

# Dataflow III: Local, Global and Loop Optimizations

---

EECS 483 – Lecture 25

University of Michigan

Monday, December 4, 2006

# Announcements and Reading

---

## ❖ Schedule

- » Today 12/4 – Finish optimizations
- » Wednes 12/6 – Register allocation, Exam 2 review
- » Mon 12/11 – Exam 2 in class
- » Wednes 12/13 – No class (Project 3 due)

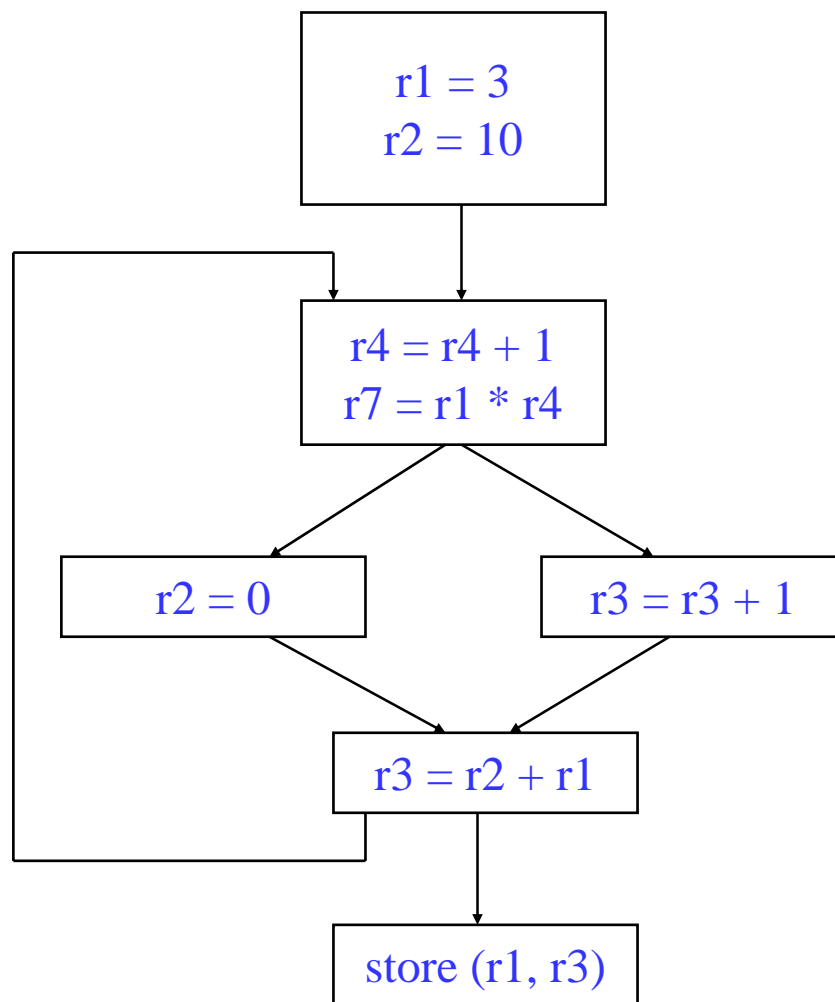
## ❖ Reading for today's class

- » 10.7

# From Last Time - Dead Code Elimination

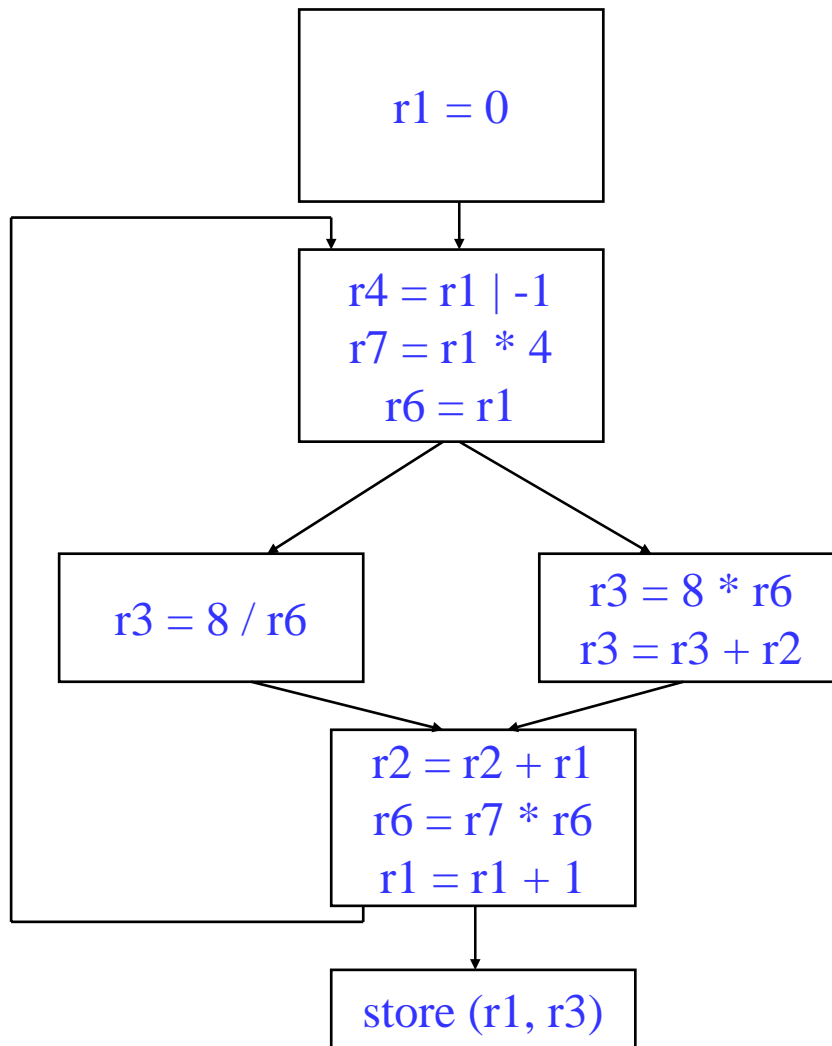
---

- ❖ Remove any operation whose result is never consumed
- ❖ Rules
  - » X can be deleted
    - no stores or branches
  - » DU chain empty or dest not live
- ❖ This misses some dead code!!
  - » Especially in loops
  - » Critical operation
    - store or branch operation
  - » Any operation that does not directly or indirectly feed a critical operation is dead
  - » Trace UD chains backwards from critical operations
  - » Any op not visited is dead



# Class Problem

---



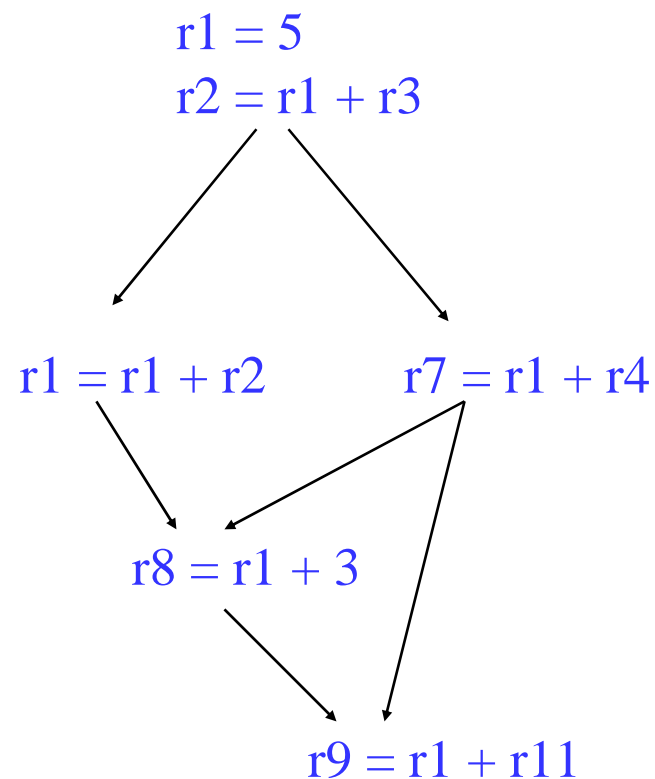
Optimize this applying

1. constant folding
2. strength reduction
3. dead code elimination

# Constant Propagation

---

- ❖ Forward propagation of moves of the form
  - »  $rx = L$  (where  $L$  is a literal)
  - » Maximally propagate
  - » Assume no instruction encoding restrictions
- ❖ When is it legal?
  - » SRC: Literal is a hard coded constant, so never a problem
  - » **DEST: Must be available**
    - Guaranteed to reach
    - May reach not good enough



# Local Constant Propagation

---

- ❖ Consider 2 ops, X and Y in a BB, X is before Y
  - » 1. X is a move
  - » 2.  $\text{src1}(X)$  is a literal
  - » 3. Y consumes  $\text{dest}(X)$
  - » 4. There is no definition of  $\text{dest}(X)$  between X and Y
    - Defn is locally available!
  - » 5. Be careful if  $\text{dest}(X)$  is SP, FP or some other special register – If so, no subroutine calls between X and Y

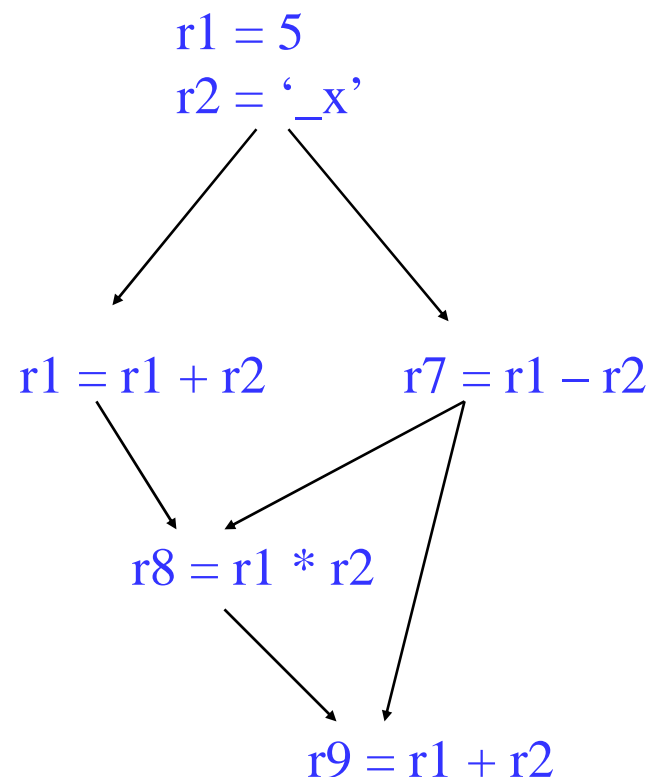
1:  $r1 = 5$   
2:  $r2 = \text{'\_x'}$   
3:  $r3 = 7$   
4:  $r4 = r4 + r1$   
5:  $r1 = r1 + r2$   
6:  $r1 = r1 + 1$   
7:  $r3 = 12$   
8:  $r8 = r1 - r2$   
9:  $r9 = r3 + r5$   
10:  $r3 = r2 + 1$   
11:  $r10 = r3 - r1$

# Global Constant Propagation

---

❖ Consider 2 ops, X and Y in different BBs

- » 1. X is a move
- » 2.  $\text{src1}(X)$  is a literal
- » 3. Y consumes  $\text{dest}(X)$
- » 4. X is in  $\text{adef\_IN}(\text{BB}(Y))$
- » 5.  $\text{dest}(X)$  is not modified between the top of  $\text{BB}(Y)$  and Y
  - Rules 4/5 guarantee X is available
- » 6. If  $\text{dest}(X)$  is SP/FP/..., no subroutine call between X and Y

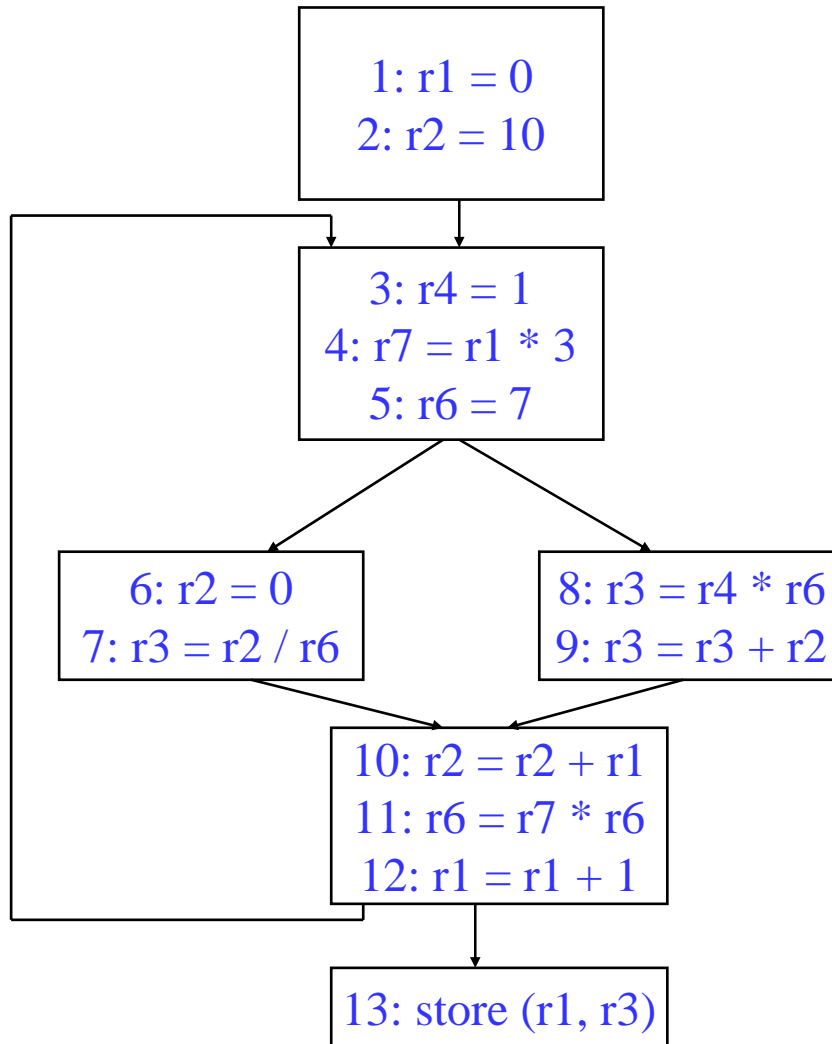


Note: checks for subroutine calls whenever SP/FP/etc. are involved is required for all optis. I will omit the check from here on!

---

# Class Problem

---

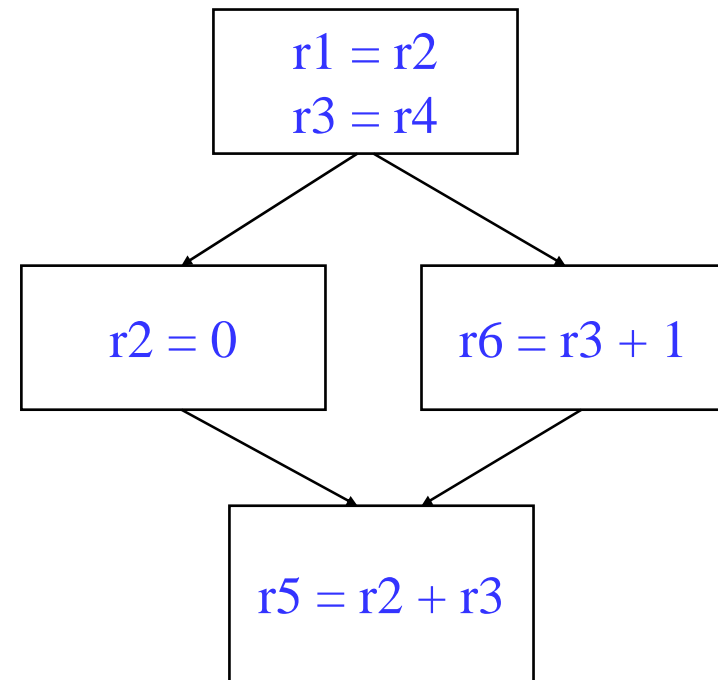


- Optimize this applying
1. constant propagation
  2. constant folding
  3. dead code elimination

# Global Forward Copy Propagation

---

- ❖ Forward propagation of the RHS of moves
  - » X:  $r1 = r2$
  - » ...
  - » Y:  $r4 = r1 + 1 \rightarrow r4 = r2 + 1$
- ❖ Benefits
  - » Reduce chain of dependences
  - » Possibly eliminate the move
- ❖ Rules (ops X and Y)
  - » X is a move
  - »  $\text{src1}(X)$  is a register
  - » Y consumes  $\text{dest}(X)$
  - » X.dest is an available def at Y
  - » X.src1 is an available expr at Y



# Backward Copy Propagation

---

❖ Backward prop. of the LHS of moves

» X:  $r1 = r2 + r3 \rightarrow r4 = r2 + r3$

» ...

»  $r5 = r1 + r6 \rightarrow r5 = r4 + r6$

» ...

» Y:  $r4 = r1 \rightarrow \text{noop}$

❖ Rules (ops X and Y in same BB)

»  $\text{dest}(X)$  is a register

»  $\text{dest}(X)$  not live out of  $\text{BB}(X)$

» Y is a move

»  $\text{dest}(Y)$  is a register

» Y consumes  $\text{dest}(X)$

»  $\text{dest}(Y)$  not consumed in  $(X...Y)$

»  $\text{dest}(Y)$  not defined in  $(X...Y)$

» There are no uses of  $\text{dest}(X)$  after the first redefinition of  $\text{dest}(Y)$

$r1 = r8 + r9$

$r2 = r9 + r1$

$r4 = r2$

$r6 = r2 + 1$

$r9 = r1$

$r10 = r6$

$r5 = r6 + 1$

$r4 = 0$

$r8 = r2 + r7$

# Local Common Subexpression Elimination

---

- ❖ Eliminate recomputation of an expr

- » X:  $r1 = r2 * r3$

- »  $\rightarrow r100 = r1$

- » ...

- » Y:  $r4 = r2 * r3 \rightarrow r4 = r100$

- ❖ Benefits

- » Reduce work

- » Moves can get copy propagated

- ❖ Rules (ops X and Y)

- » X and Y have the same opcode

- »  $\text{src}(X) = \text{src}(Y)$ , for all srcs

- » for all  $\text{srcs}(X)$  no defs of  $\text{src}_i$  in  $[X \dots Y)$

- » if X is a load, then there is no store that may write to  $\text{address}(X)$  between X and Y

$r1 = r2 + r3$

$r4 = r4 + 1$

$r1 = 6$

$r6 = r2 + r3$

$r2 = r1 - 1$

$r6 = r4 + 1$

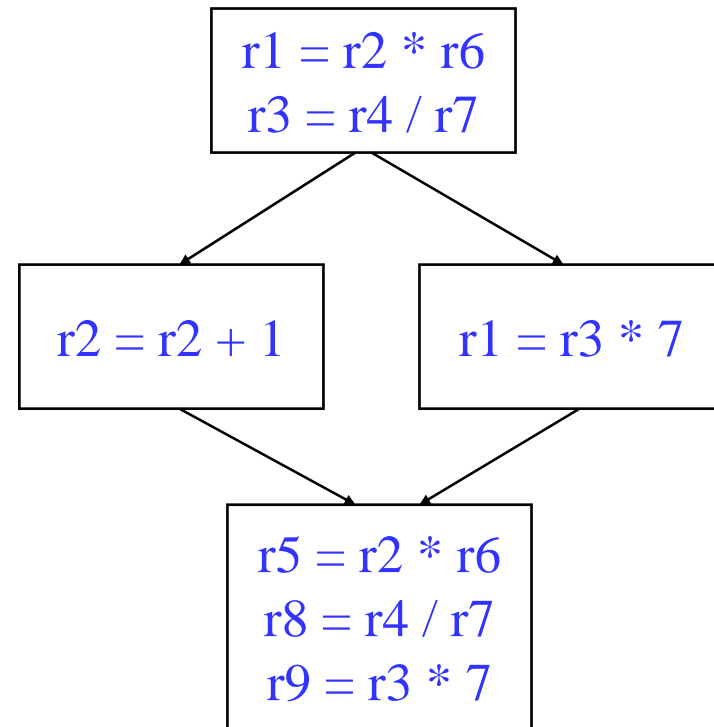
$r7 = r2 + r3$

# Global CSE

---

## ❖ Rules (ops X and Y)

- » X and Y have the same opcode
- »  $\text{src}(X) = \text{src}(Y)$ , for all srcs
- »  **$\text{expr}(X)$  is available at Y**
- » if X is a load, then there is no store that may write to  $\text{address}(X)$  along any path between X and Y



if op is a load, call it redundant load elimination rather than CSE

# Class Problem

---

```
r1 = 9
r4 = 4
r5 = 0
r6 = 16
r2 = r3 * r4
r8 = r2 + r5
r9 = r3
r7 = load(r2)
r5 = r9 * r4
r3 = load(r2)
r10 = r3 / r6
store (r8, r7)
r11 = r2
r12 = load(r11)
store(r12, r3)
```

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

# Loop Optimizations

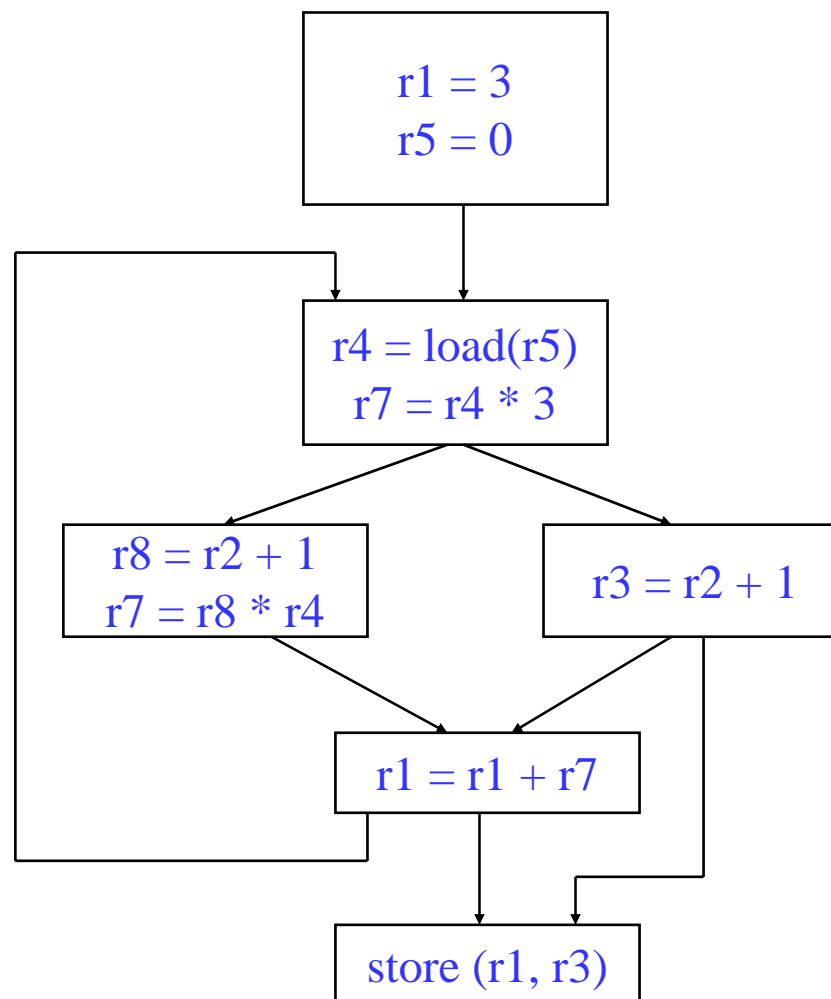
---

- ❖ **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
- ❖ Optis
  - » Invariant code removal
  - » Global variable migration
  - » Induction variable strength reduction
  - » Induction variable elimination

# 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
  - » Be careful with memory operations!
  - » Be careful with ops not executed every iteration

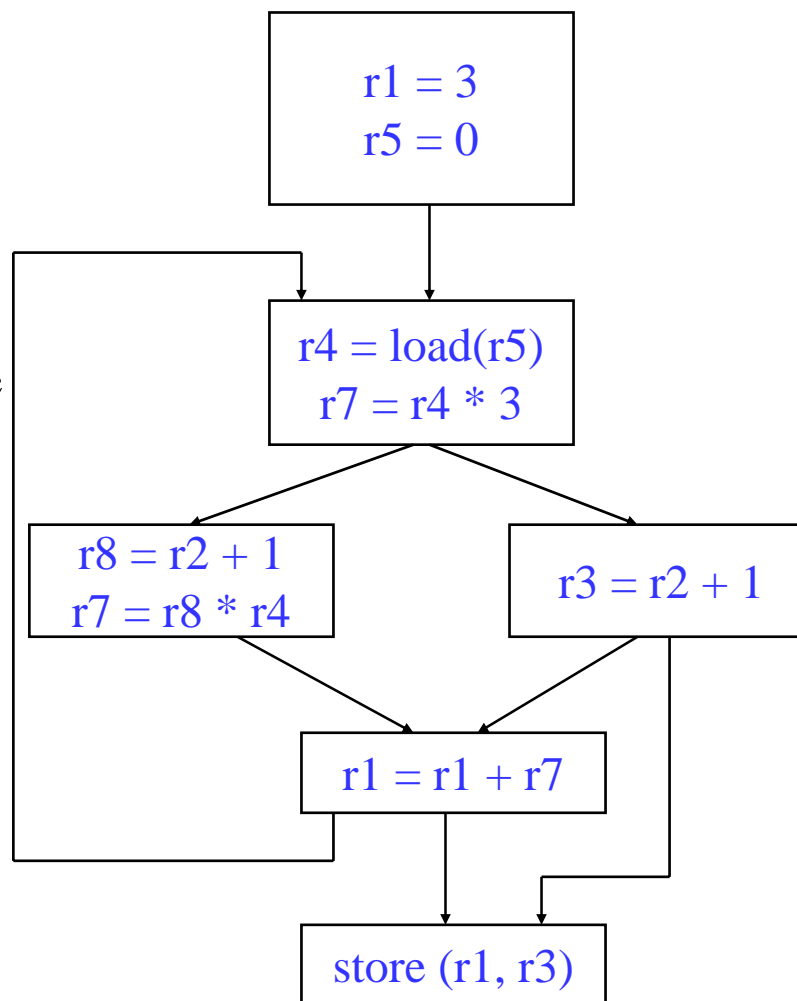


# Invariant Code Removal (2)

---

## ❖ Rules

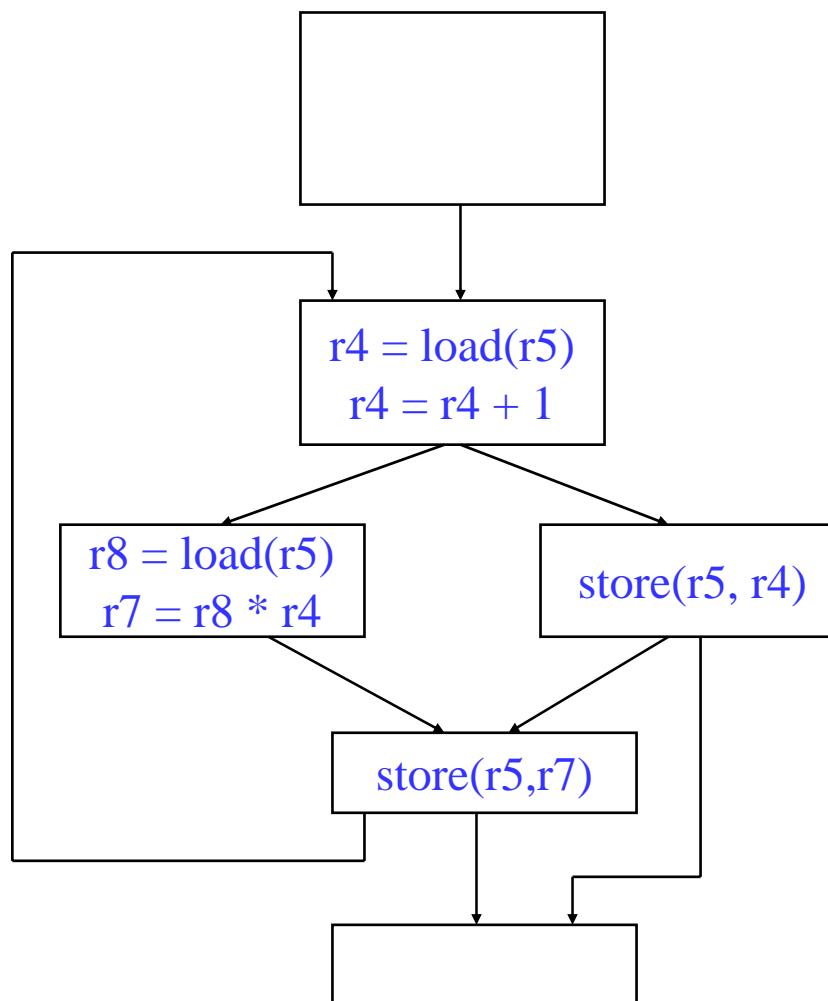
- » X can be moved
- »  $\text{src}(X)$  not modified in loop body
- » X is the only op to modify  $\text{dest}(X)$
- » for all uses of  $\text{dest}(X)$ , X is in the available defs set
- » for all exit BB, if  $\text{dest}(X)$  is live on the exit edge, X is in the available defs set on the edge
- » 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  $\text{address}(X)$  in loop



# Global Variable Migration

---

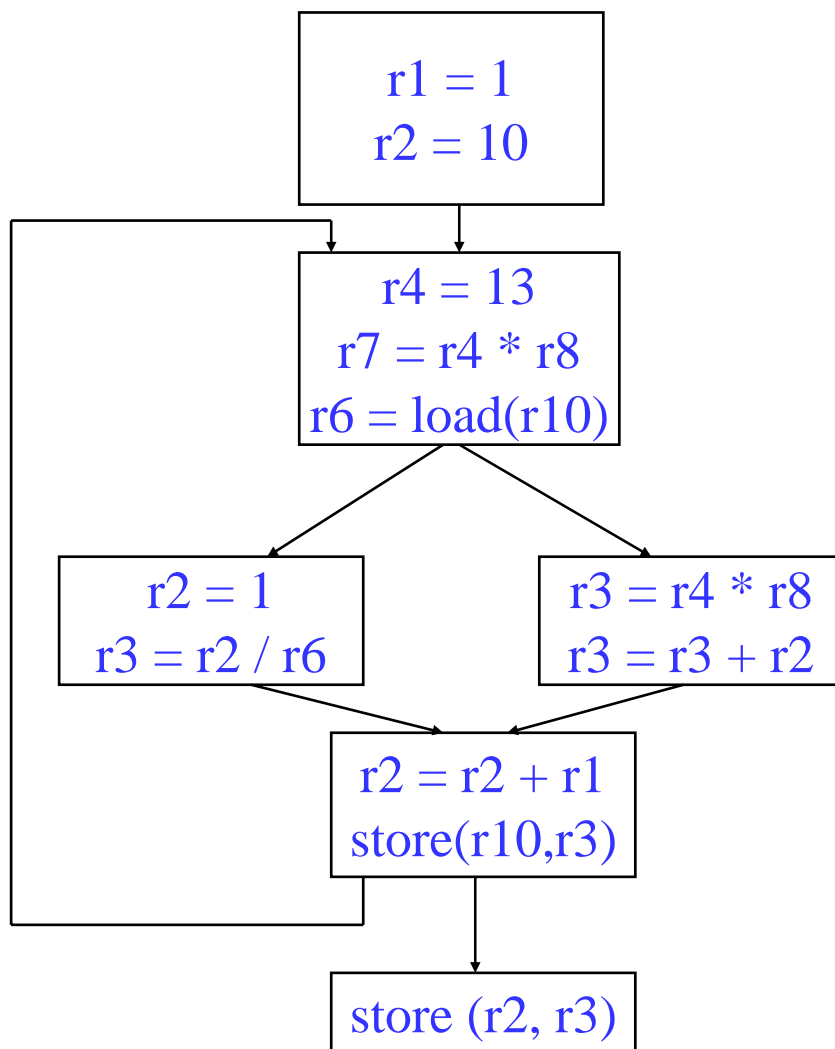
- ❖ Assign a global variable temporarily to a register for the duration of the loop
  - » Load in preheader
  - » Store at exit points
- ❖ Rules
  - » X is a load or store
  - »  $\text{address}(X)$  not modified in the loop
  - » if X not executed on every iteration, then X must provably not cause an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



# Class Problem

---

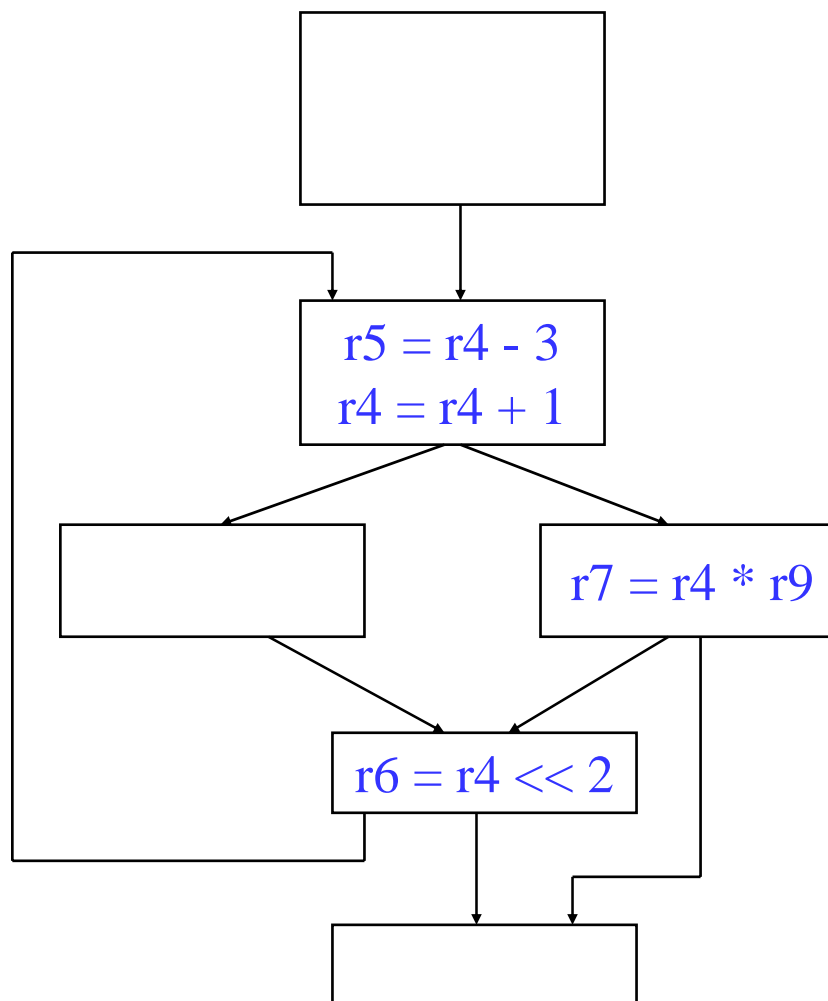
Apply global variable migration to the following program segment



# Induction Variable Strength Reduction

---

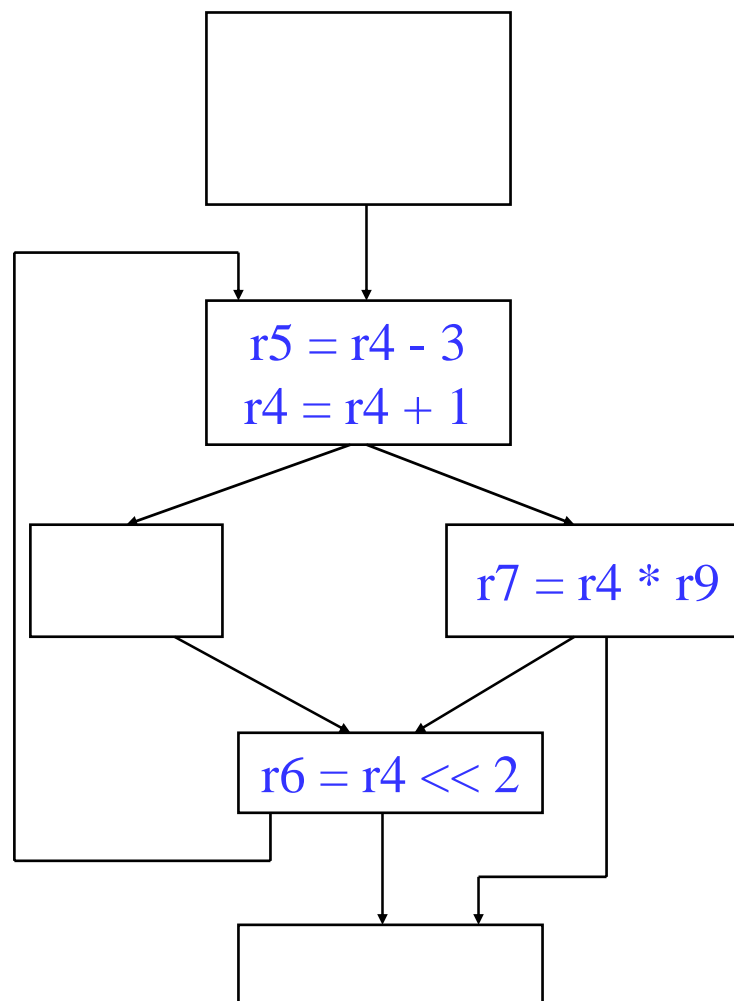
- ❖ Create basic induction variables from derived induction variables
- ❖ Rules
  - »  $X$  is a  $*$ ,  $\ll$ ,  $+$  or  $-$  operation
  - »  $\text{src1}(X)$  is a basic ind var
  - »  $\text{src2}(X)$  is invariant
  - » No other ops modify  $\text{dest}(X)$
  - »  $\text{dest}(X) \neq \text{src}(X)$  for all  $\text{srcs}$
  - »  $\text{dest}(X)$  is a register



# Induction Variable Strength Reduction (2)

## ❖ Transformation

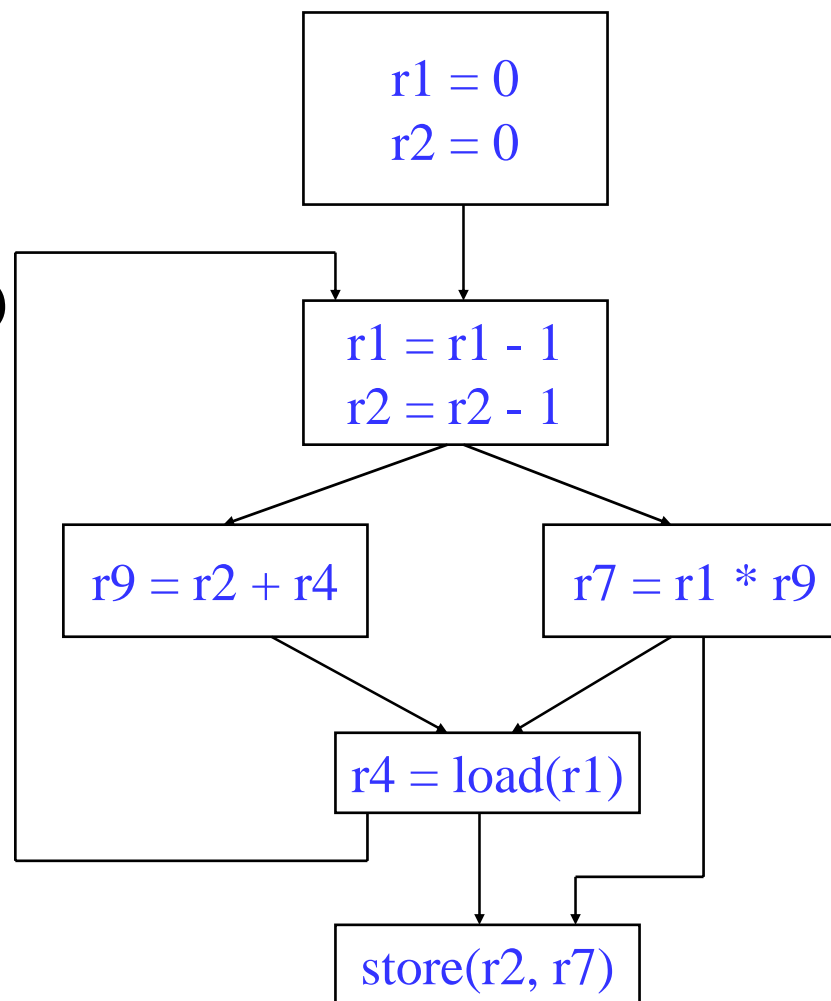
- » Insert the following into the bottom of preheader
  - $\text{new\_reg} = \text{RHS}(X)$
- » if opcode(X) is not add/sub, insert to the bottom of the preheader
  - $\text{new\_inc} = \text{inc}(\text{src1}(X)) \text{ opcode}(X) \text{ src2}(X)$
- » else
  - $\text{new\_inc} = \text{inc}(\text{src1}(X))$
- » Insert the following at each update of src1(X)
  - $\text{new\_reg} += \text{new\_inc}$
- » Change  $X \rightarrow \text{dest}(X) = \text{new\_reg}$



# 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)

---

## ❖ 5 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 (prev slide)
- » 3. Same increment, initial values are a known constant offset from one another
- » 4. Same inc, no nothing about relation of initial values
- » 5. Different increments, no nothing about initial values

## ❖ The higher the number, the more complex the elimination

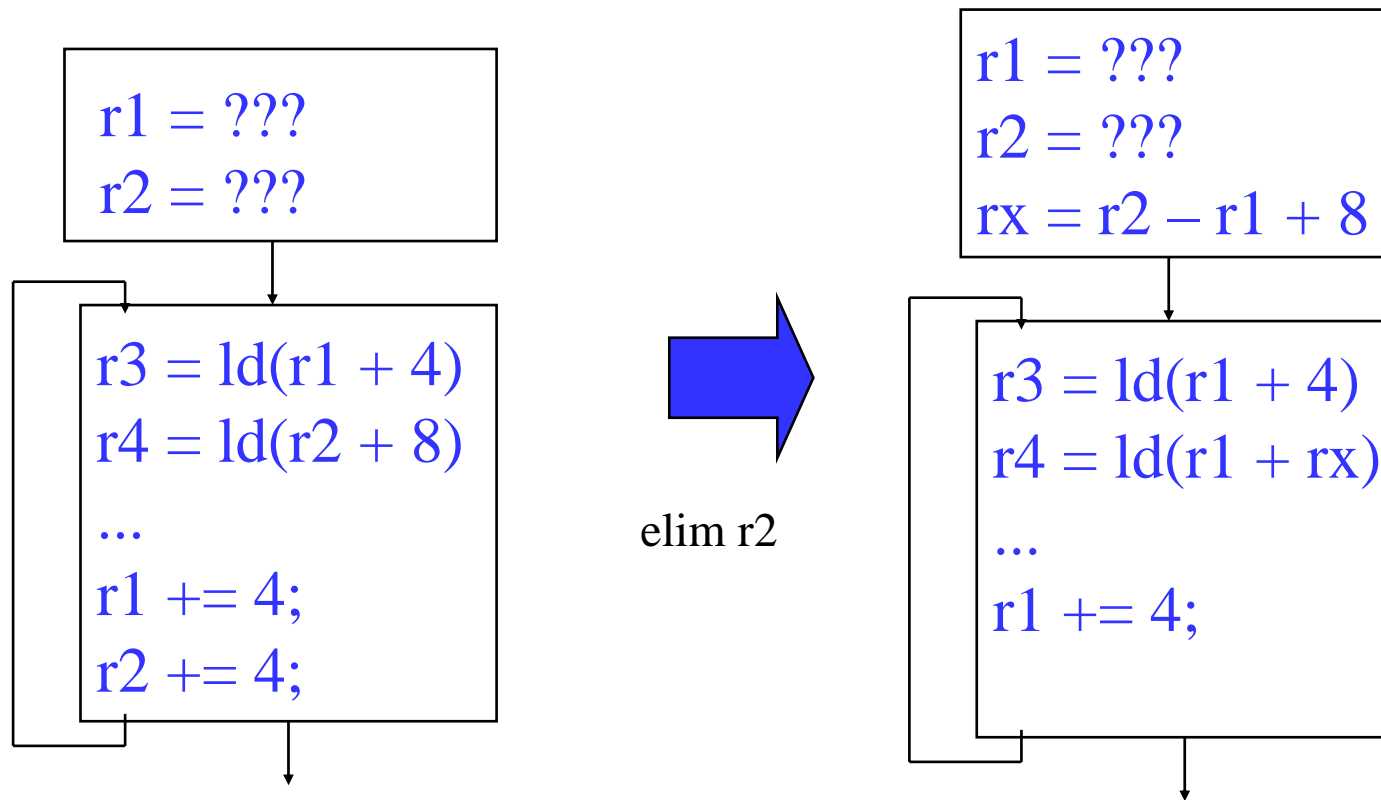
- » Also, the more expensive it is
- » 1,2 are basically free, so always should be done
- » 3-5 require preheader operations

# IVE Example

---

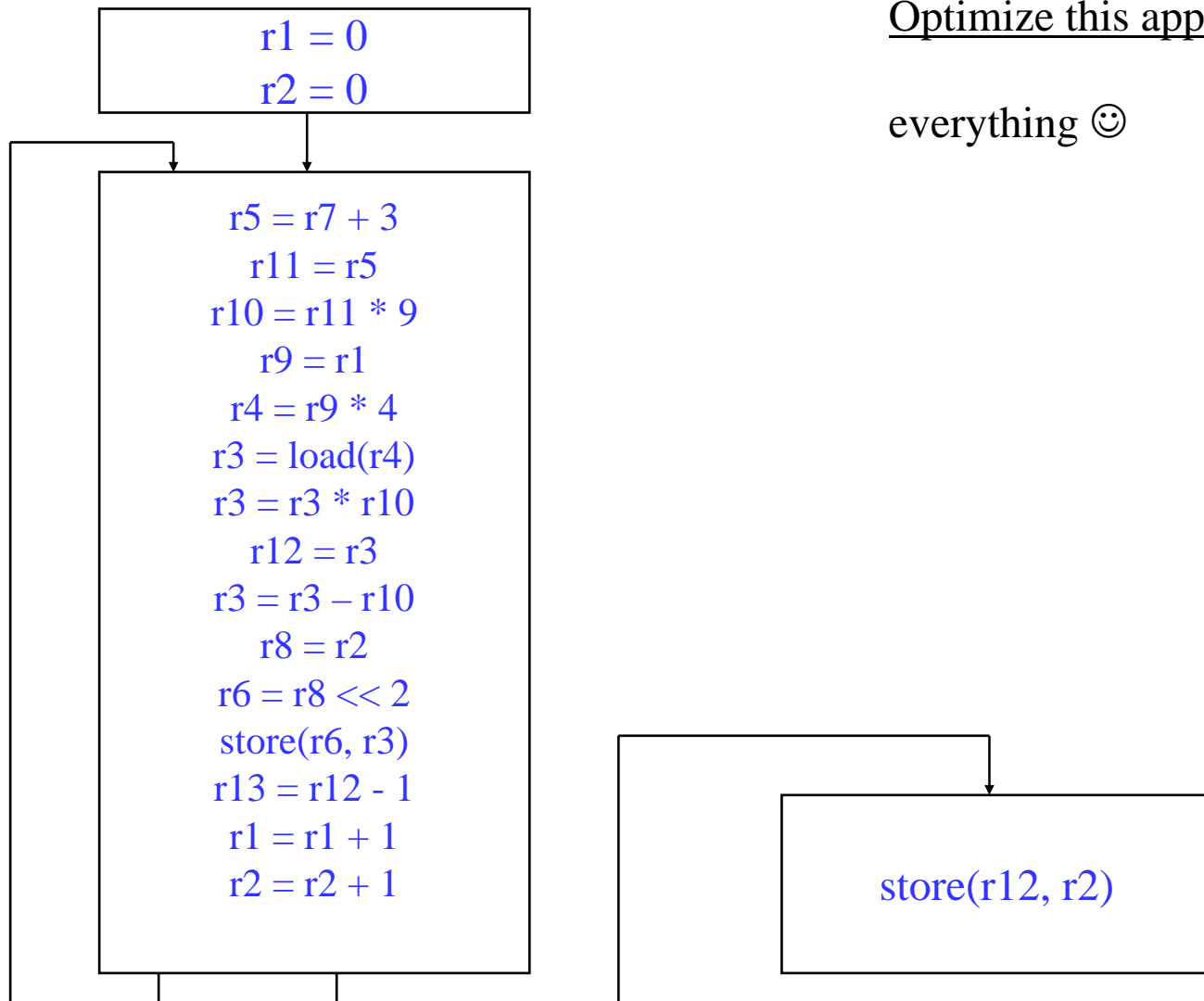
## Case 4: Same increment, unknown initial values

For the ind var you are eliminating, look at each non-increment use, need to regenerate the same sequence of values as before. If you can do that w/o adding any ops to the loop body, then apply xform



# Class Problem

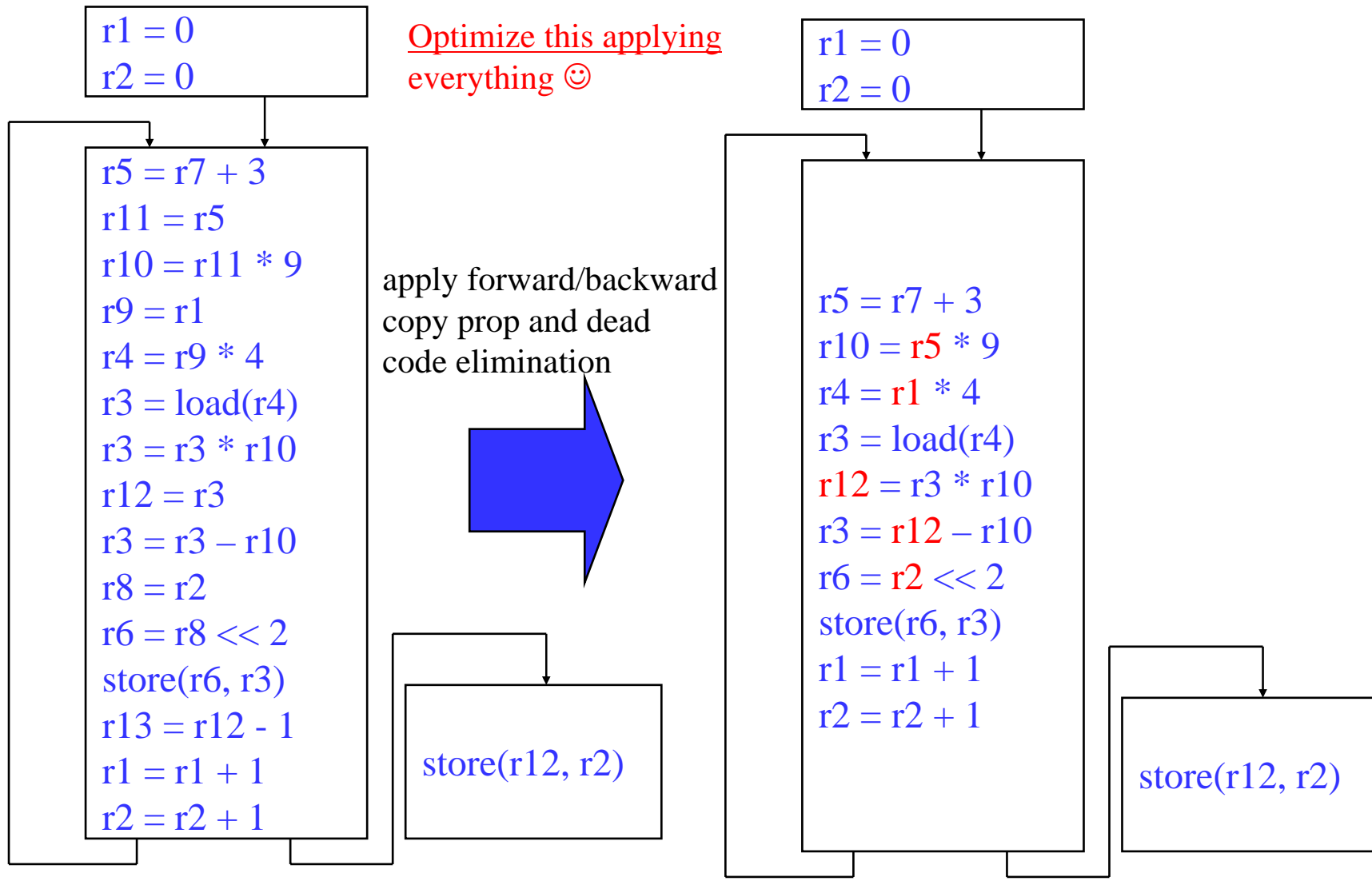
---



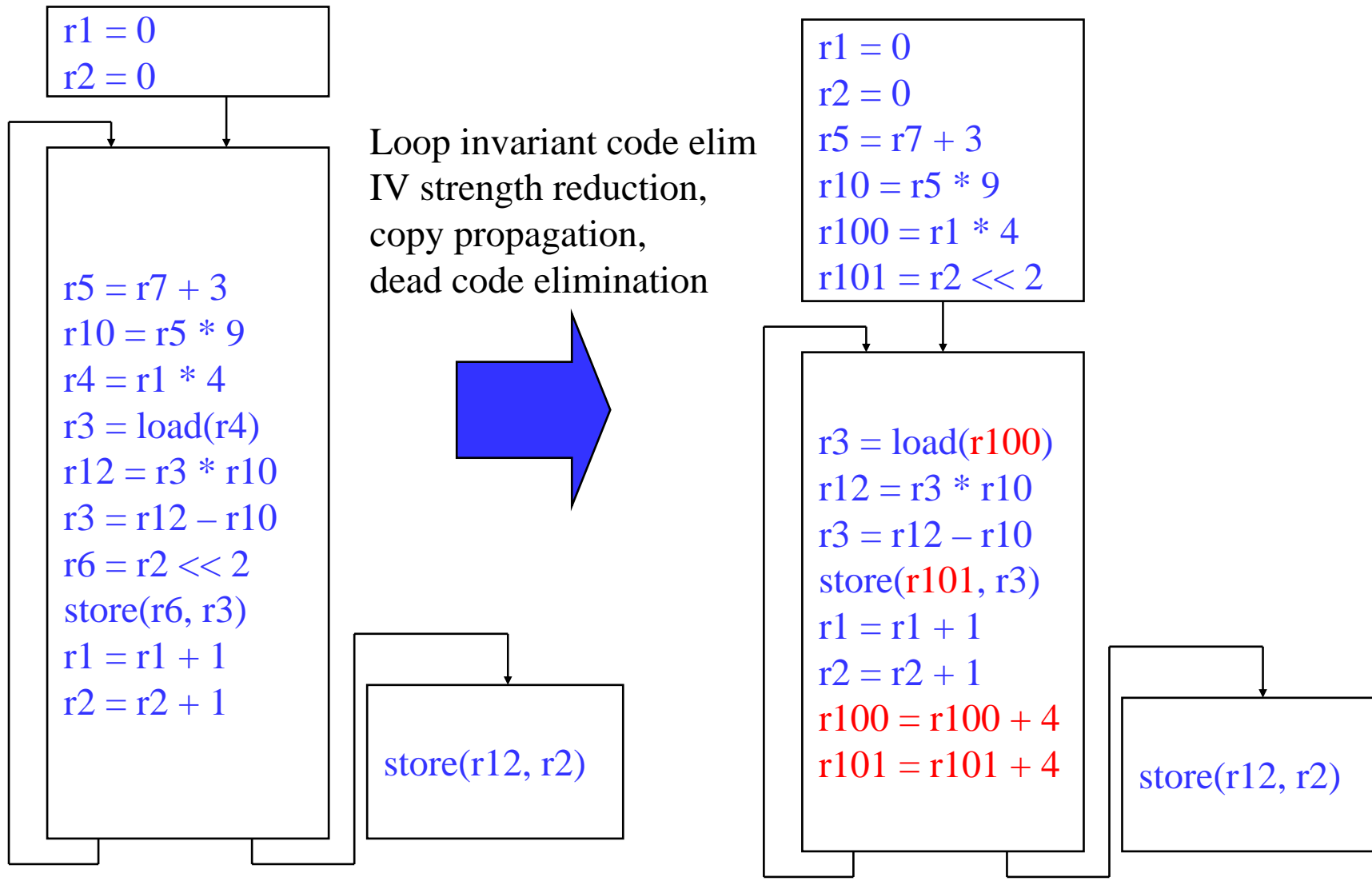
Optimize this applying

everything 😊

# Class Problem – Answer (1)

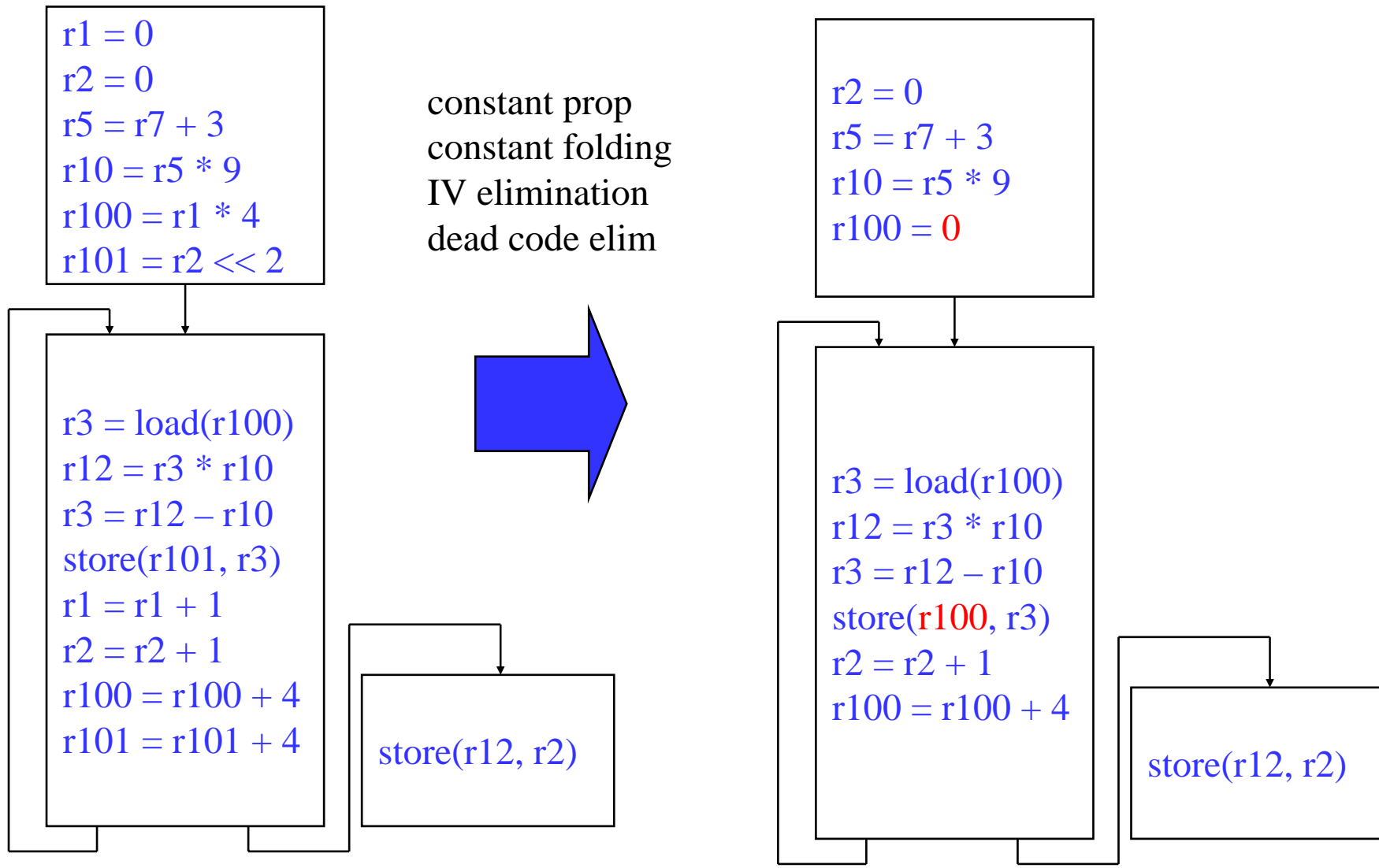


# Class Problem – Answer (2)



# Class Problem – Answer (3)

---



# Extra Material - Some Additional Optimizations

---

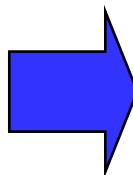
These will not be on the Exam

# 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
      r6 = r6 + r5
      r2 = r2 + 4
      r4 = r4 + 4
-----
      r1 = load(r2)
      r3 = load(r4)
      r5 = r1 * r3
      r6 = r6 + r5
      r2 = r2 + 4
      r4 = r4 + 4
-----
      r1 = load(r2)
      r3 = load(r4)
      r5 = r1 * r3
      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

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

# 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)
iter2  r15 = r11 * r13
      r16 = r16 + r15
      r2 = r2 + 4
      r4 = r4 + 4
-----
      r21 = load(r2)
      r23 = load(r4)
iter3  r25 = r21 * r23
      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

# 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)
iter2  r15 = r11 * r13
      r16 = r16 + r15
      r12 = r12 + 12
      r14 = r14 + 12
-----
      r21 = load(r22)
      r23 = load(r24)
iter3  r25 = r21 * r23
      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  $n * \text{original step}$