

# Control Flow IV: Control Flow Optimizations Start Dataflow Analysis

---

EECS 483 – Lecture 22

University of Michigan

Wednesday, November 22, 2006

# Announcements and Reading

---

## ❖ Project 3

- » Get started building openimpact!!
- » Make sure you get the environment variables set up before trying to install

## ❖ Reading

- » 10.6

# l\_code.h: Core Lcode Data Structures

---

- ❖ L\_Operand – src/dest operands
  - » Register, Macro, Literal (Int, Flt, Dbl, Label, String, Cb)
  - » Ignore ptype
  - » ctype = data type
- ❖ L\_Oper – operations/instructions
  - » Has id, opcode, src/dest operands
  - » Ignore pred operand
  - » Connected in doubly linked list (next\_op, prev\_op)
- ❖ L\_Cb – blocks
  - » More general than a basic block – single entry, 1 or more exits
  - » First/last ops, weight, src\_flow, dest\_flow
  - » Connected in doubly linked list (next\_cb, prev\_cb)

## l\_code.h: Core Lcode Data Structures (cont)

---

- ❖ L\_Flow – control flow edges for CFG
  - » src\_cb, dst\_cb, weight
  - » Connected in doubly linked list (prev/next)
- ❖ L\_Func – function
  - » First/last cbs, weight
  - » Oper and cb hash tables (id -> L\_Oper/L\_Cb)
- ❖ L\_Attr – extra information
  - » L\_Oper, L\_Cb, and L\_Func have attributes
  - » Name, array of L\_Operands

# Example Opti: Local Constant Propagation

---

```
for (opA = cb->first_op; opA != NULL; opA = opA->next_op) {
    /* Match pattern */
    if (!L_move_opcode (opA)) continue;
    if (!L_is_constant (opA->src[0])) continue;
    for (opB = opA->next_op; opB != NULL; opB = opB->next_op) {
        if (!L_is_src_operand (opA->dest[0], opB)) continue;
        if (!L_can_change_src_operand (opB, opA->dest[0])) continue;
        if (!L_no_defs_between (opA->dest[0], opA, opB)) continue;
        macro_flag = L_is_fragile_macro (opA->dest[0]);
        load_flag = 0;
        store_flag = 0;
        if (!L_no_danger (macro_flag, load_flag, store_flag, opA, opB))
            break;
        /* Replace pattern */
        for (i = 0; i < L_max_src_operand; i++) {
            if (L_same_operand (opA->dest[0], opB->src[i])) {
                opB->src[i] = L_copy_operand (opA->src[0]);
            }
        }
    }
} In reality, more code than this, but this is the important stuff
```

# Running Things Manually, Debugging

---

- ❖ First, generate lc files using script
  - » `compile_bench wc -c2lc`
- ❖ `tar zxvf wc.lc.tgz`
  - » These are the unoptimized Lcode files
- ❖ Run `Lopti` manually
  - » `Lopti -Farch=IMPACT -Fmodel=Lcode -Fopti=1 -i wc.lc -o wc.O`
    - `opti=0`: will disable all optimizations
  - » Compare `wc.lc` and `wc.O`, should see results of your optimizations
  - » Note, if benchmark has multiple files, need to run `Lopti` on each `.lc` file

# Running Things Manually, Debugging - cont

---

## ❖ Running Lemulate manually

- » `ls *.O > list`
- » `Lemulate -i list`
- » Generates a `.c` file for each `.O` file

## ❖ Compile the `.c` files with `gcc`

- » `gcc -g *.c $IMPACT_REL_PATH/platform/x86lin_gcc/impact_lemul_lib.o -lm`
- » Generates `a.out`, so now run it.

## ❖ Debugging – `gdb`

- » Can `gdb` the benchmark `a.out` or `Lopti`
- » Can also add print statements to `Lopti`

# From Last Time: 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
- » Expose optimizations across iterations

❖ Type 1 – compile-time constant trip count

❖ Type 2 – counted, but trip count unknown at compile time

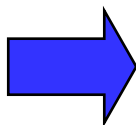
❖ Type 3 – non-counted

# From Last Time - 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;  
}
```



```
for (i=0; i<100; i+=2) {  
    j = i;  
    while (j < 100) {  
        A[j++]--;  
    }  
    B[i] = 0;  
    j = i+1;  
    while (j < 100) {  
        A[j++]--;  
    }  
    B[i+1] = 0;  
}
```

Outer → type 1 (known trip count)

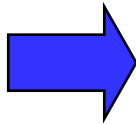
Inner → type 2 (counted loop)

# From Last Time: Class Problem (cont'd)

---

Expanding the while loop – For the problem this needs to be done for both while loops

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



```
j = i
tripcount = 100 - j
tripcount = tripcount / 1
remainder = tc % 2
final = remainder * 1
final = final + j /* I forgot this on the
unroll 2 slide */
```

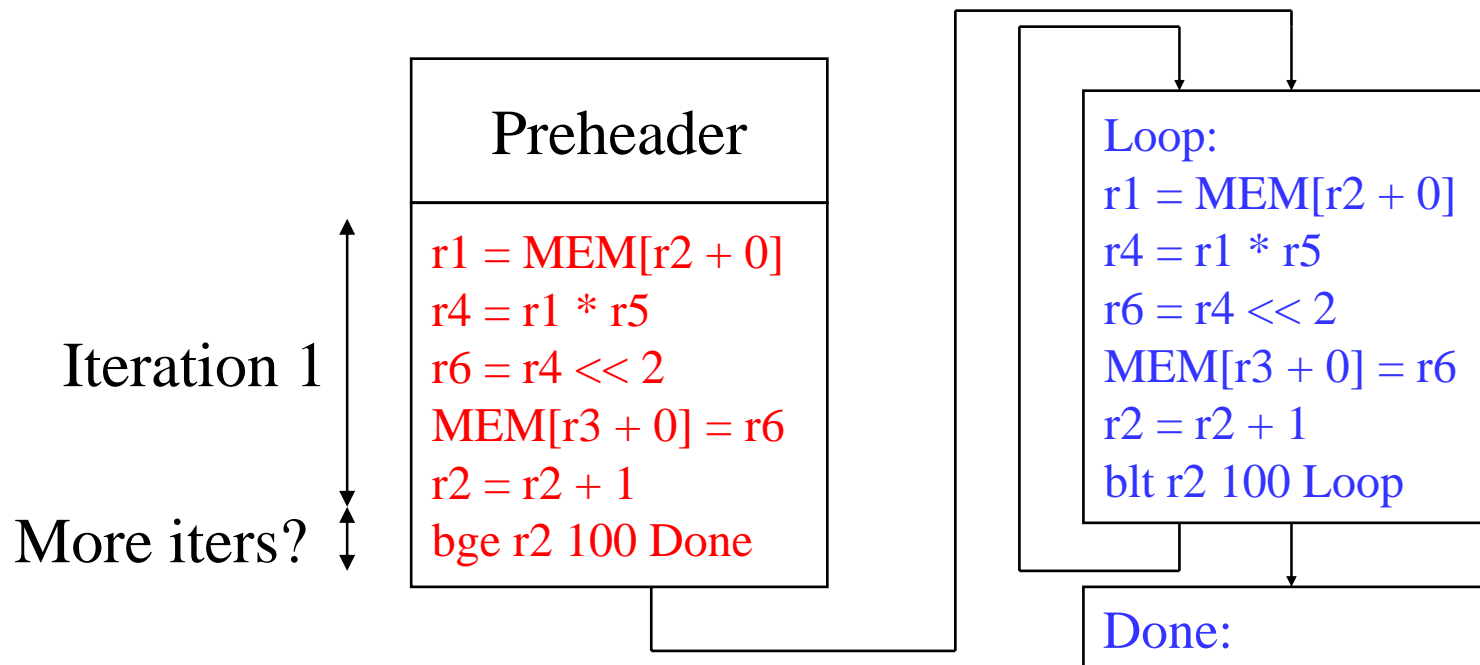
```
Remloop:
while (j < final) {
    A[j++]--;
```

```
Unrolledloop:
while (j < 100) {
    A[j]--;
    A[j+1]--;
    j+=2;
}
```

# 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)

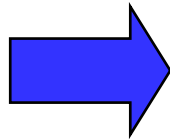
---

## 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

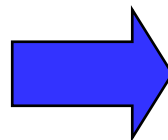
## 2. Unconditional branch to branch

L1: goto L2

...

L2: if (a < b) goto L3

L4:



L1: if (a < b) goto L3

goto L4:

...

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

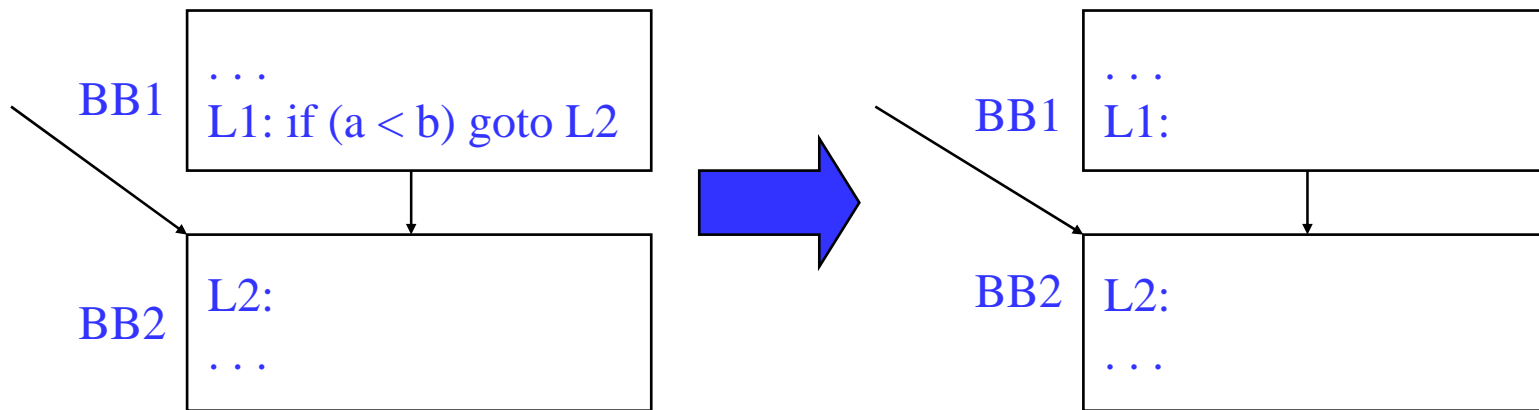
L4:

# Acyclic Control Flow Optimizations (2)

---

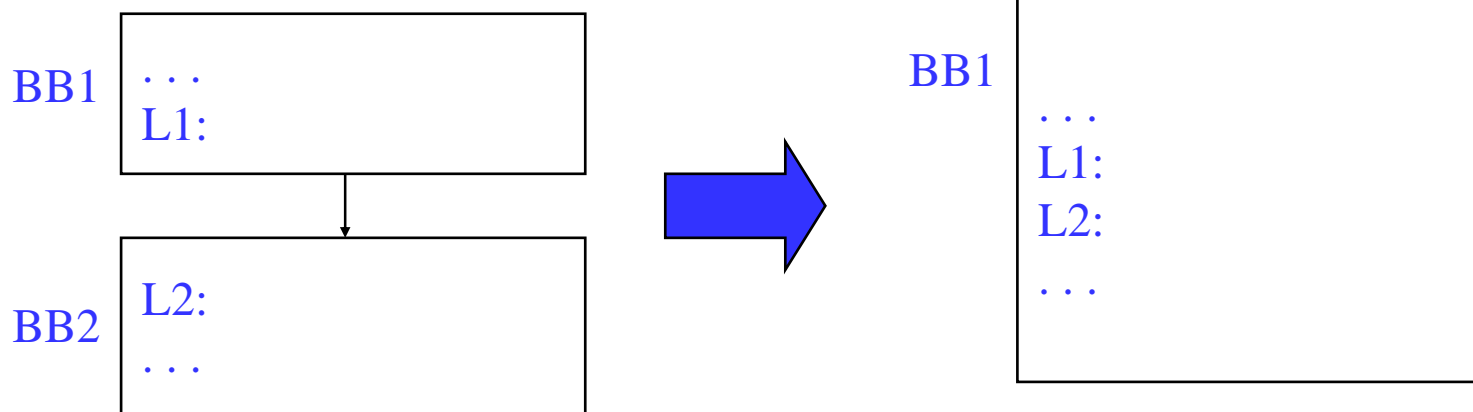
## 3. Branch to next basic block

Branch is unnecessary



## 4. Basic block merging

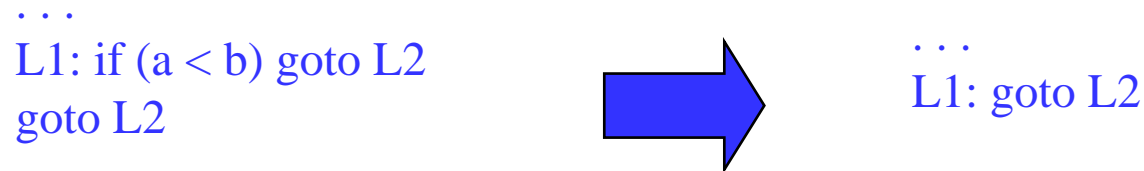
Merge BBs when single edge between



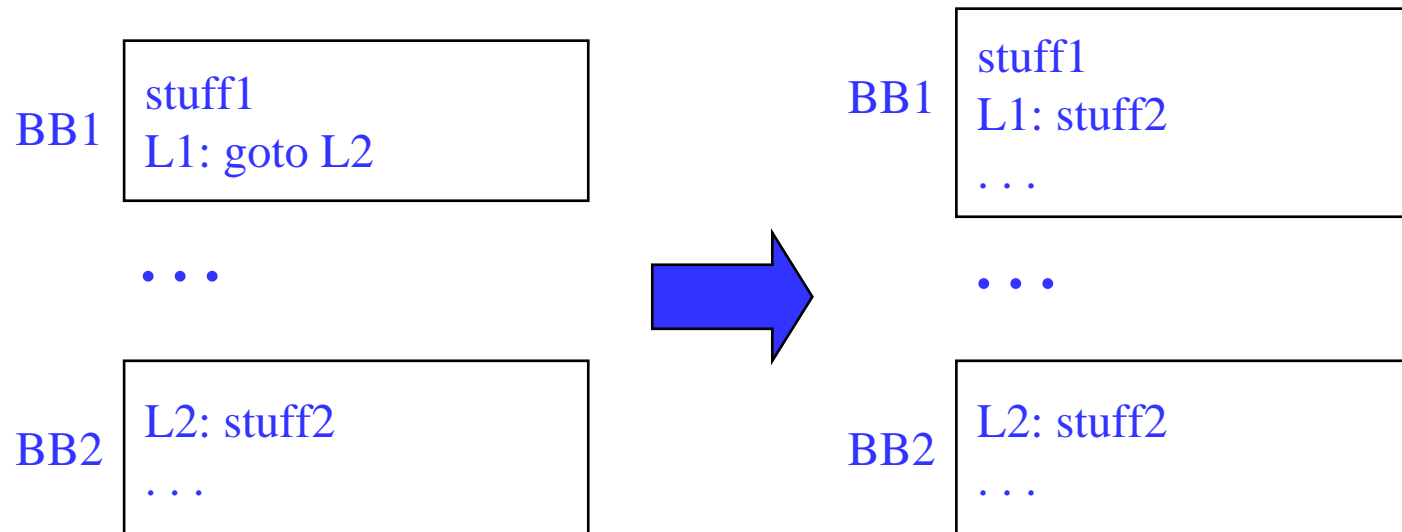
# Acyclic Control Flow Optimizations (3)

---

## 5. Branch to same target



## 6. Branch target expansion



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

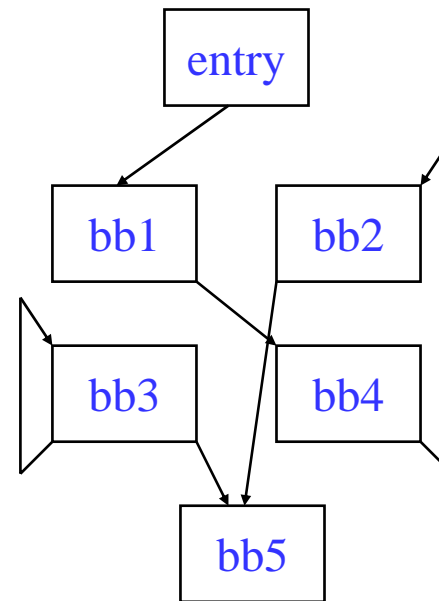
---

# 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

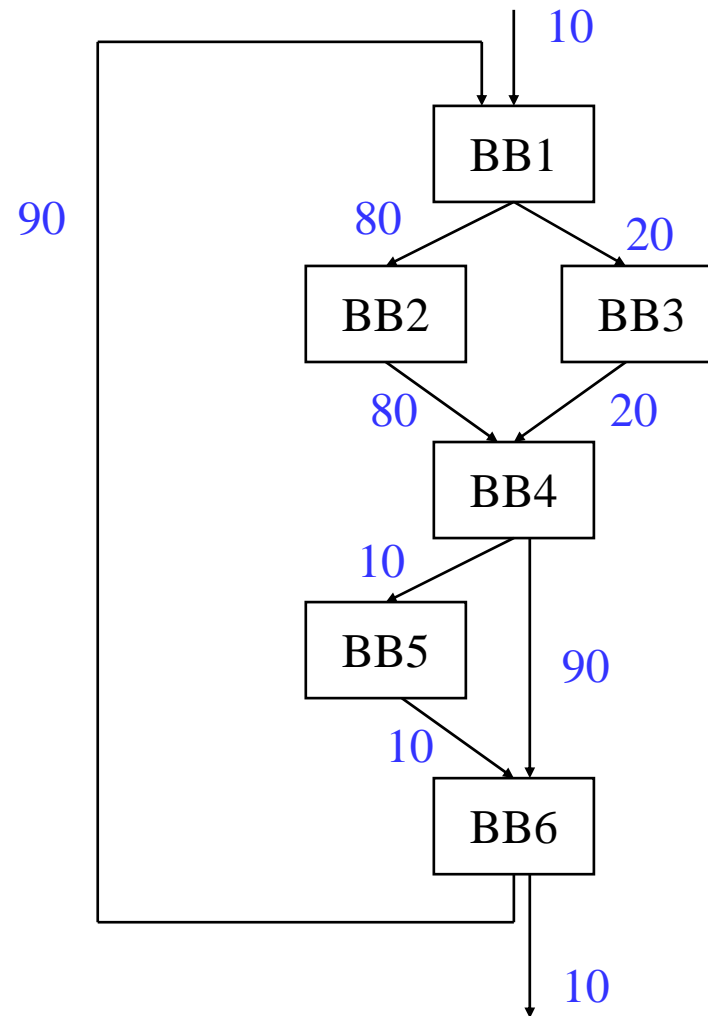
---

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

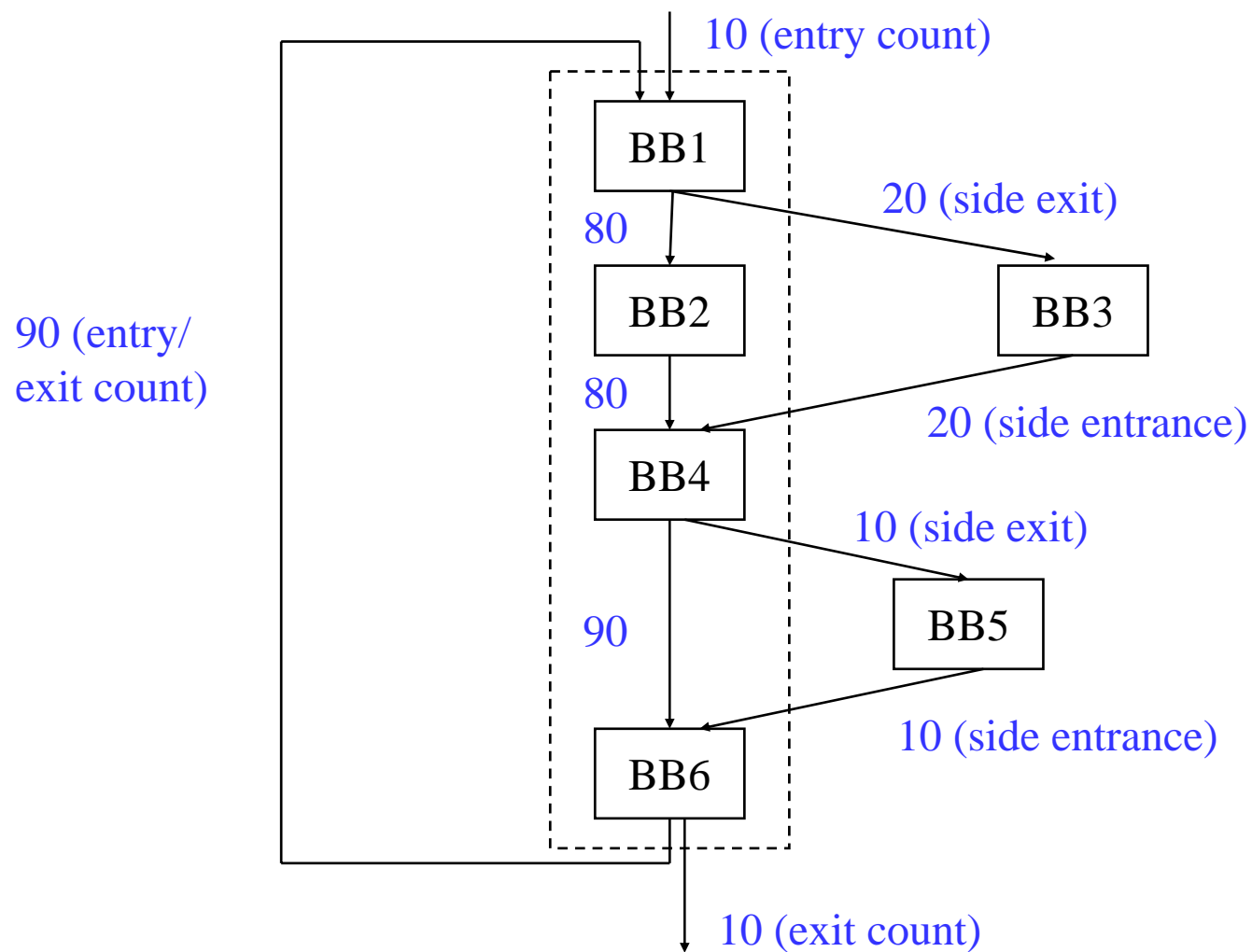
# Profile-based Control Flow Optimization: Trace Selection

- ❖ 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

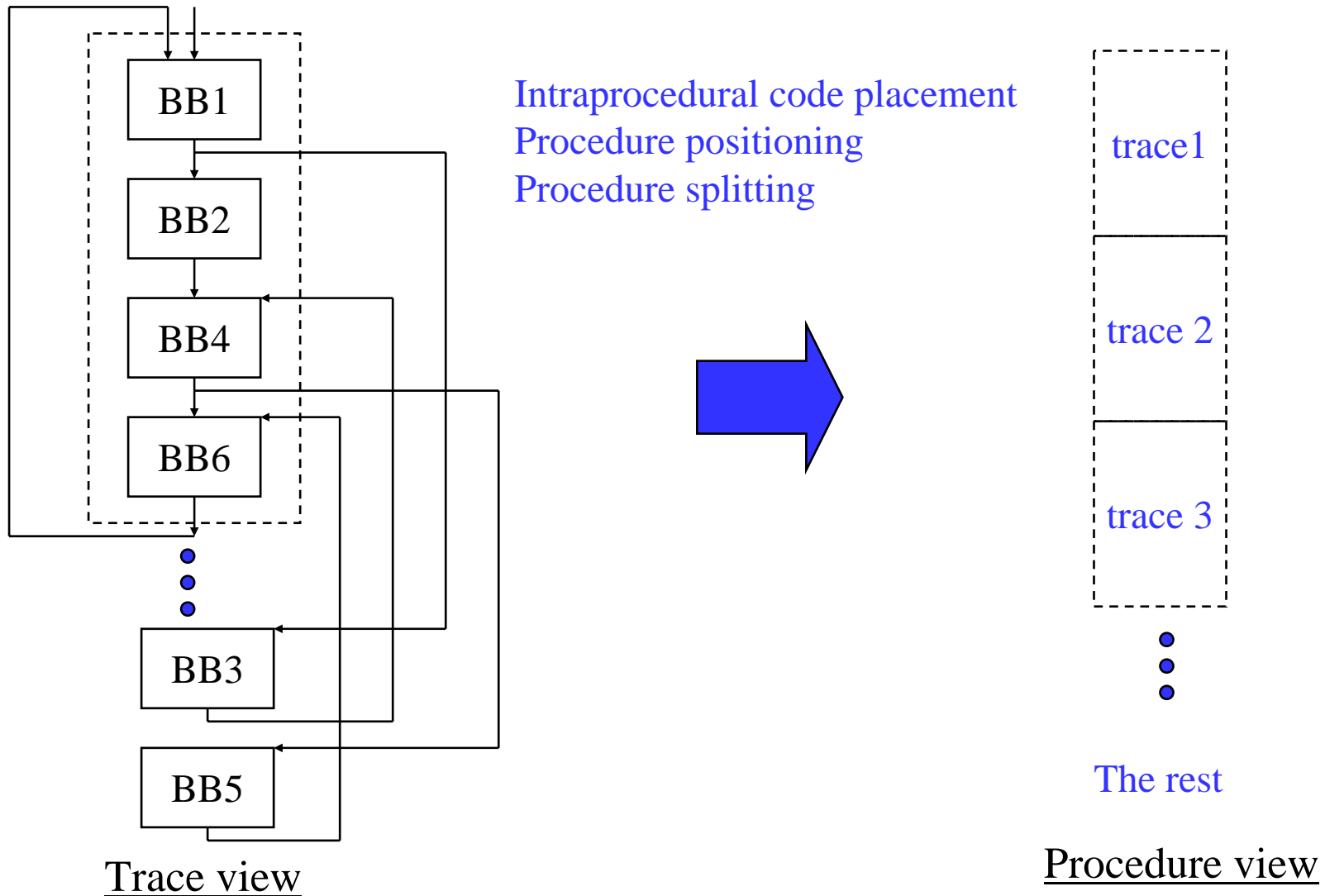


# Linearizing a Trace

---

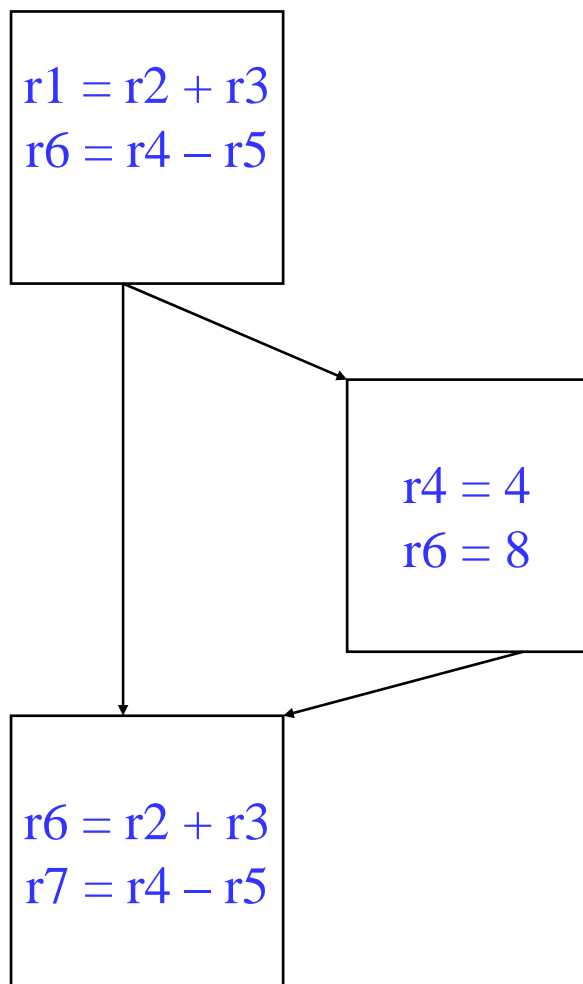


# Intelligent Trace Layout for Icache Performance



# Dataflow Analysis + Optimization

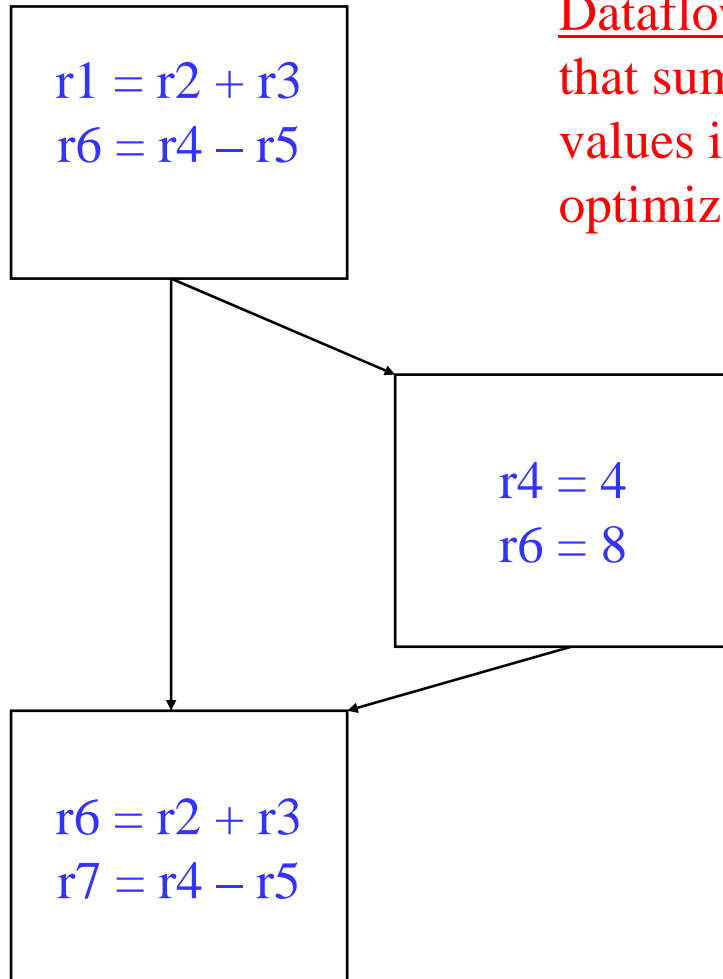
---



- ❖ Control flow analysis
  - » Treat BB as black box
  - » Just care about branches
- ❖ Now ...
  - » Start looking at operations in BBs
  - » What's computed and where
- ❖ Classical optimizations
  - » Make the computation more efficient
  - » Get rid of redundancy
  - » Simplify
- ❖ Ex: Common Subexpression Elimination
  - » Is  $r2 + r3$  redundant? What about  $r4 - r5$ ?
  - » What if there were 1000 BB's
  - » Dataflow analysis !!

# Dataflow Analysis Introduction

---



Dataflow analysis – Collection of information that summarizes the creation/destruction of values in a program. Used to identify legal optimization opportunities.

Pick an arbitrary point in the program

Which VRs contain useful data values?  
(liveness or upward exposed uses)

Which definitions may reach this point?  
(reaching defns)

Which definitions are guaranteed to reach this point?  
(available defns)

Which uses below are exposed?  
(downward exposed uses)

# Live Variable (Liveness) Analysis

---

- ❖ Defn: For each point  $p$  in a program and each variable  $y$ , determine whether  $y$  can be used before being redefined starting at  $p$
- ❖ Algorithm sketch
  - » For each BB,  $y$  is live if it is used before defined in the BB or it is live leaving the block
  - » Backward dataflow analysis as propagation occurs from uses upwards to defs
- ❖ 4 sets
  - » **USE** = set of external variables consumed in the BB
  - » **DEF** = set of variables defined in the BB
  - » **IN** = set of variables that are live at the entry point of a BB
  - » **OUT** = set of variables that are live at the exit point of a BB

# Liveness Example

---

$r1 = r2 + r3$   
 $r6 = r4 - r5$

$r2, r3, r4, r5$  are all live as they are consumed later,  $r6$  is dead as it is redefined later

$r4 = 4$   
 $r6 = 8$

$r4$  is dead, as it is redefined. So is  $r6$ .  $r2, r3, r5$  are live

$r6 = r2 + r3$   
 $r7 = r4 - r5$

What does this mean?  $r6 = r4 - r5$  is useless, it produces a dead value !!  
Get rid of it!

# Compute USE/DEF Sets For Each BB

---

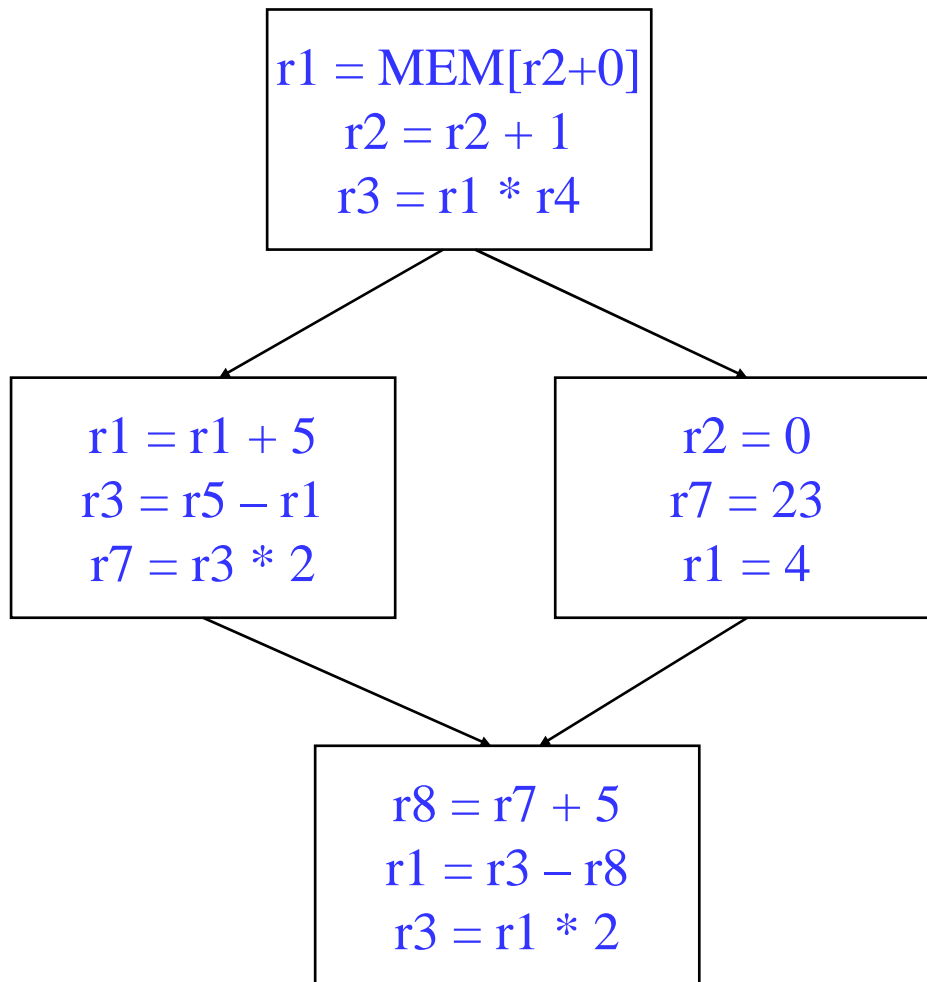
```
for each basic block in the procedure, X, do  
  DEF(X) = 0  
  USE(X) = 0  
  for each operation in sequential order in X, op, do  
    for each source operand of op, src, do  
      if (src not in DEF(X)) then  
        USE(X) += src  
      endif  
    endfor  
    for each destination operand of op, dest, do  
      DEF(X) += dest  
    endfor  
  endfor  
endfor
```

def is the union of all the LHS's  
use is all the VRs that are used before defined

---

# Example USE/DEF Calculation

---



# Compute IN/OUT Sets For All BBs

---

```
initialize IN(X) to 0 for all basic blocks X
change = 1
while (change) do
  change = 0
  for each basic block in procedure, X, do
    old_IN = IN(X)
    OUT(X) = Union(IN(Y)) for all successors Y of X
    IN(X) = USE(X) + (OUT(X) - DEF(X))
    if (old_IN != IN(X)) then
      change = 1
    endif
  endfor
endfor
```

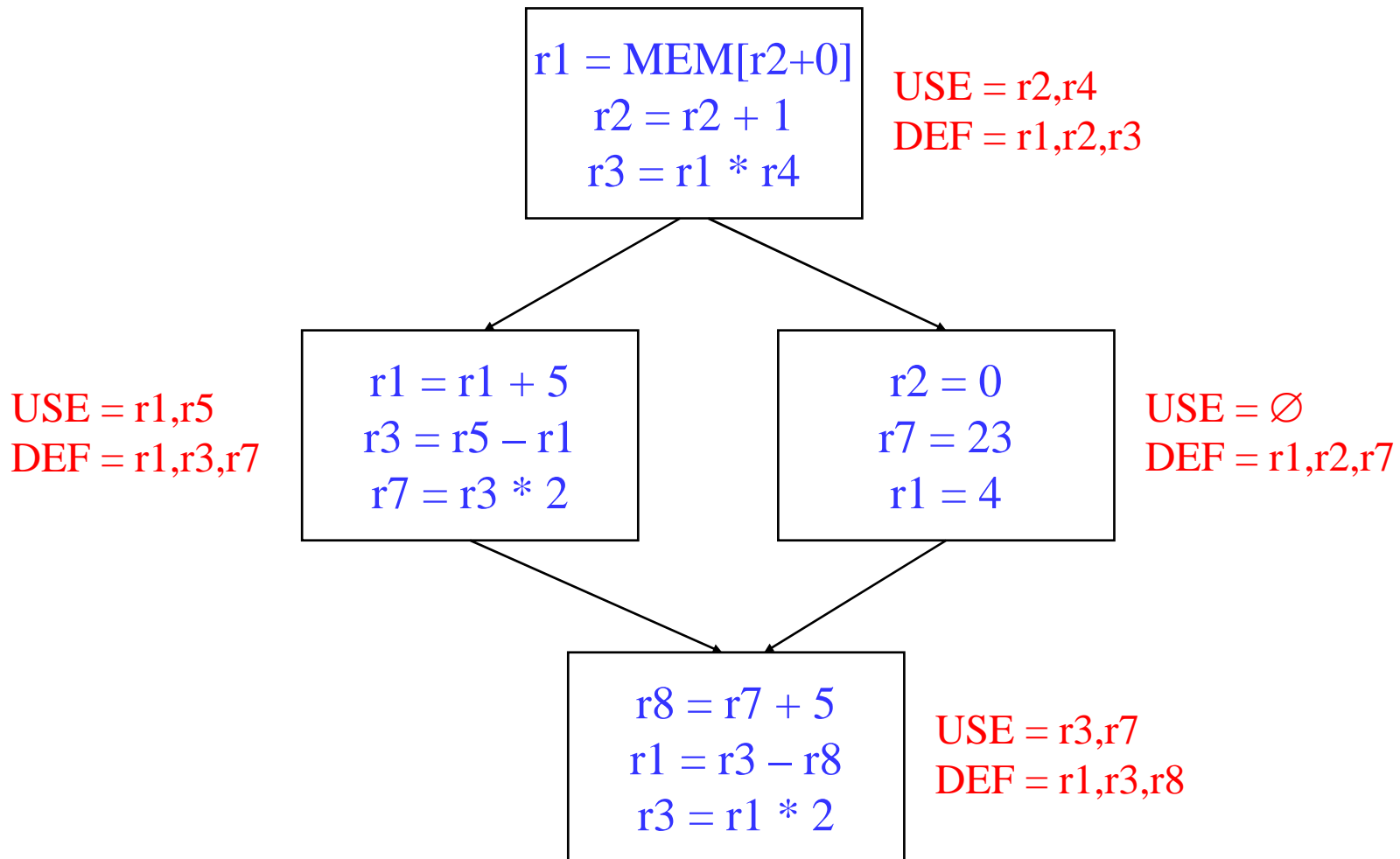
IN = set of variables that are live when the BB is entered

OUT = set of variables that are live when the BB is exited

---

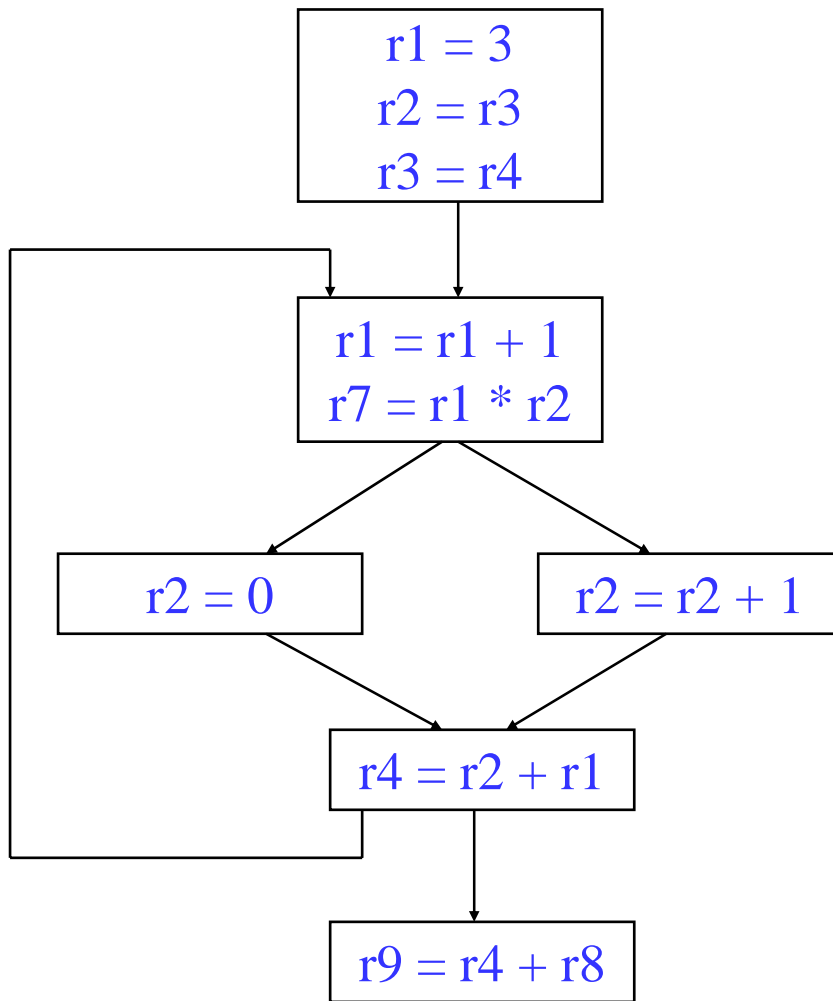
# Example IN/OUT Calculation

---



# Class Problem

---



Compute liveness, ie  
calculate USE/DEF  
calculate IN/OUT