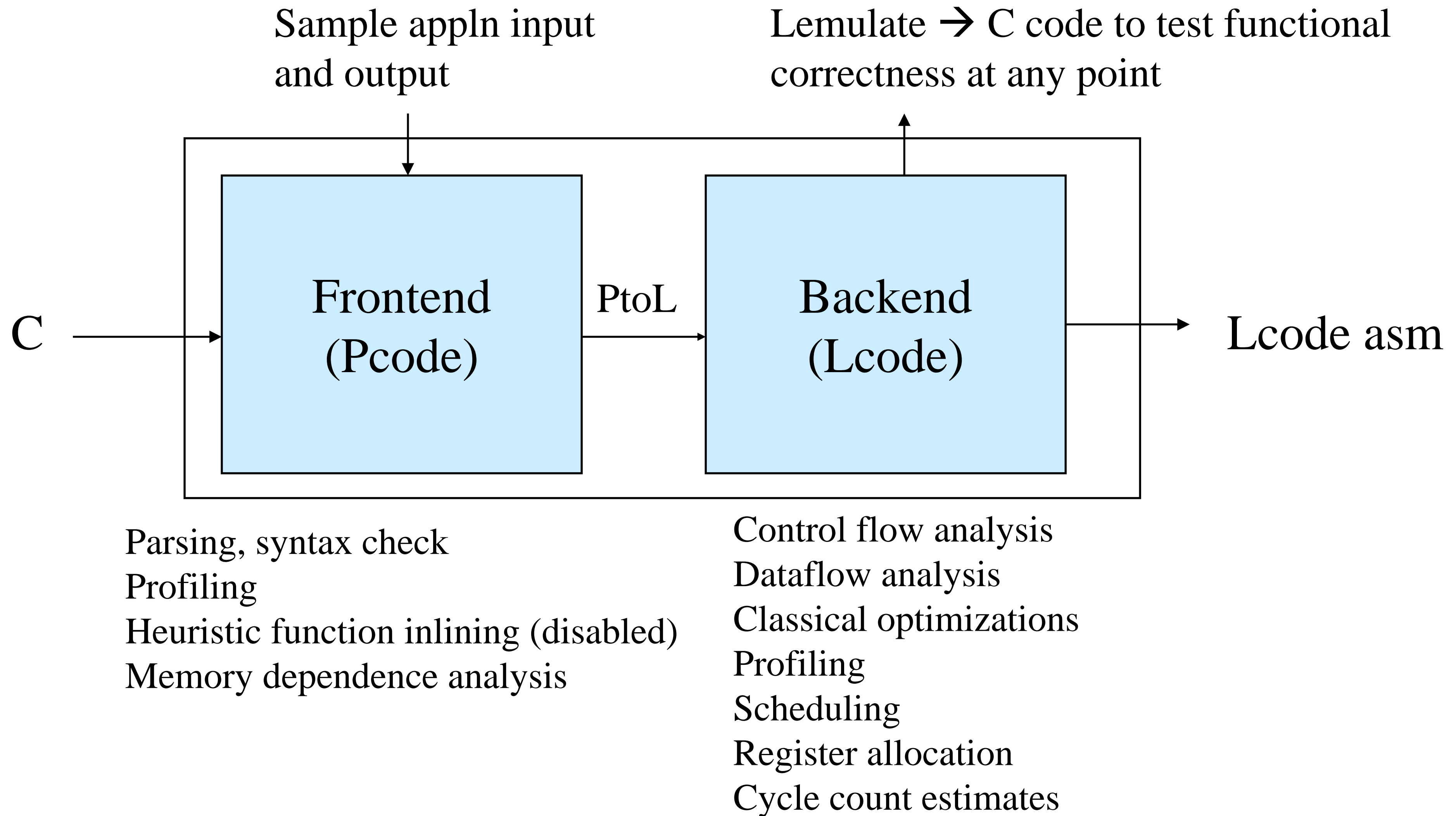
# Control Flow III: Control Flow Optimizations

EECS 483 – Lecture 21
University of Michigan
Monday, November 20, 2006

# Announcements and Reading

- Project 3 – Assigned today
  - Due Dec 13 - midnight
  - Grading will occur Dec 14/15 (thurs/fri)
  - Openimpact source code will be available tonight
- We will have class Wednes (Nov 22)
- Exam 2 – Mon Dec 11 in class
- Reading
  - Todays material not in book

# Project 3 Compiler System - Openimpact



# Compilation Steps

- 1. EDG frontend –
  - Lexical/syntax/semantic analysis
  - Pcode is created
- 2. Pcode flattening
  - remove complex control flow (ie &&,|| operators)
- 3. Pcode profiling
  - Note – compilation not halted if profile fails
- 4. Pcode inlining (disabled by default)
  - Ask me if you want to enable it
- 5. Interprocedural pointer analysis
- 6. Lcode generation (PtoL)
  - Generates *.lc files, textual assembly, this is the input to your optimizations

# Compilation Steps - Continued

- 7. Lcode optimization, lc to O (Lopti)
  - This is what you will be changing for P3
- 8. Lcode profiling, O to O_p
- 9a. Scheduling/register allocation, O_p to O_im_p
  - This essentially generates code for a "fake" machine, it's a 1-issue Lcode machine with 64 int/64 fp registers
  - Generates cycle stats – estimate of number of cycles to execute benchmark ignoring cache misses (see *.sumview)
- 9b. Lemulate – take any .lc, .O, .O_p and generate C code. From this compile with gcc to test functional correctioness

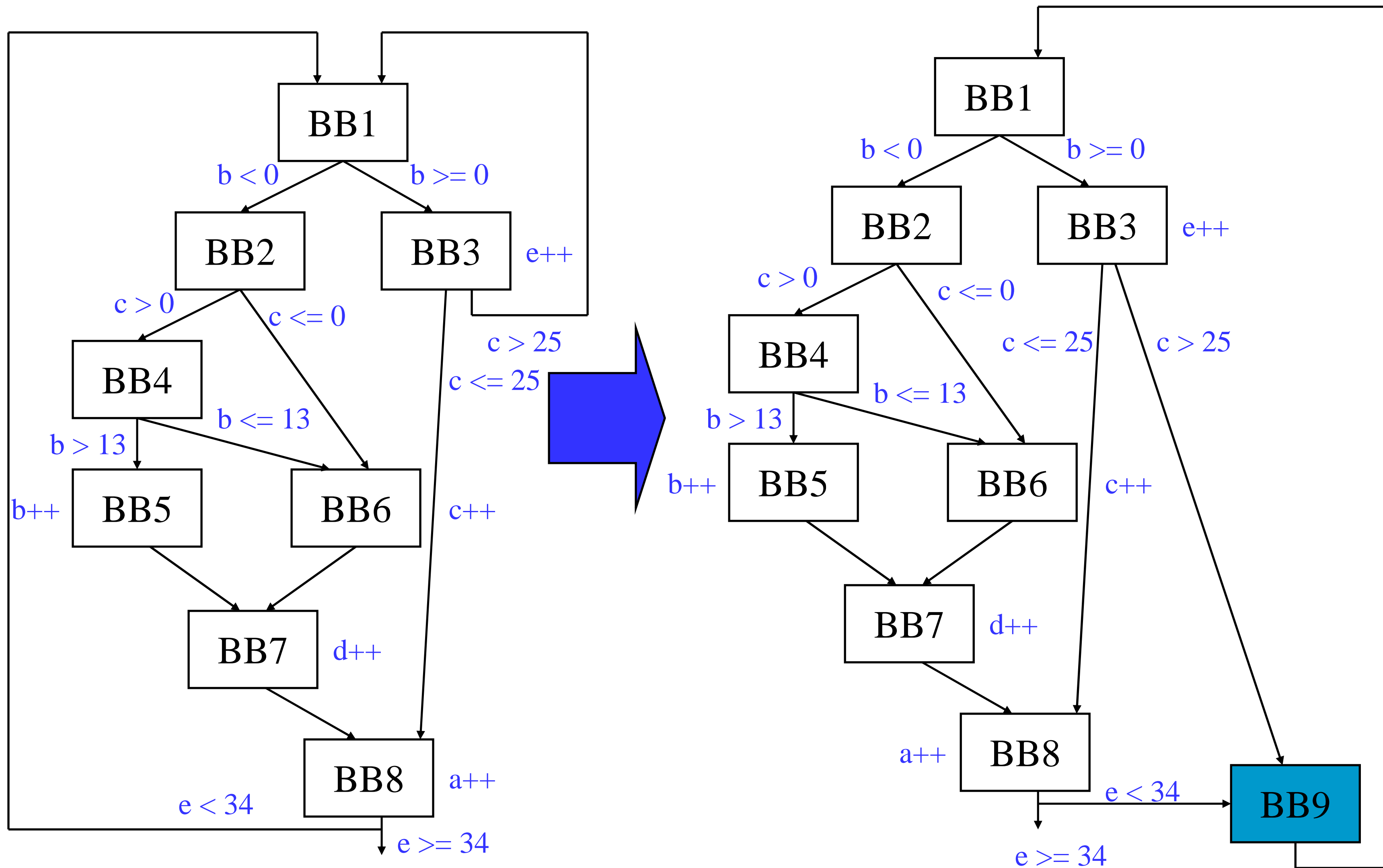
# Openimpact Notes

- Compiler composed of a series of passes
  - Each pass saves its results as a set of text files
  - Get used to the file extensions so you can look at correct input/output files
  - You care about .lc (your input), .O (your output)
- Profiling steps in compile_bench
  - Look for RESULT_CHECK_BEGIN/END, diff of result check printed here
  - Error in profile check does not stop compile_bench
- All your changes should be to the Lopti module – please do not change the Lcode dir
  - Type make inside the openimpact directory to rebuild when you make changes

# Project 3 – 3 Parts

- 0. Install and run openimpact
- 1. Implement backedge coalescing on all loops
  - Loop detection provided
  - Many other utilities available to create blocks,
    - See l_code.h, l_lcode.h, l_loop.h
- 2. Implement 2 optimizations of your choice – try to pick ones that you think will yield the largest performance gains
  - You are given a baseline compiler that performs dead code elimination, constant propagation, copy propagation, and constant folding

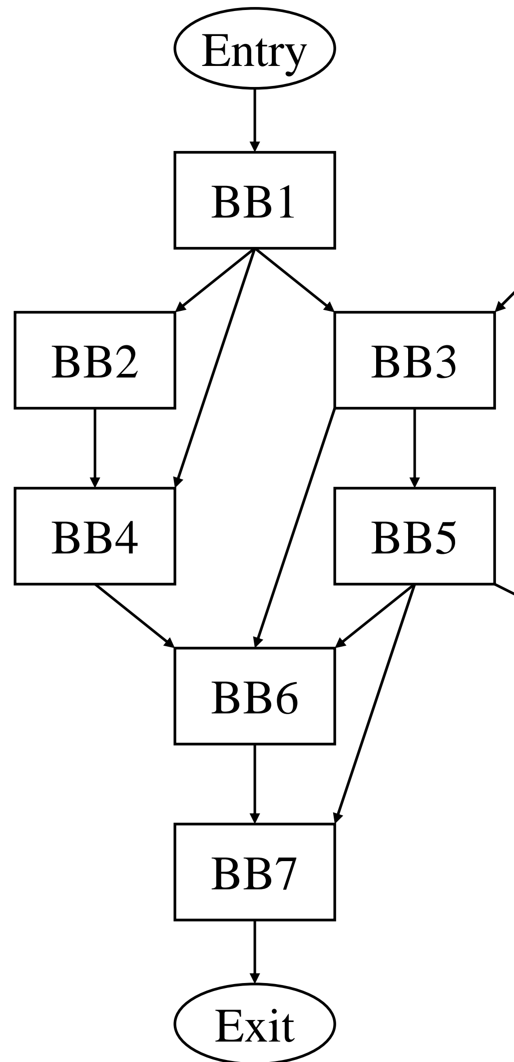
# Backedge Coalescing Example



# From Last Time: Class Problem Answer

Calculate the PDOM set for each BB

| BB | pdom |
|---|---|
| 1 | 1,7,X |
| 2 | 2,4,6,7,X |
| 3 | 3,7,X |
| 4 | 4,6,7,X |
| 5 | 5,7,X |
| 6 | 6,7,X |
| 7 | 7,X |



# Loop Induction Variables

- Induction variables are variables such that every time they changes value, they are incremented/decremented by some constant
- Basic induction variable – induction variable whose only assignments within a loop are of the form j = j +/- C, where C is a constant
- Primary induction variable – basic induction variable that controls the loop execution (for i=0; i<100; i++), i (virtual register holding i) is the primary induction variable
- Derived induction variable – variable that is a linear function of a basic induction variable

# Class Problem

Identify the basic, primary, and derived inductions variables in this loop.

```
       r1 = 0
       r7 = &A
Loop:  r2 = r1 * 4
       r4 = r7 + 3
       r7 = r7 + 1
       r1 = load(r2)
       r3 = load(r4)
       r9 = r1 * r3
       r10 = r9 >> 4
       store (r10, r2)
       r1 = r1 + 4
       blt r1 100 Loop
```

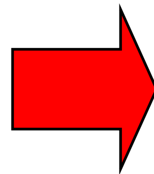
# Reducible Flow Graphs

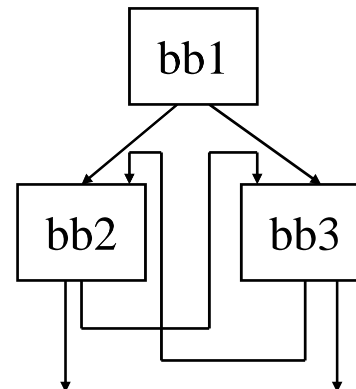
- A flow graph is <u>reducible</u> if and only if we can partition the edges into 2 disjoint groups often called forward and back edges with the following properties
  - » The forward edges form an acyclic graph in which every node can be reached from the Entry
  - » The back edges consist only of edges whose destinations dominate their sources
- More simply – Take a CFG, remove all the backedges (x→ y where y dominates x), you should have a <u>connected, acyclic</u> graph

# Irreducible Flow Graph Example

* In C/C++, its not possible to create an irreducible flow graph without using goto's
* Cyclic graphs that are NOT natural loops cannot be optimized by the compiler

```
L1: x = x + 1
if (x) {
   L2: y = y + 1
   if (y > 10) goto L3
} else {
   L3: z = z + 1
   if (z > 0) goto L2
}
```

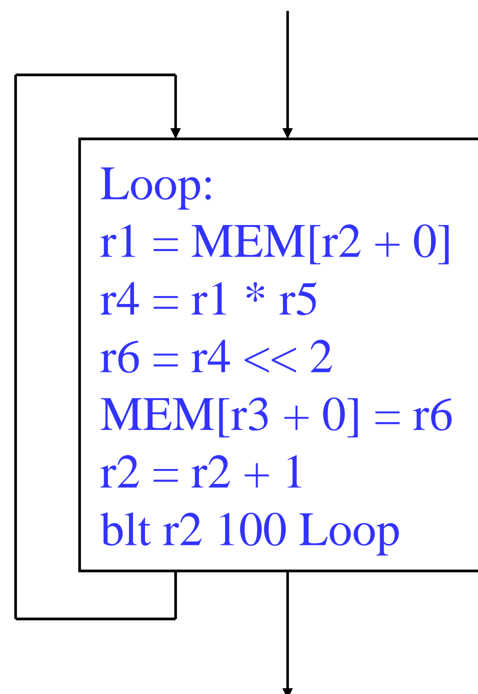





Non-reducible!

# Control Flow Optis: Loop Unrolling

- Most renowned control flow opti
- Replicate the body of a loop N-1 times (giving N total copies)
  - Loop unrolled N times or Nx unrolled
  - Enable overlap of operations from different iterations
  - Increase potential for ILP (instruction level parallelism)
- 3 variants
  - Unroll multiple of known trip count
  - Unroll with remainder loop
  - While loop unroll

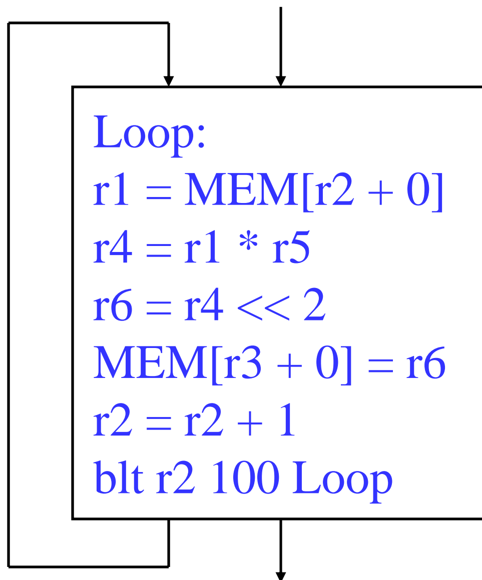


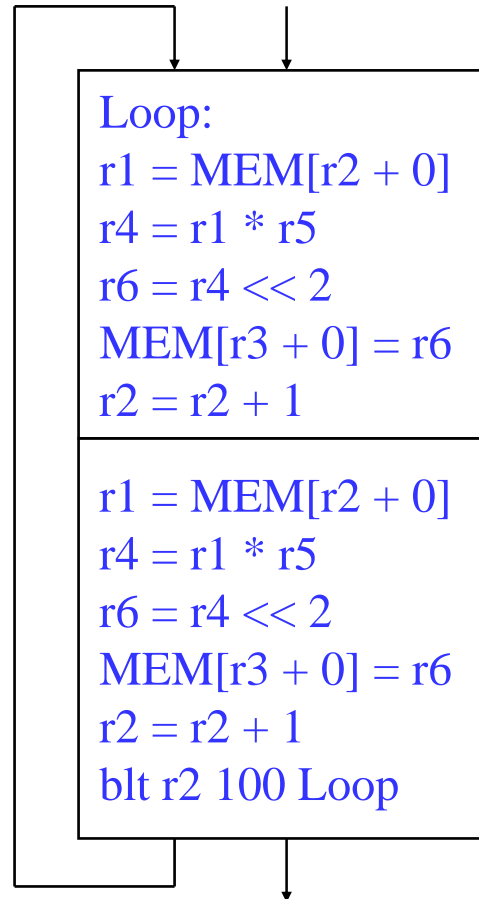
# Loop Unroll – Type 1

Counted loop

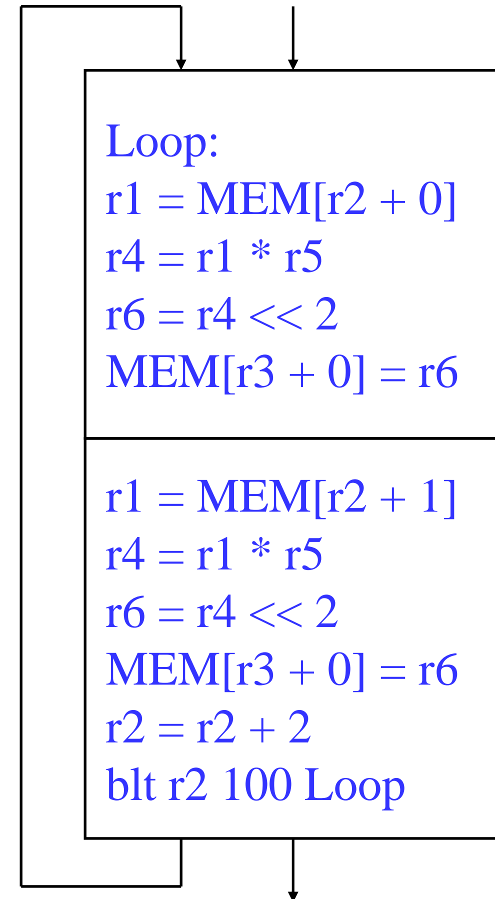
All parms known

r2 is the loop variable,
Increment is 1
Initial value is 0
Final value is 100
Trip count is 100




Remove branch from first N-1 iterations

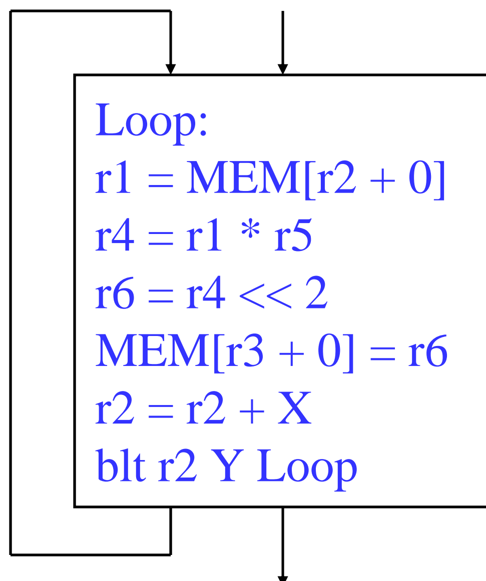


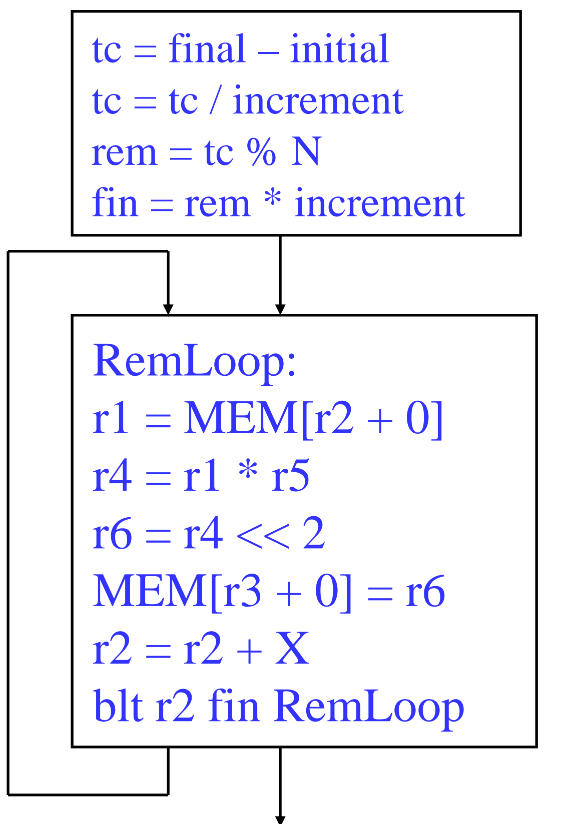

Remove r2 increments from first N-1 iterations and update last increment

# Loop Unroll – Type 2

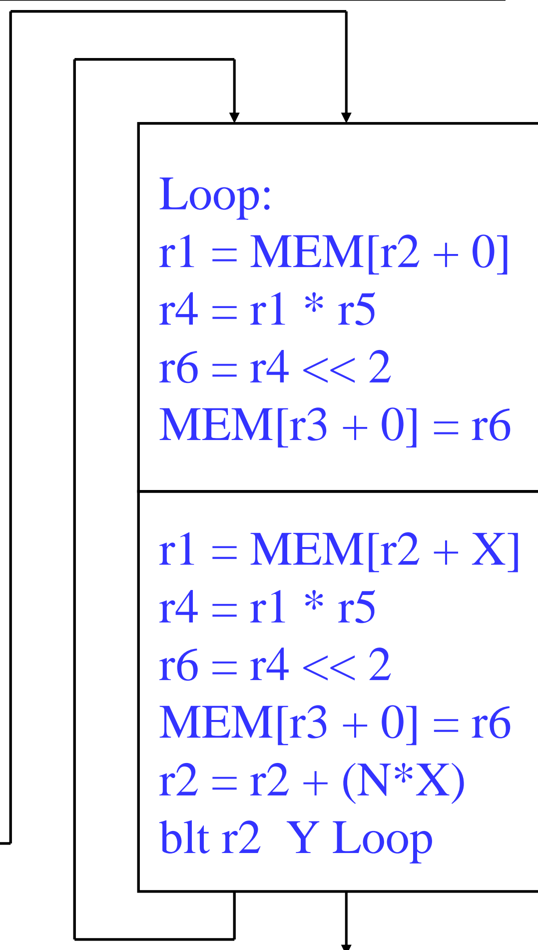
Counted loop
Some parms unknown

r2 is the loop variable,
Increment is ?
Initial value is ?
Final value is ?
Trip count is ?




Remainder loop executes the “leftover” iterations

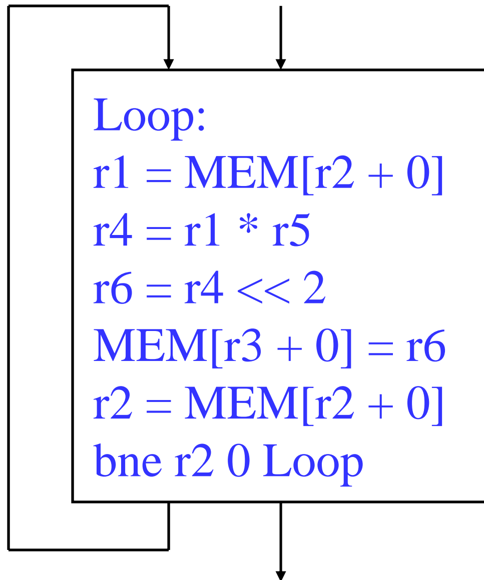


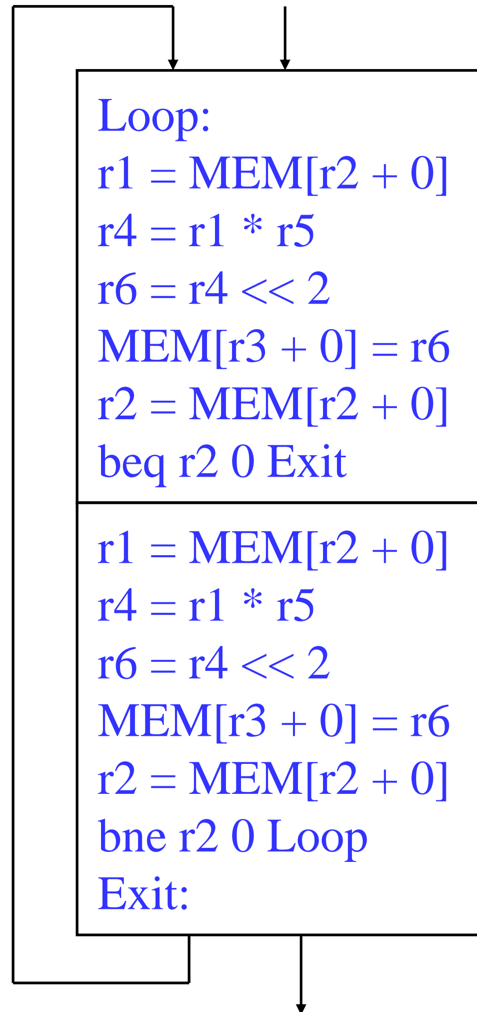

Unrolled loop same as Type 1, and is guaranteed to execute a multiple of N times

# Loop Unroll – Type 3

Non-counted loop
Some parms unknown

pointer chasing, loop var modified in a strange way, etc.




Just duplicate the body, none of the loop branches can be removed. Instead they are converted into conditional breaks

Can apply this to any loop!

# Loop Unroll Summary

- Goals
  - » Reduce number of executed branches inside loop
    - Note: Type1/Type2 only
  - » Enable the overlapped execution of multiple iterations
    - Reorder instructions between iterations
  - » Enable dataflow optimization across iterations
- Type 1 is the most effective
  - » All intermediate branches removed, least code expansion
  - » Only applicable to a small fraction of loops

# Loop Unroll Summary (2)

- Type 2 is almost as effective
  - All intermediate branches removed
  - 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

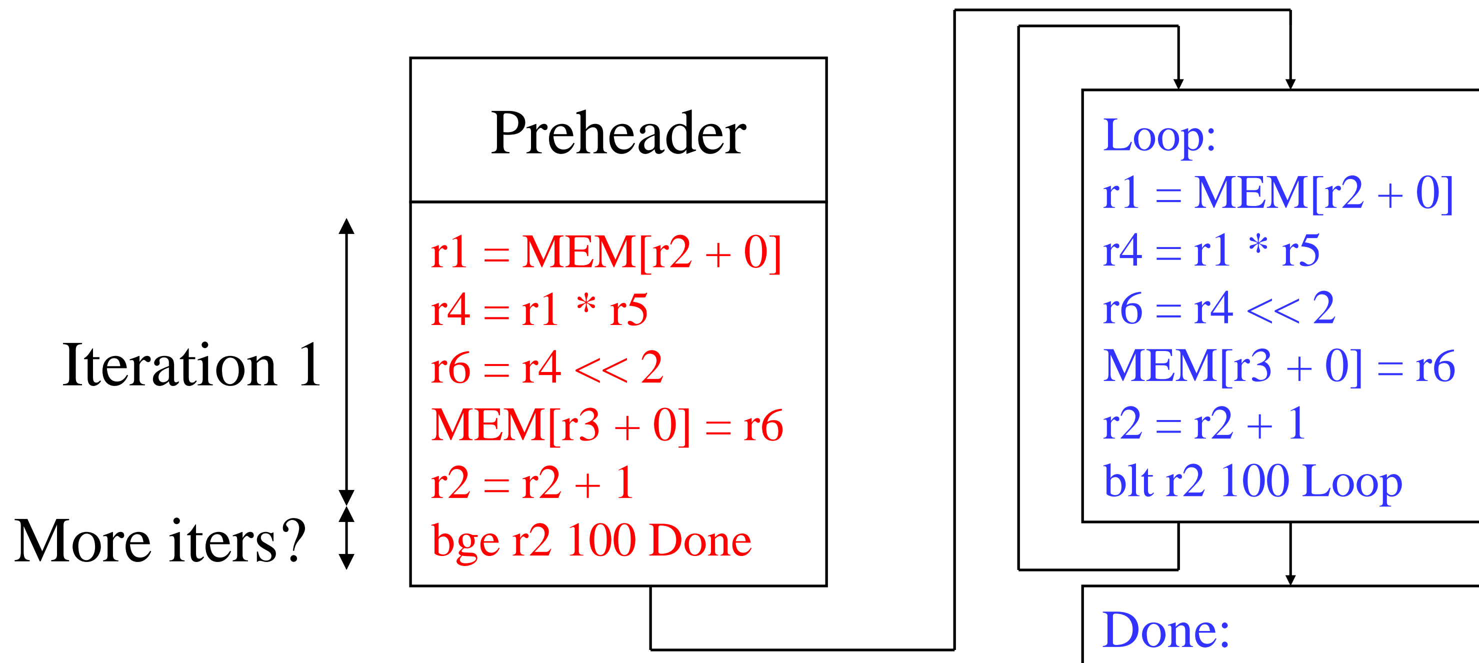
# Class Problem

Unroll both the outer loop and inner loop 2x. Apply the most aggressive style unrolling that you can, e.g., Type 1 if possible, else Type 2, else Type 3

```
for (i=0; i<100; i++) {
   j = i;
   while (j < 100) {
         A[j]--;
   }
   B[i] = 0;
}
```

# Loop Peeling

- Unravel first P iterations of a loop
  - Enable overlap of instructions from the peeled iterations with preheader instructions
  - Increase potential for ILP
  - Enables further optimization of main body



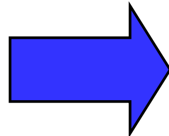
# Control Flow Opti for Acyclic Code

- Rather simple transformations with these goals:
  - Reduce the number of dynamic branches
  - Make larger basic blocks
  - Reduce code size
- Classic control flow optimizations
  - Branch to unconditional branch
  - Unconditional branch to branch
  - Branch to next basic block
  - Basic block merging
  - Branch to same target
  - Branch target expansion
  - Unreachable code elimination

# Acyclic Control Flow Optimizations (1)

1. Branch to unconditional branch

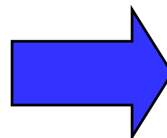
```
L1: if (a < b) goto L2
. . .
L2: goto L3
```



```
L1: if (a < b) goto L3
. . .
L2: goto L3
```
→ may be deleted

2. Unconditional branch to branch

```
L1: goto L2
. . .
L2: if (a < b) goto L3
L4:
```

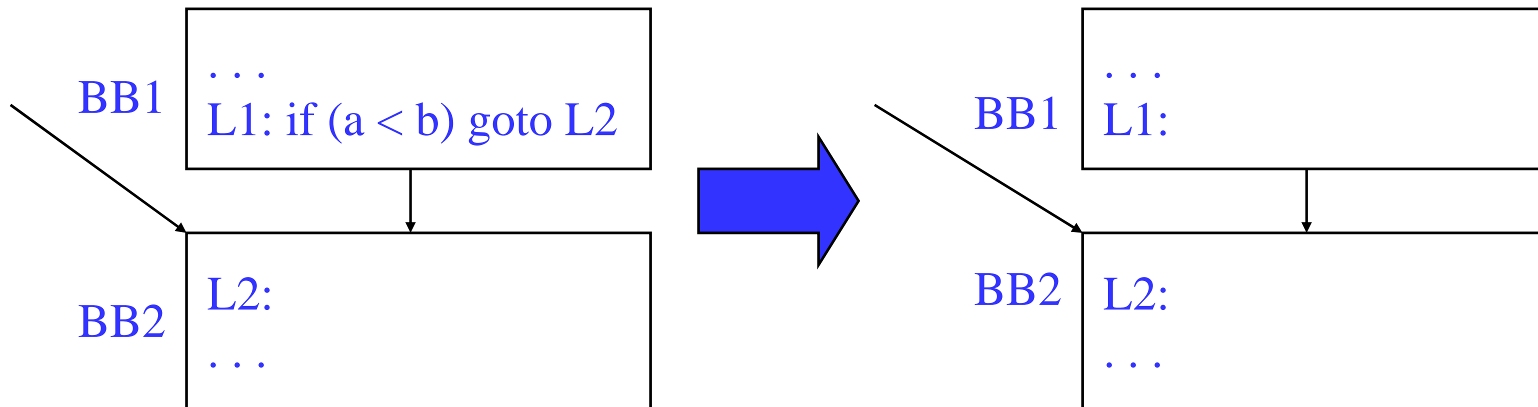


```
L1: if (a < b) goto L3
goto L4:
. . .
L2: if (a < b) goto L3
L4:
```
→ may be deleted (L2)

# Acyclic Control Flow Optimizations (2)

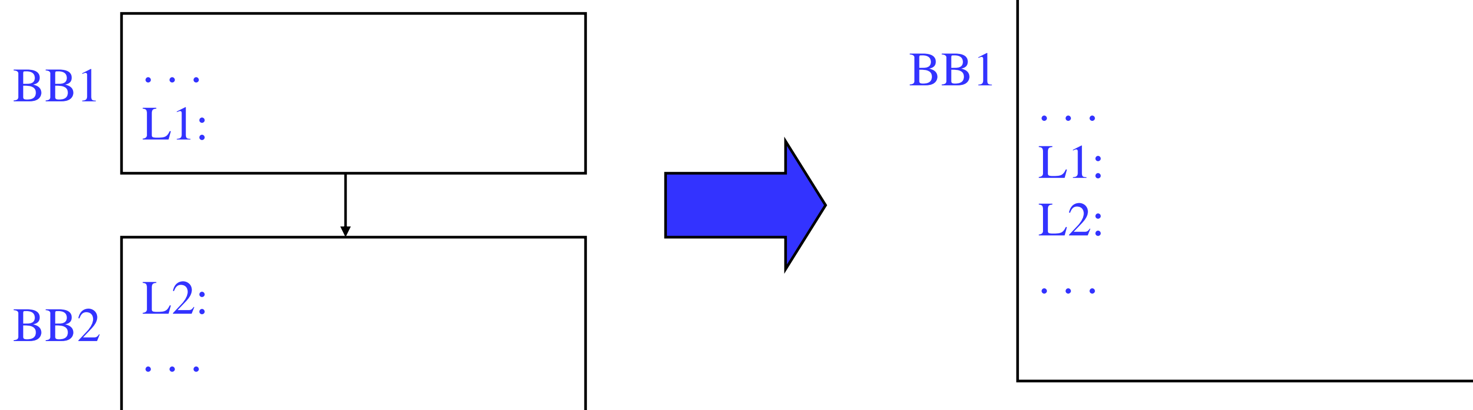
3. Branch to next basic block

Branch is unnecessary



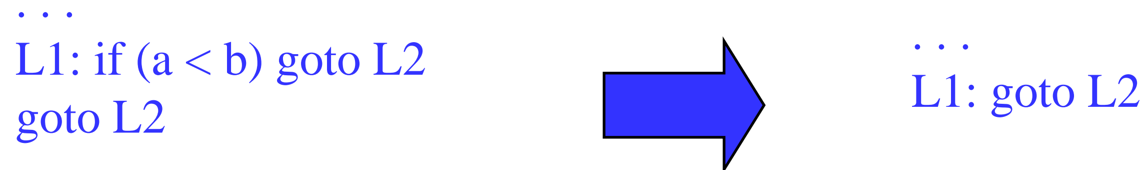

4. Basic block merging

Merge BBs when single edge between

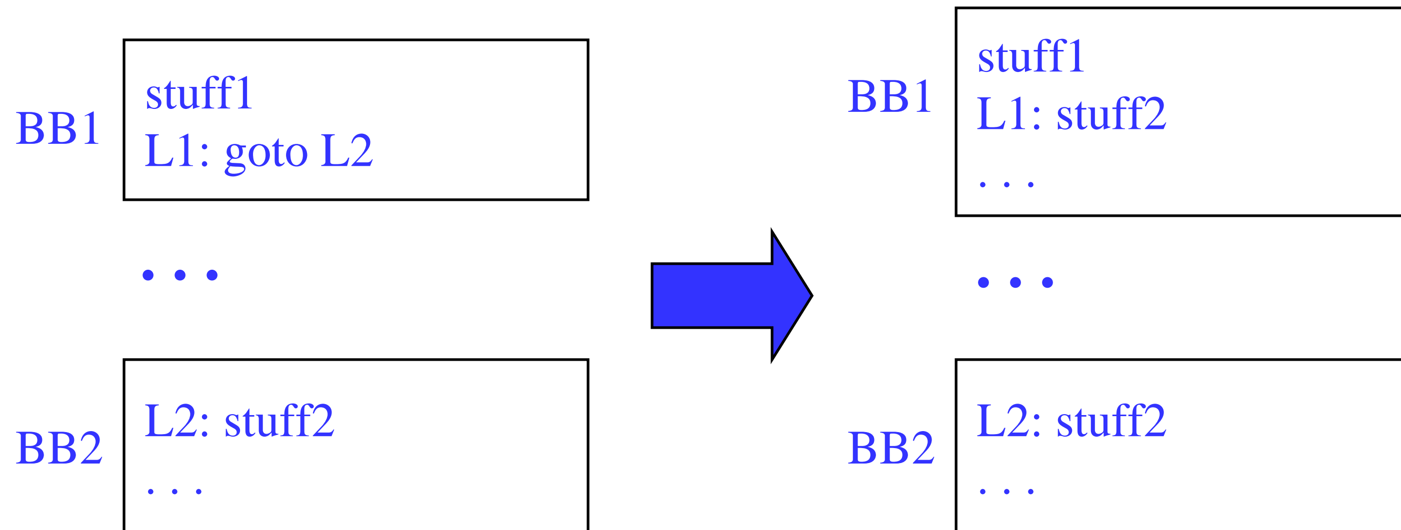


# Acyclic Control Flow Optimizations (3)

5. Branch to same target




6. Branch target expansion

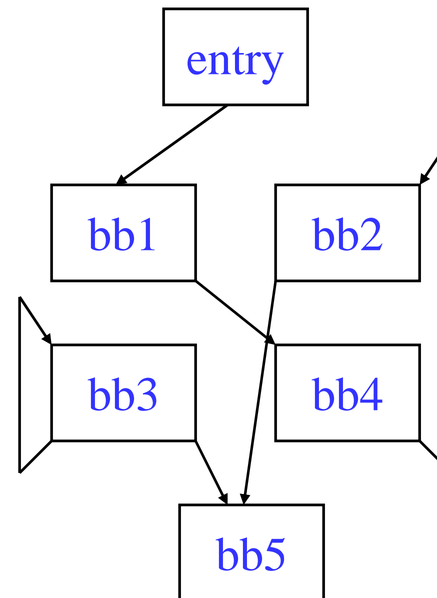



What about expanding a conditional branch? -- Almost the same

# Unreachable Code Elimination

## Algorithm

```
Mark procedure entry BB visited
to_visit = procedure entry BB
while (to_visit not empty) {
    current = to_visit.pop()
    for (each successor block of current) {
            Mark successor as visited;
            to_visit += successor
    }
}
Eliminate all unvisited blocks
```




Which BB(s) can be deleted?

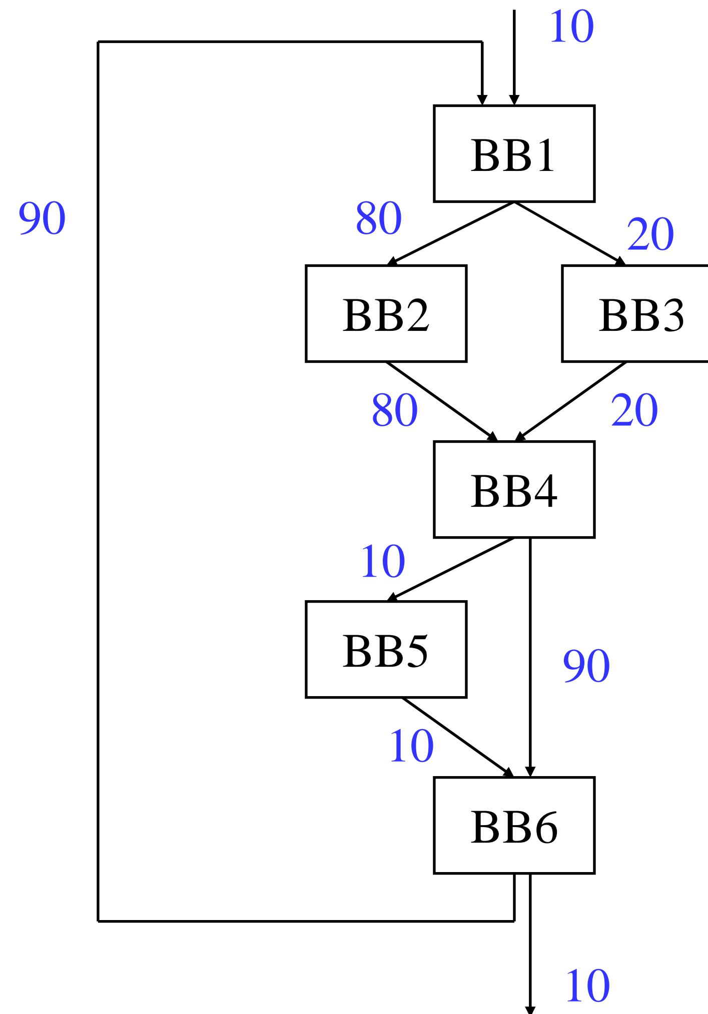
# Class Problem

Maximally optimize the control flow of this code

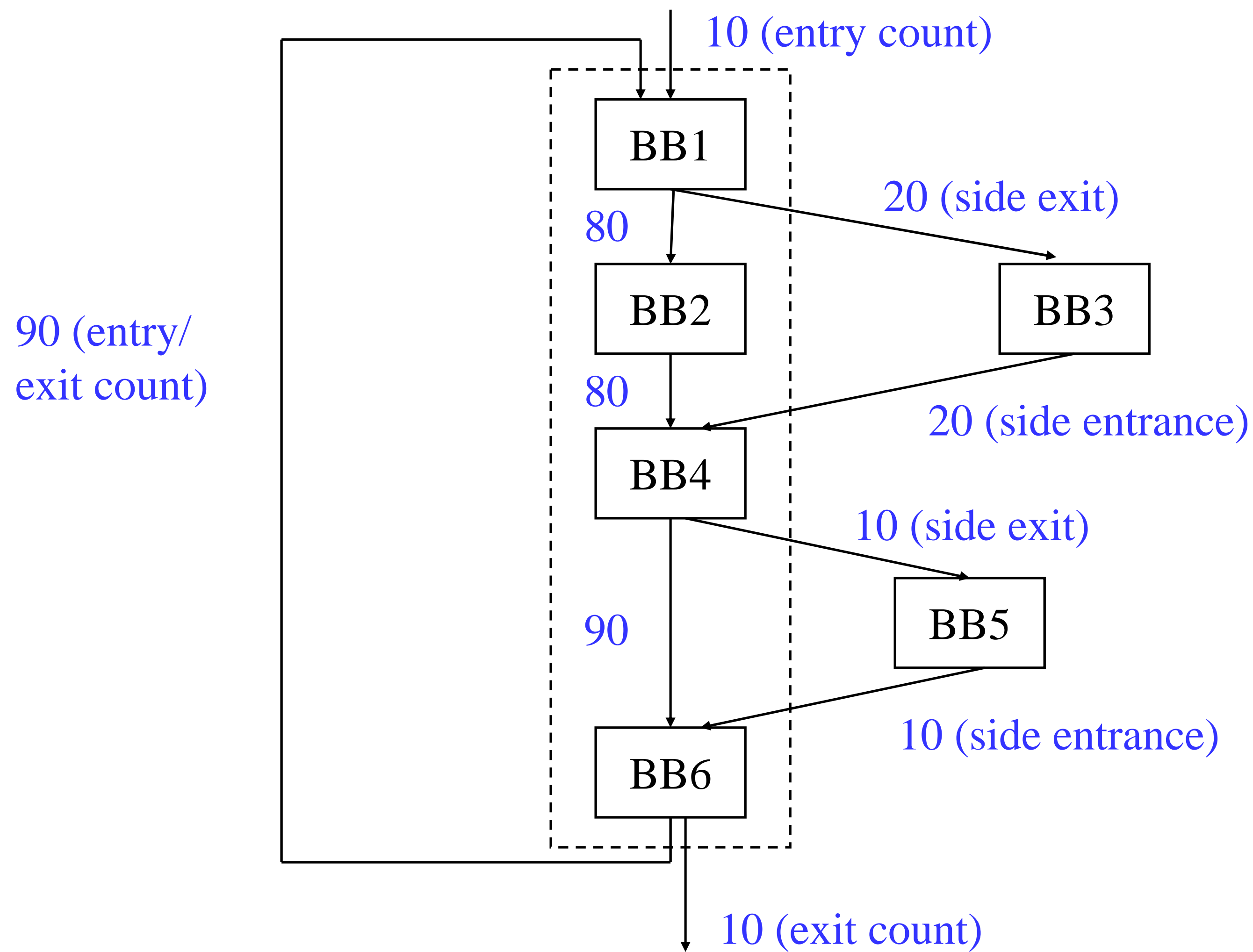
```
L1: if (a < b) goto L11
---
L2: goto L7
---
L3: goto L4
---
L4: stuff4
L5: if (c < d) goto L15
---
L6: goto L2
---
L7: if (c < d) goto L13
---
L8: goto L12
---
L9: stuff 9
L10: if (a < c) goto L3
---
L11:goto L9
---
L12: goto L2
---
L13: stuff 13
L14: if (e < f) goto L11
---
L15: stuff 15
L16: rts
```

# Profile-based Control Flow Optimization: Trace Selection

- Trace - Linear collection of basic blocks that tend to execute in sequence
  - "Likely control flow path"
  - Acyclic (outer backedge ok)
- Side entrance – branch into the middle of a trace
- Side exit – branch out of the middle of a trace



# Linearizing a Trace



# Intelligent Trace Layout for Icache Performance