Loop Optimizations

Compilers for High Performance Architectures

Class outline

- Loop – Body Optimizations
- Loop Unrolling

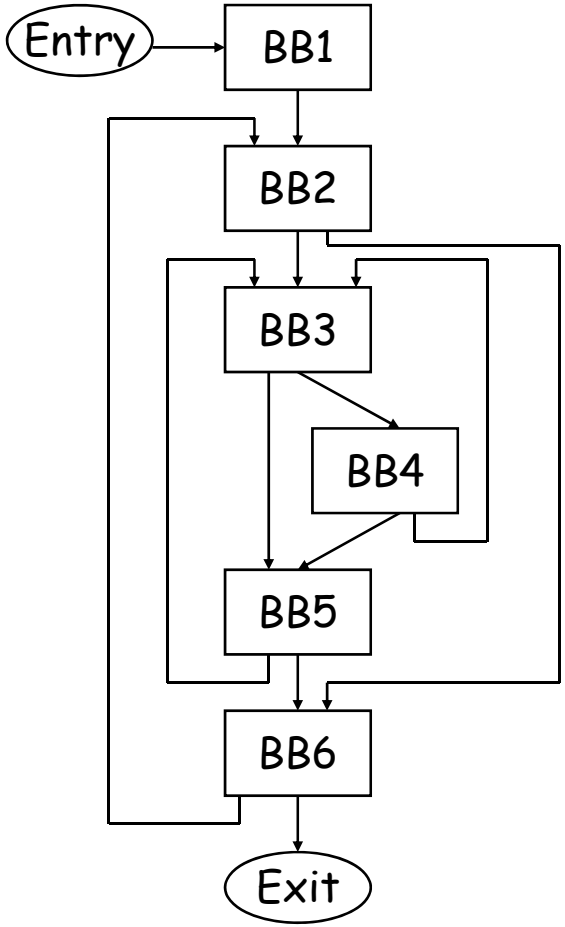
Loop - Body Optimizations

- The most important set of optimizations
  - Because programs spend so much time in loops
- Optimize given that you know a sequence of code will be repeatedly executed
- Optimizations
  - Invariant code removal
  - Global variable migration
  - Induction variable strength reduction
  - Induction variable elimination

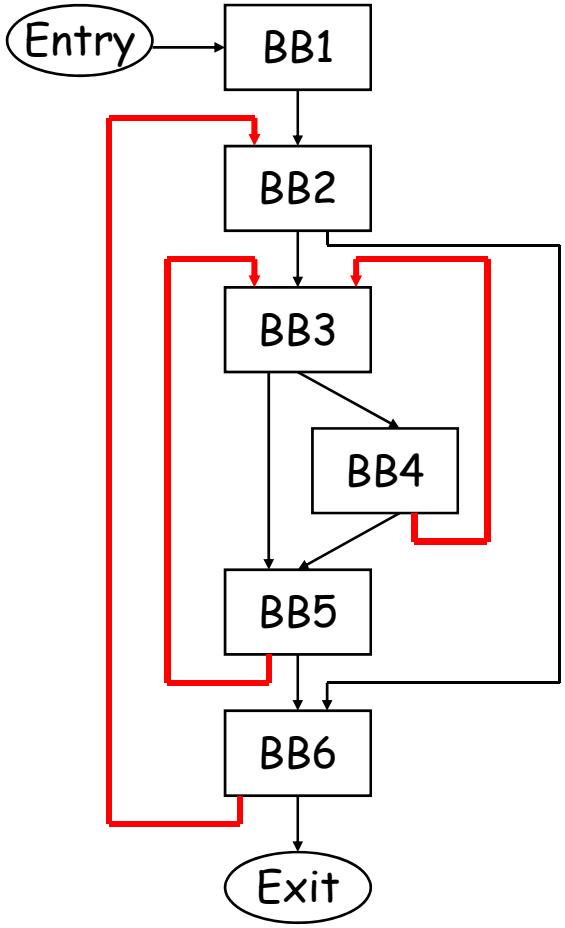
Loops

- Cycle suitable for optimization
- 2 properties
  - Single entry point called the header
    - Header dominates all blocks in the loop
  - Must be one way to iterate the loop (ie at least 1 path back to the header from within the loop) called a backedge
- Backedge detection
  - Edge, x→ y where the target (y) dominates the source (x)

Backedge Example



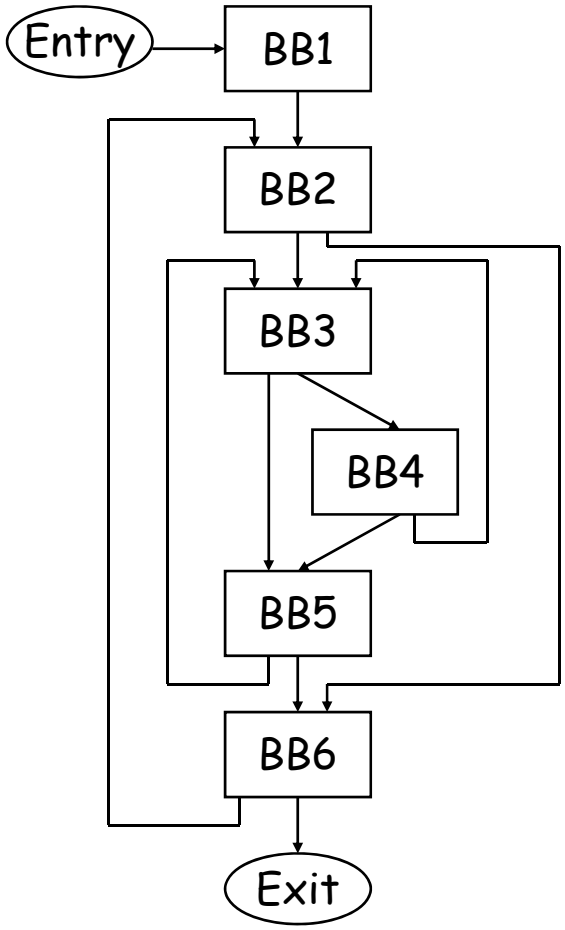
Backedge Example



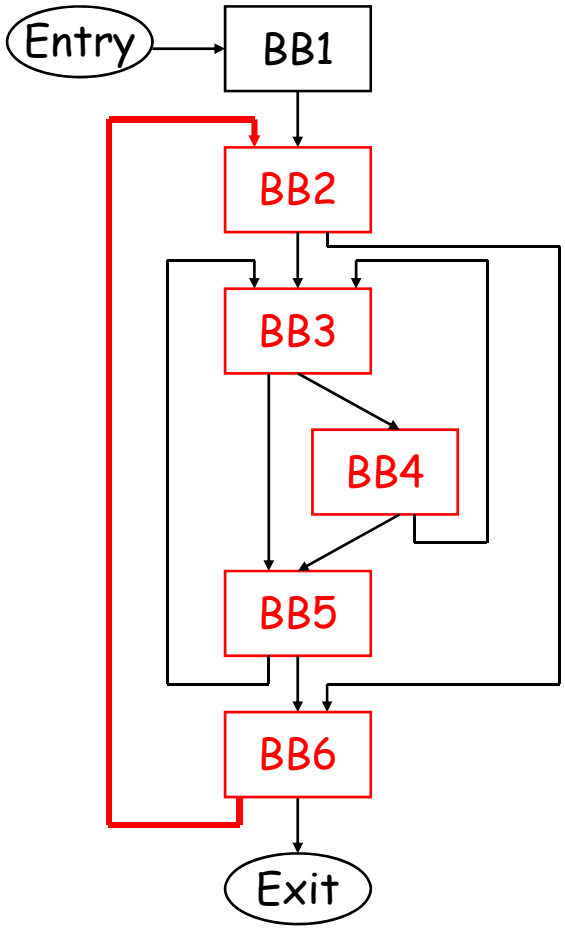
Loop Detection

- Identify all backedges using Dom info
- Each backedge (x → y) defines a loop
  - Loop header is the backedge target (y)
  - Loop BB – basic blocks that comprise the loop
    - All predecessor blocks of x for which control can reach x without going through y are in the loop
- Merge loops with the same header
  - i.e., a loop with 2 continues
  - LoopBackedge = LoopBackedge1 + LoopBackedge2
  - LoopBB = LoopBB1 + LoopBB2
- Important property
  - Header dominates all LoopBB

Loop Detection Example

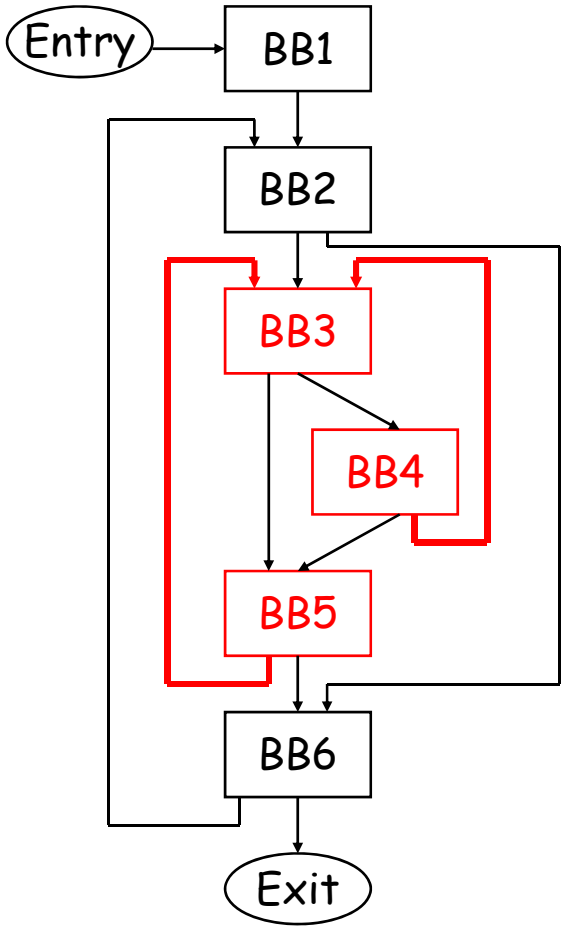


Loop Detection Example



Outer loop

Loop Detection Example



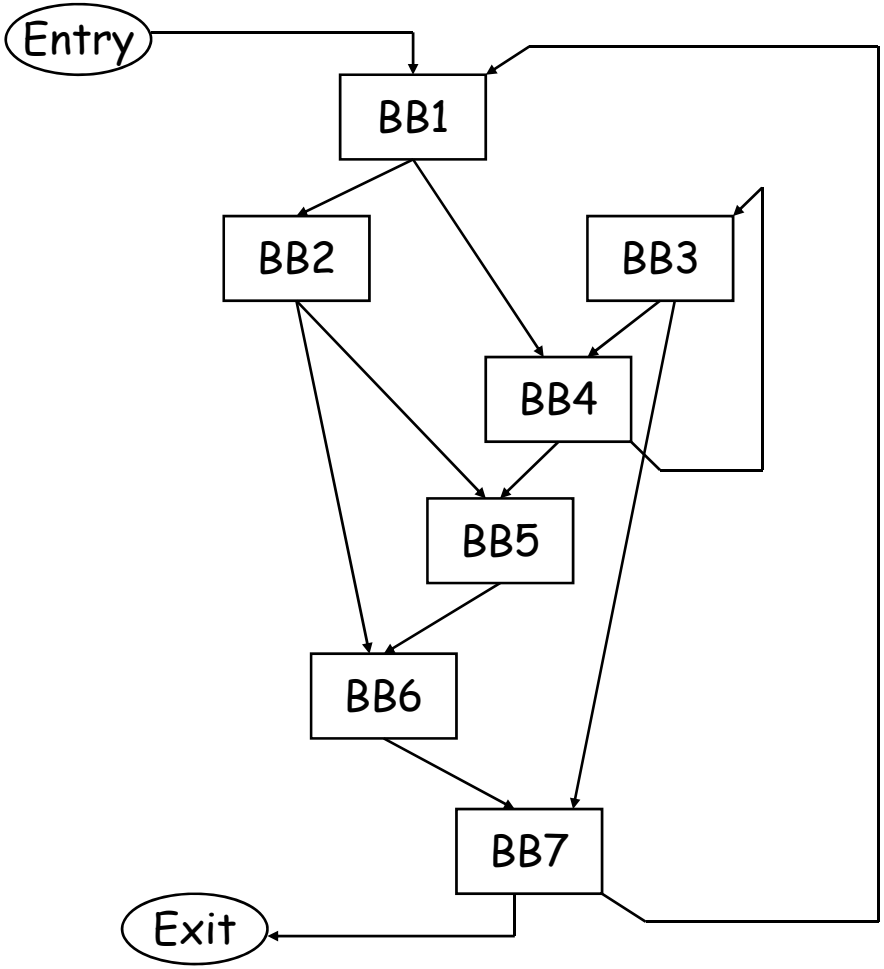
Inner loop
With 2 back edges

Class Problem

Find the loops

What are the header(s)?

What are the backedge(s)?



Important Parts of a Loop

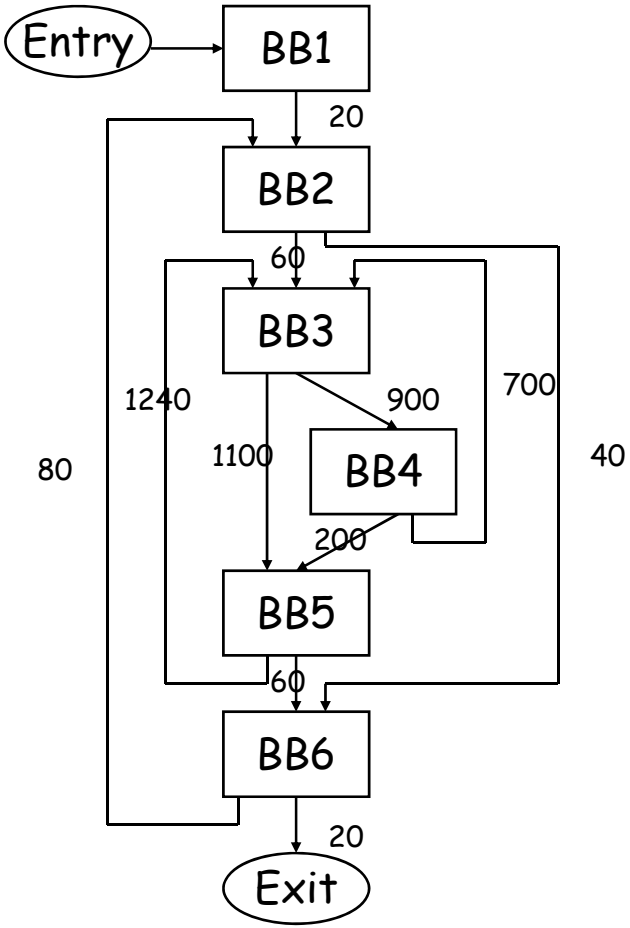
- Header, LoopBB
- Backedges, BackedgeBB
- Exitedges, ExitBB
  - For each LoopBB, examine each outgoing edge
  - If the edge is to a BB not in LoopBB, then its an exit
- Preheader (Preloop)
  - New block before the header (falls through to header)
  - Whenever you invoke the loop, preheader executed
  - Whenever you iterate the loop, preheader NOT executed
  - All edges entering header
    - Backedges – no change
    - All others, retarget to preheader
- Postexit (Postloop) - analogous

Characteristics of a Loop

- Nesting (generally within a procedure scope)
  - Inner loop – Loop with no loops contained within it
  - Outer loop – Loop contained within no other loops
  - Nesting depth
    - depth(outer loop) = 1
    - depth = depth(parent or containing loop) + 1
- Trip count (average trip count)
  - How many times (on average) does the loop iterate
  - for (I=0; I<100; I++) → trip count = 100
  - Ave trip count = weight(header) / weight(preheader)

Trip Count Calculation Example

Calculate the trip counts for all the loops in the graph



Loop Induction Variables

- Induction variables are variables such that every time they change 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

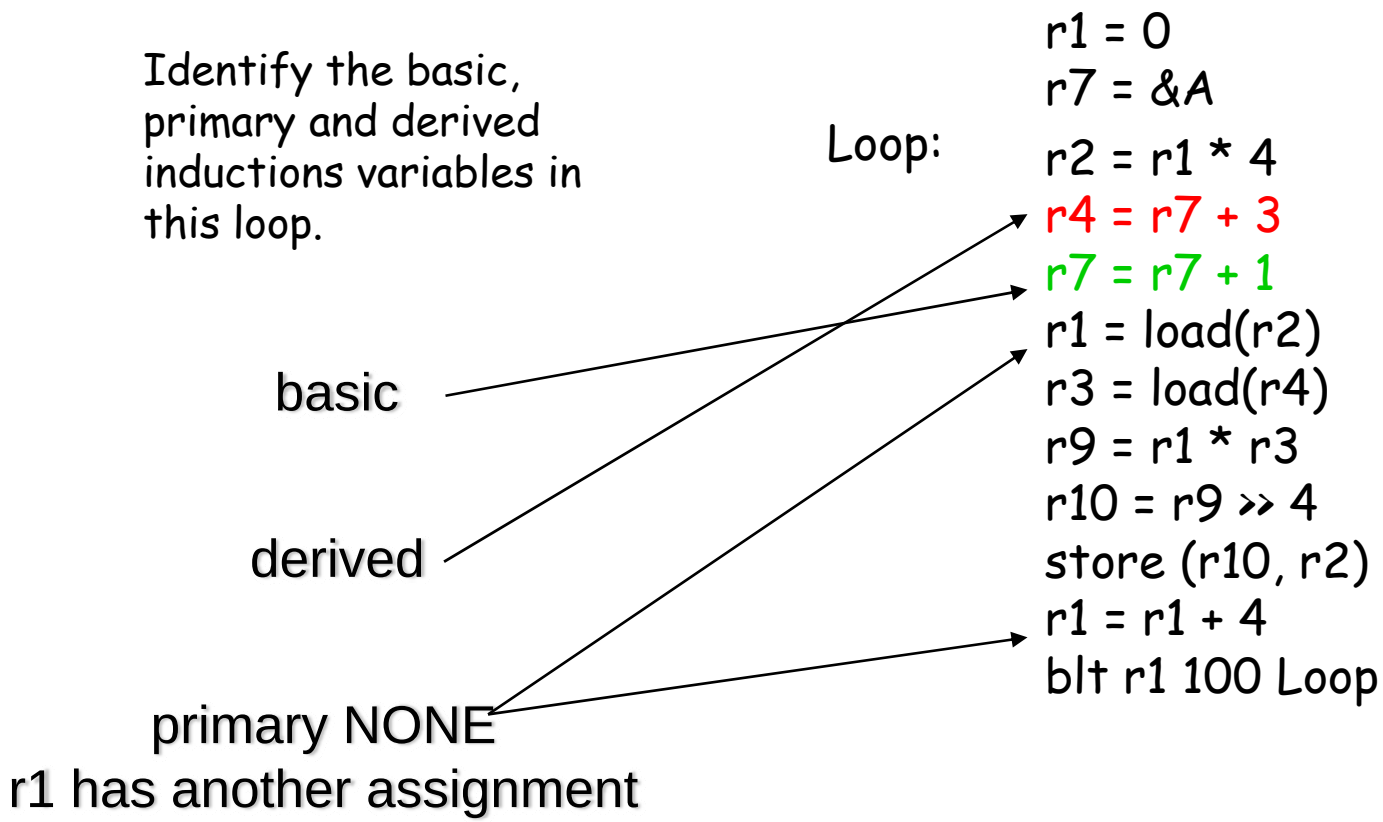
Class Problem

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
```

Class Problem

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



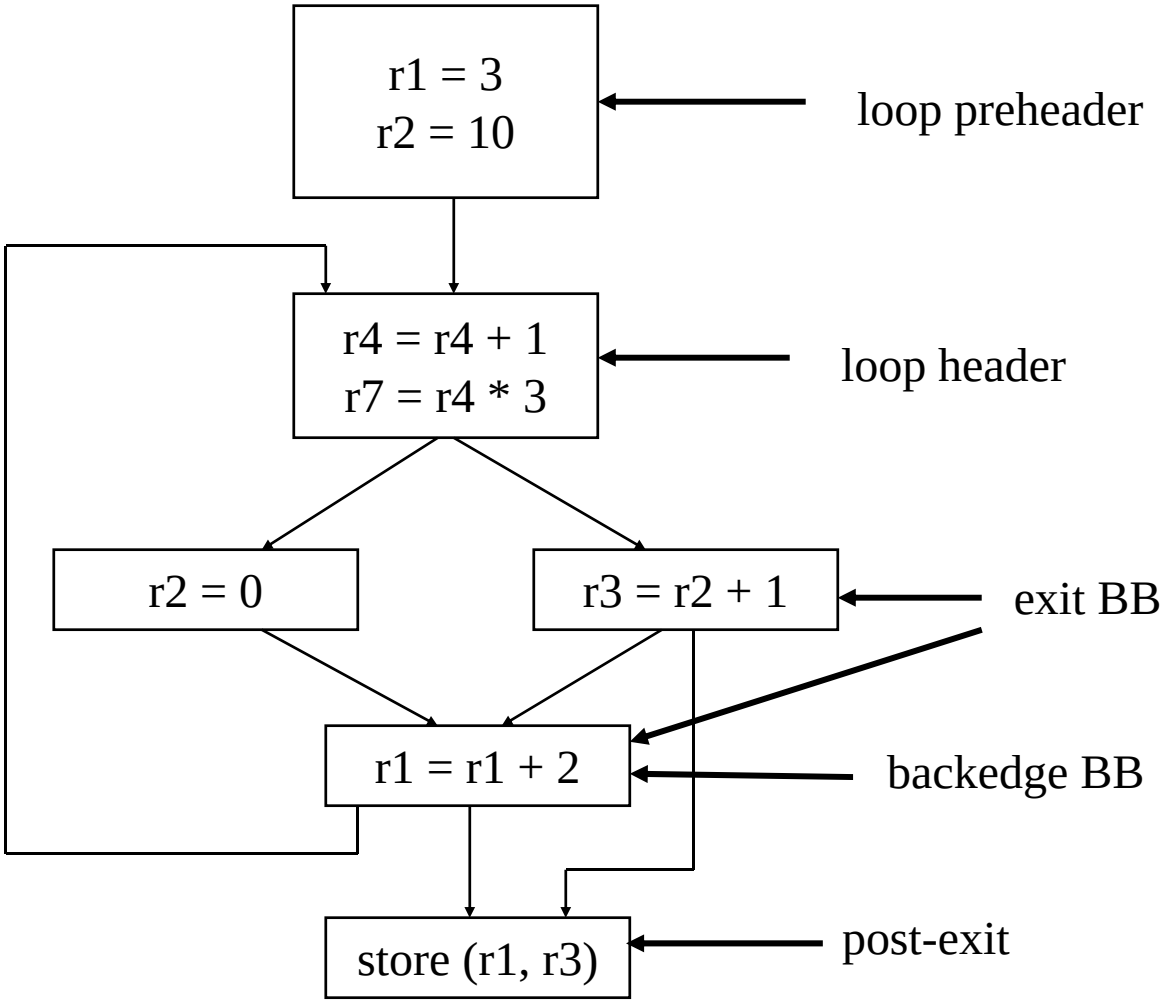
Class Problem

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
        r5 = load(r2)
        r3 = load(r4)
        r9 = r5 * r3
        r10 = r9 >> 4
        store (r10, r2)
        r1 = r1 + 4
        blt r1 100 Loop
```

Summary of Loop Terminology

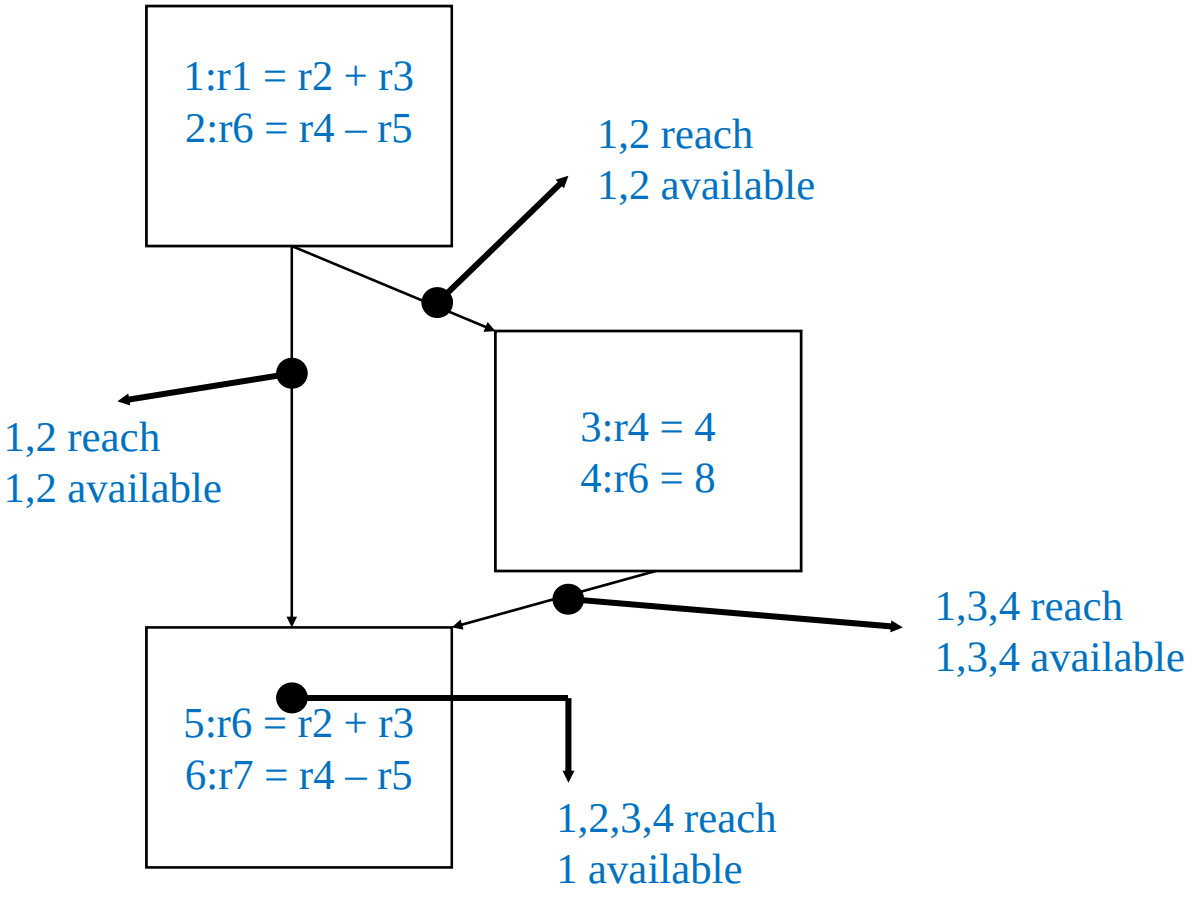
- r1, r4 are basic induction variables
- r7 is a derived induction variable



Available Definition Analysis

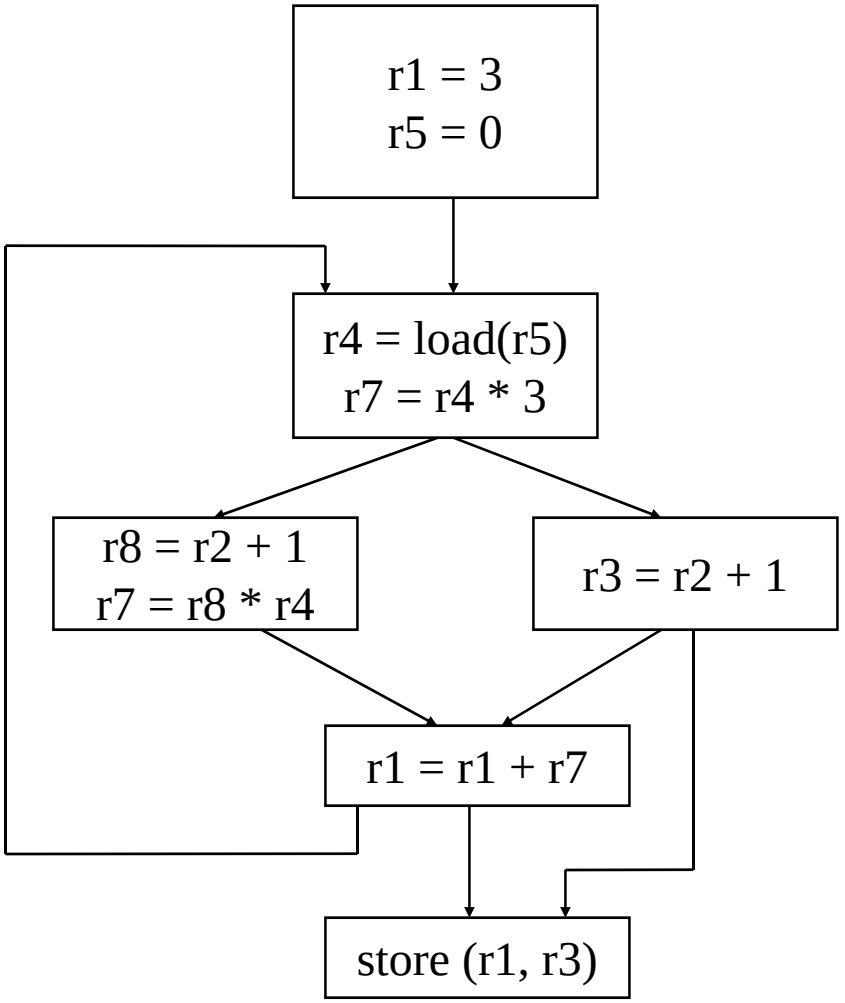
- A definition d is available at a point p if along all paths from d to p, d is not killed
- Remember, a definition of a variable is killed between 2 points when there is another definition of that variable along the path
  - r1 = r2 + r3 kills previous definitions of r1
- Algorithm
  - Forward dataflow analysis as propagation occurs from defs downwards
  - Use the Intersect function as the meet operator to guarantee the all-path requirement

Reaching vs Available Definitions



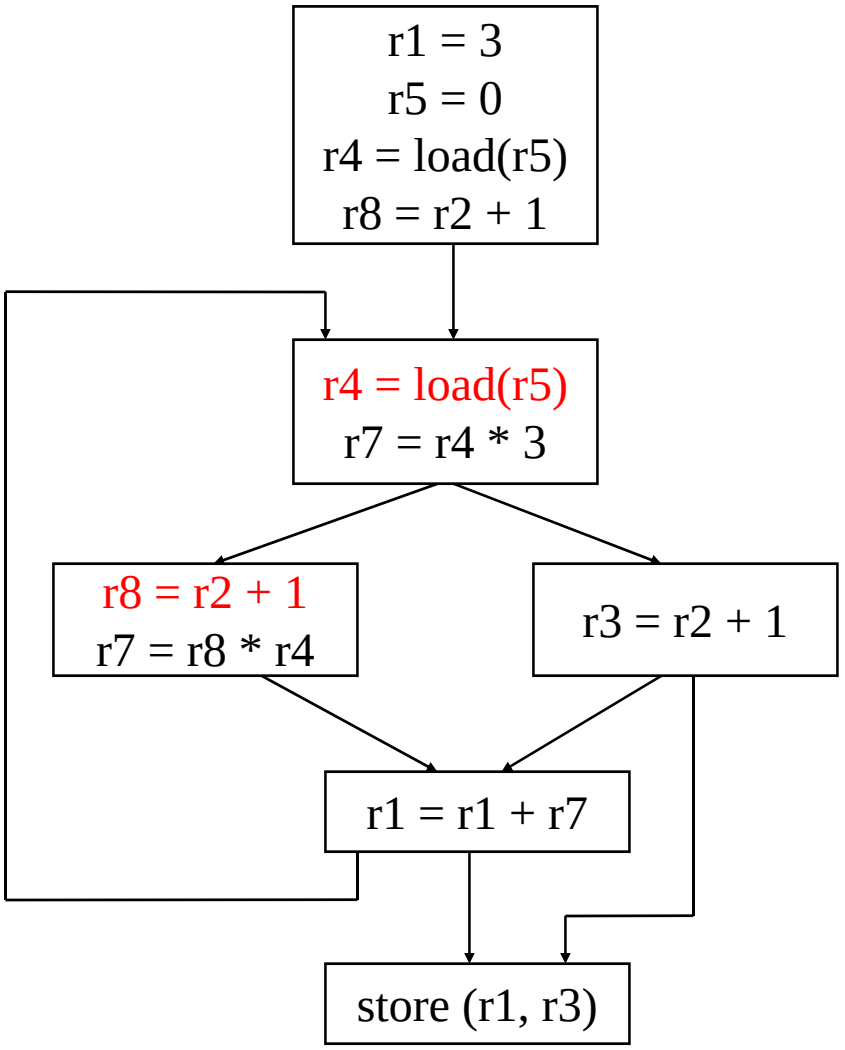
Invariant code removal

- Move operations whose source operands do not change within the loop to the loop preheader
  - Execute them only 1x per invocation of the loop
- Rules
  - X can be moved
  - src(X) not modified in loop body
  - X is the only op to modify dest(X)
  - for all uses of dest(X), X is in the available defs set
  - for all exit BB, if dest(X) is live on the exit edge, X is in the available defs set on the edge
  - if X not executed on every iteration, then X must provably not cause exceptions
  - if X is a load or store, then there are no writes to address(X) in loop



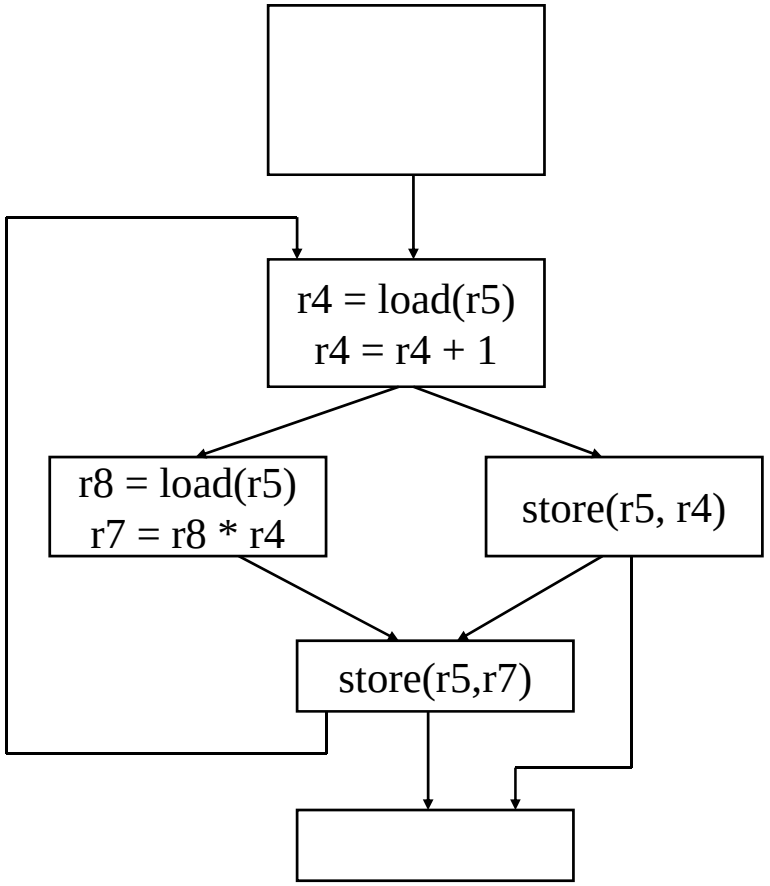
Invariant code removal

- Move operations whose source operands do not change within the loop to the loop preheader
  - Execute them only 1x per invocation of the loop
- Rules
  - X can be moved
  - src(X) not modified in loop body
  - X is the only op to modify dest(X)
  - for all uses of dest(X), X is in the available defs set
  - for all exit BB, if dest(X) is live on the exit edge, X is in the available defs set on the edge
  - if X not executed on every iteration, then X must provably not cause exceptions
  - if X is a load or store, then there are no writes to address(X) in loop



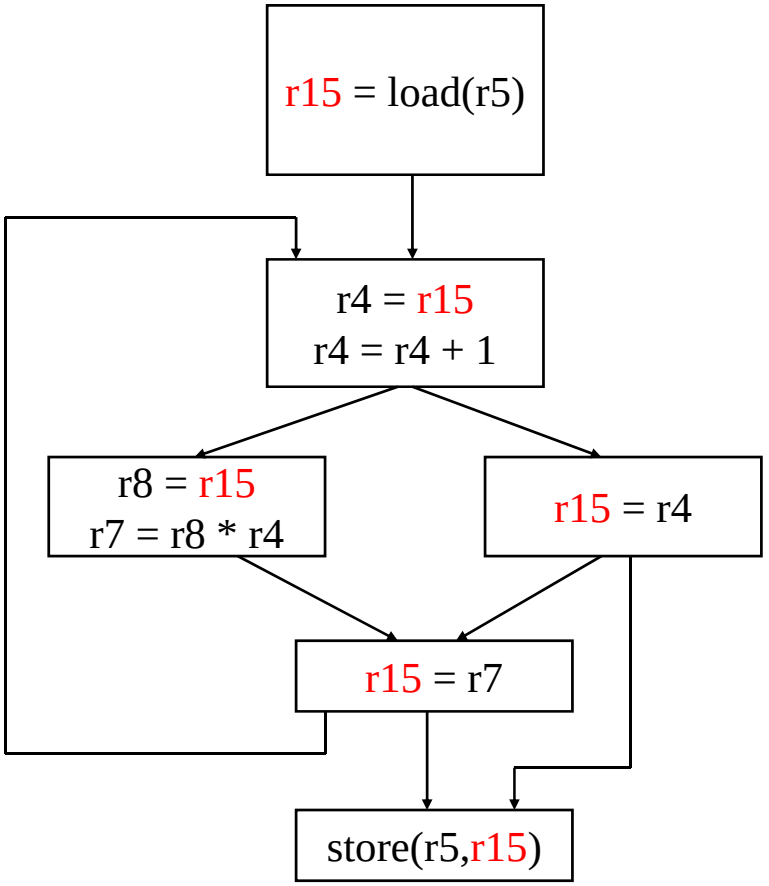
Global Variable Migration

- Assign a global variable temporarily to a register for the duration of the loop
  - Load in preheader
  - Store at exit points
- Rules
  - X is a load or store
  - address(X) not modified in the loop
  - if X not executed on every iteration, then X must provably not cause an exception
  - All memory ops in loop whose address can equal address(X) must always have the same address as X

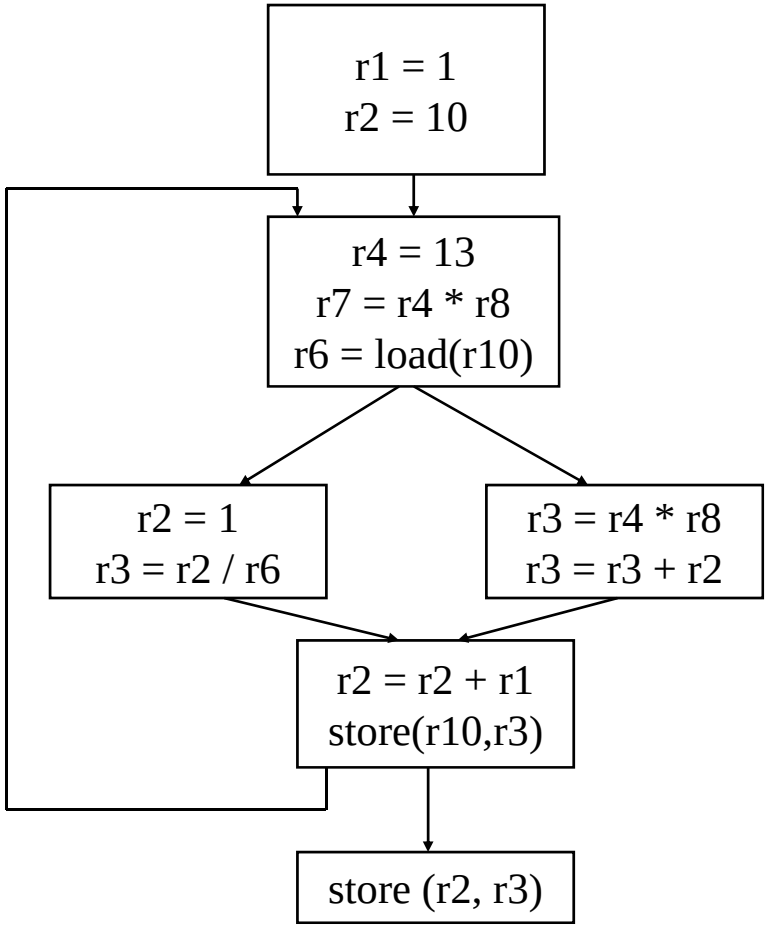


Global Variable Migration

- Assign a global variable temporarily to a register for the duration of the loop
  - Load in preheader
  - Store at exit points
- Rules
  - X is a load or store
  - address(X) not modified in the loop
  - if X not executed on every iteration, then X must provably not cause an exception
  - All memory ops in loop whose address can equal address(X) must always have the same address as X



Class Problem



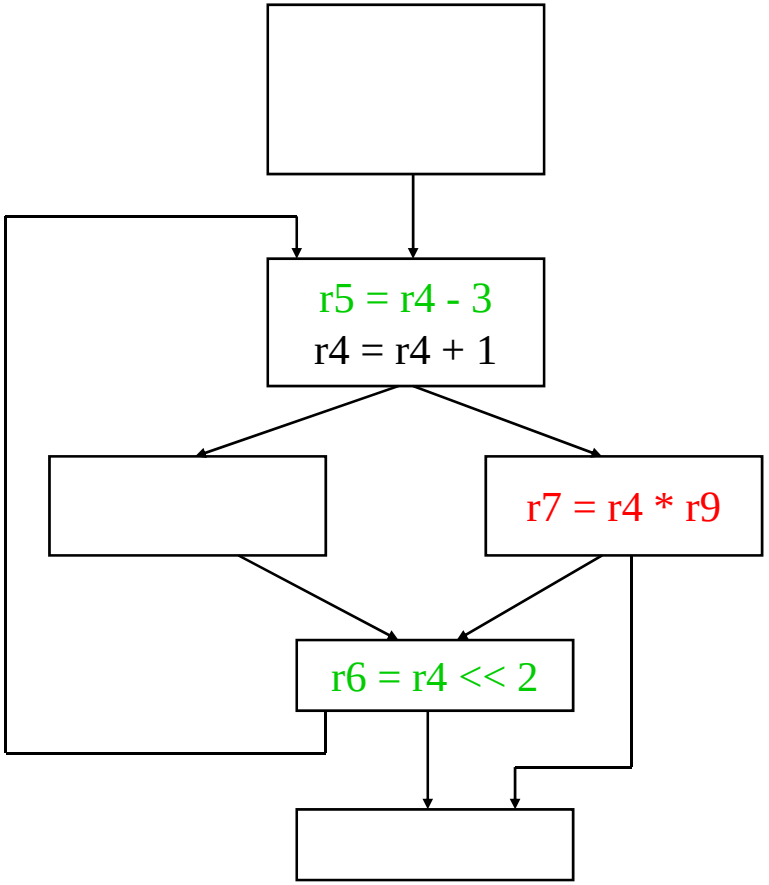
Optimize this applying

1. constant propagation
2. constant folding
3. strength reduction
4. dead code elimination
5. forward copy propagation
6. backward copy propagation
7. CSE
8. constant combining
9. operation folding
10. loop invariant removal
11. global variable migration

Induction Variable Strength Reduction

- Create basic induction variables from derived induction variables

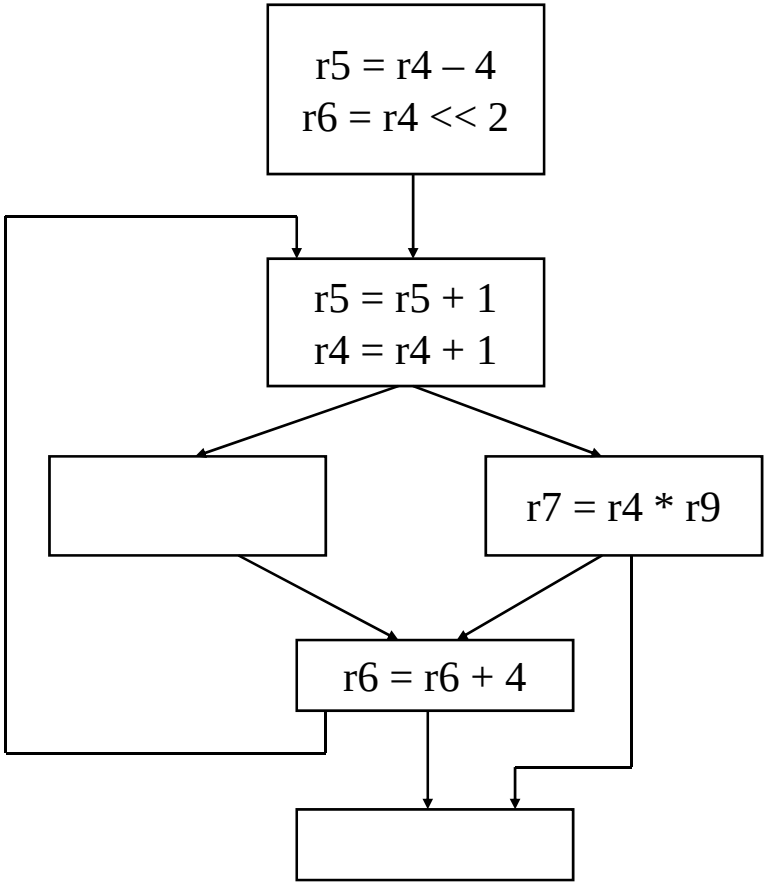
- Rules
  - X is a *, <<, + or – operation
  - src1(X) is a basic ind var
  - src2(X) is invariant
  - No other ops modify dest(X)
  - dest(X) != src(X) for all srcs
  - dest(X) is a register



Induction Variable Strength Reduction

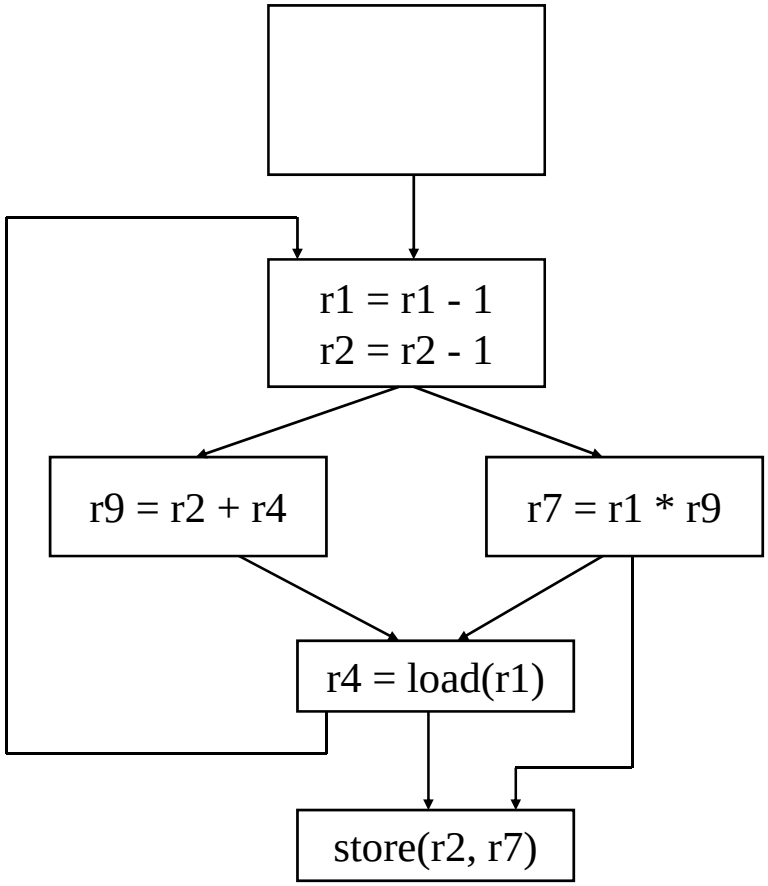
- Create basic induction variables from derived induction variables

- Rules
  - X is a *, <<, + or – operation
  - src1(X) is a basic ind var
  - src2(X) is invariant
  - No other ops modify dest(X)
  - dest(X) != src(X) for all srcs
  - dest(X) is a register



Induction Variable Elimination

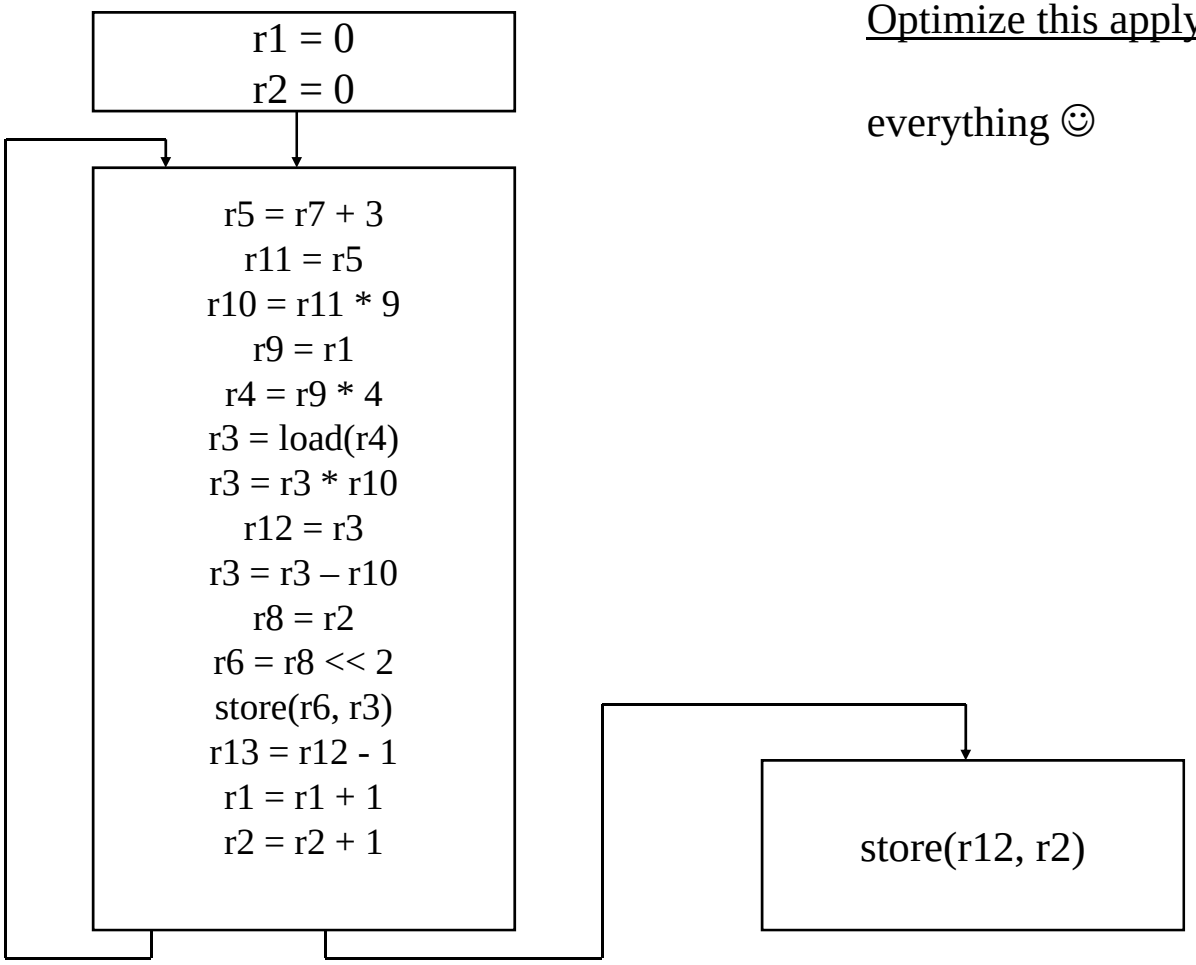
- Remove unnecessary basic induction variables from the loop by substituting uses with another BIV
- Rules (same init val, same inc)
  - Find 2 basic induction vars x,y
  - x,y in same family
    - incremented in same places
  - increments equal
  - initial values equal
  - x not live when you exit loop
  - for each BB where x is defined, there are no uses of x between first/last defn of x and last/first defn of y



Induction Variable Elimination (2)

- Several variants
  - 1. Trivial – induction variable that is never used except by the increments themselves, not live at loop exit
  - 2. Same increment, same initial value
  - 3. Same increment, initial values are a known constant offset from one another
  - 4. Same increment, know nothing about relation of initial values
  - 5. Different increments, know nothing about initial values
- The higher the number, the more complex the elimination
  - Also, the more expensive it is
  - 1,2 are basically free, so always should be done
  - 3-5 require preheader operations

Class Problem



Optimize this applying

everything ☺

Loop Unrolling

- Loop unrolling is a “region enlargement” technique
- Loop unrolling copies the loop body including the exit test
  - Most compilers remove exit tests when unrolling a loop by preconditioning / postconditioning:
    - To unroll by n, we add a pre-loop to handle trip-count mod n iterations, and the loop processes n loop bodies at a time
  - We cannot precondition loops with data dependent loop exits (e.g., while loops)
    - However, they can still be unrolled and optimized
  - Small counted loops can be unrolled completely
- Loop unrolling achievements
  - We can remove loop redundancies (compares, branches)
  - We can exploit inter-iteration ILP

Loop Unrolling Styles

loop

```
L:  if-- goto E
    body
    goto L
E:
```

unrolled by 4

```
L:  if-- goto E
    body
    if-- goto E
    body
    if-- goto E
    body
    if-- goto E
    body
    goto L
E:
```

pre-cond by 4

```
    if-- goto L
    body
    if-- goto L
    body
    if-- goto L
    body
L:  if-- goto E
    body
    body
    body
    body
    goto L
E:
```

post-cond by 4

```
L:  if-- goto X
    body
    body
    body
    body
    goto L
X:  if-- goto E
    body
    if--goto E
    body
    if-- goto E
    body
E:
```

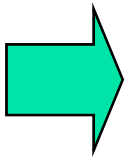
Removing Loop Dependences

- An unrolled loop has little parallelism between unrollings
  - Because of the dependencies on induction variables (IVs)
- A VLIW compiler must transform the loop to expose the available parallelism
  - Compilers for scalar machines don't have to, removing redundancies is the reason for unrollings
- Temporary renaming
  - Splits temporaries into disjoint use-def webs
  - By renaming IVs,and then copy propagating, we can remove those dependencies
- Copy propagation
  - Propagates a copied value to its uses
  - By propagating integer additions of a constant, we can generate independent secondary IVs

Optimizing Unrolled Loops

```
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
        r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop
```

unroll 3 times



```
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        ----------------
        r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter2   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        ----------------
        r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter3   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop
```

Unroll = replicate loop body n-1 times.

Hope to enable overlap of operation execution from different iterations

Not possible!

Register Renaming on Unrolled Loop

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ----------------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter2  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ----------------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter3  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ----------------
       r11 = load(r2)
       r13 = load(r4)
       r15 = r11 * r13
iter2  r6 = r6 + r15
       r2 = r2 + 4
       r4 = r4 + 4
       ----------------
       r21 = load(r2)
       r23 = load(r4)
       r25 = r21 * r23
iter3  r6 = r6 + r25
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

Register Renaming is Not Enough!

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ----------------
       r11 = load(r2)
       r13 = load(r4)
       r15 = r11 * r13
iter2  r6 = r6 + r15
       r2 = r2 + 4
       r4 = r4 + 4
       ----------------
       r21 = load(r2)
       r23 = load(r4)
       r25 = r21 * r23
iter3  r6 = r6 + r25
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

- Still not much overlap possible
- Problems
  - r2, r4, r6 sequentialize the iterations
  - Need to rename these
- 2 specialized renaming optis
  - Accumulator variable expansion (r6)
  - Induction variable expansion (r2, r4)

Accumulator Variable Expansion

```
        r16 = r26 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        ----------------
        r11 = load(r2)
        r13 = load(r4)
        r15 = r11 * r13
iter2   r16 = r16 + r15
        r2 = r2 + 4
        r4 = r4 + 4
        ----------------
        r21 = load(r2)
        r23 = load(r4)
        r25 = r21 * r23
iter3   r26 = r26 + r25
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop
        r6 = r6 + r16 + r26
```

- Accumulator variable
  - x = x + y or x = x – y
  - where y is loop variant!!
- Create n-1 temporary accumulators
- Each iteration targets a different accumulator
- Sum up the accumulator variables at the end
- May not be safe for floating-point values
- Generalization to reductions
  - Reductions are a recurrence of the form
    a = a op fn(i)
    where op is commutative and associative
- The compiler can rewrite a loop containing a reduction
  - The reduction becomes n interleaved reductions
  - The n results are combined on loop exit

Induction Variable Expansion

```
                r12 = r2 + 4, r22 = r2 + 8
                r14 = r4 + 4, r24 = r4 + 8
                r16 = r26 = 0
loop:           r1 = load(r2)
                r3 = load(r4)
                r5 = r1 * r3
iter1           r6 = r6 + r5
                r2 = r2 + 12
                r4 = r4 + 12
        ----------------------
                r11 = load(r12)
                r13 = load(r14)
                r15 = r11 * r13
iter2           r16 = r16 + r15
                r12 = r12 + 12
                r14 = r14 + 12
        ----------------------
                r21 = load(r22)
                r23 = load(r24)
                r25 = r21 * r23
iter3           r26 = r26 + r25
                r22 = r22 + 12
                r24 = r24 + 12
                if (r4 < 400) goto loop
                r6 = r6 + r16 + r26
```

- Induction variable
  - x = x + y or x = x – y
  - where y is loop invariant!!
- Create n-1 additional induction variables
- Each iteration uses and modifies a different induction variable
- Initialize induction variables to init, init+step, init+2*step, etc.
- Step increased to n*original step
- Now iterations are completely independent !!

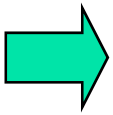
Better Induction Variable Expansion

```
        r16 = r26 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5

        ----------------
        r11 = load(r2+4)
        r13 = load(r4+4)
iter2   r15 = r11 * r13
        r16 = r16 + r15

        ----------------
        r21 = load(r2+8)
        r23 = load(r4+8)
iter3   r25 = r21 * r23
        r26 = r26 + r25
        r2 = r2 + 12
        r4 = r4 + 12
        if (r4 < 400) goto loop
        r6 = r6 + r16 + r26
```

- With base+displacement addressing, often don't need additional induction variables
  - Just change offsets in each iterations to reflect step
  - Change final increments to n * original step

Class Problem

```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```



```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```

Optimize the unrolled loop

Renaming
Tree height reduction
Ind/Acc expansion

How much unrolling?

- More unrolling exposes more inter-iteration ILP
- More unrolling decreases the overall loop overhead
  - Compare, branches, IV increments, time to fill and drain the pipelines, etc.
- However:
  - Unrolling increases code size (and Icache pressure)
  - Unrolling increases register pressure
    - If we have to spill, at some point performance degrades
  - Unrolling increases compile time
  - We don't need to expose more ILP than we can exploit
- Finding the "optimal unrolling" is practically unsolvable
  - A search technique (trial-and-error) is very expensive
- Choice of unrolling amount uses heuristics
  - Usually: a few "canned recipes" for different optimizations