EECS 583 – Class 9
Classic Optimization

University of Michigan

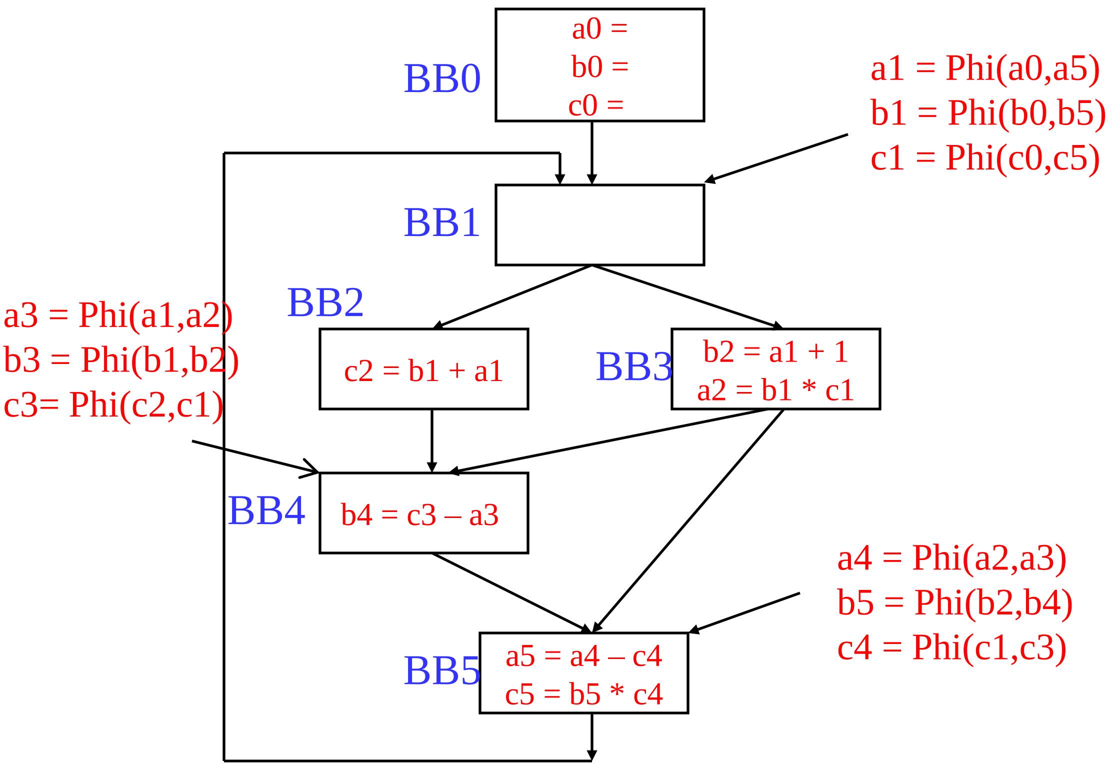
October 2, 2019

Announcements & Reading Material

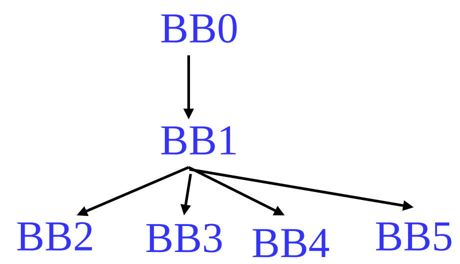
- HW2 – Get busy on it ASAP!
- Today's class
  - » *Compilers: Principles, Techniques, and Tools*, A. Aho, R. Sethi, and J. Ullman, Addison-Wesley, 1988, 9.9, 10.2, 10.3, 10.7 Edition 1; 8.5, 8.7, 9.1, 9.4, 9.5 Edition 2
- Material for Wednesday
  - » "Compiler Code Transformations for Superscalar-Based High-Performance Systems," S. Mahlke, W. Chen, J. Gyllenhaal, W. Hwu, P. Chang, and T. Kiyohara, *Proceedings of Supercomputing '92*, Nov. 1992, pp. 808-817
  - » And if you want more on ILP optimizations: D. J. Kuck, The Structure of Computers and Computations. New York, NY: John Wiley and Sons, 1978. (optional!)

Class Problem From Last Time – Answer

Rename the variables



Dominator tree



Dominance frontier

BB	DF
0	-
1	-
2	4
3	4, 5
4	5
5	1

Code Optimization

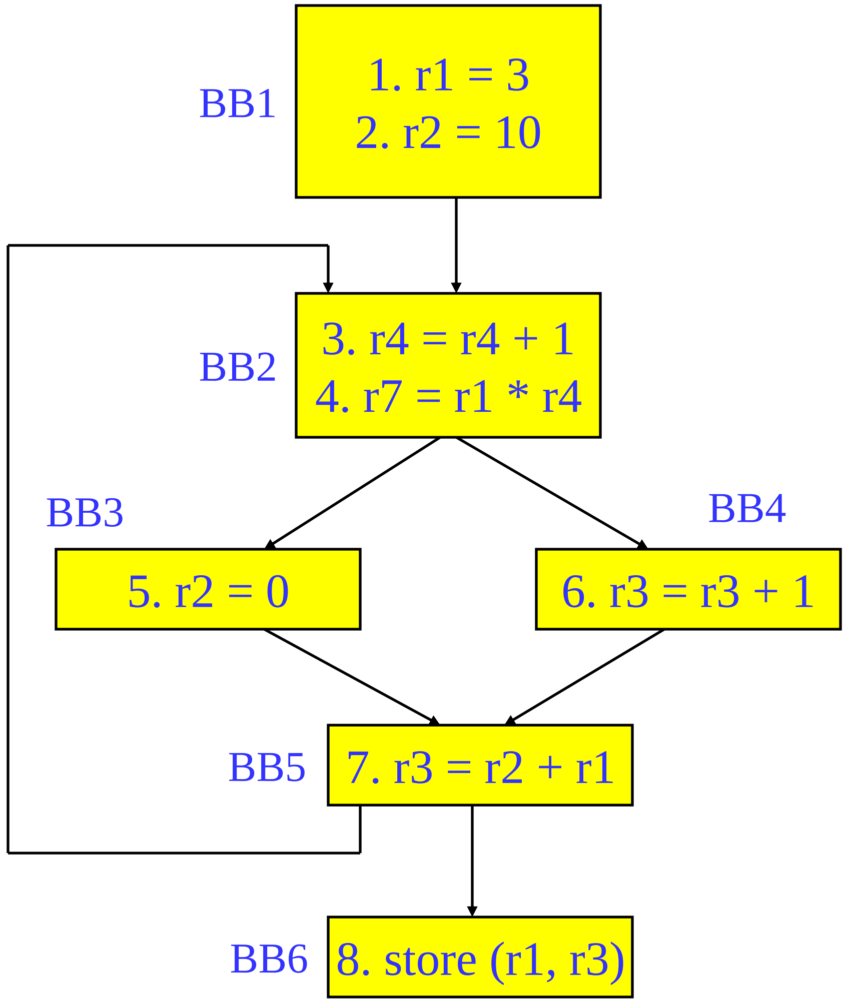
- Make the code run faster on the target processor
  - My (Scott's) favorite topic !!
  - Other objectives: Power, code size
- Classes of optimization
  - 1. Classical (machine independent)
    - Reducing operation count (redundancy elimination)
    - Simplifying operations
    - Generally good for any kind of machine
  - 2. Machine specific
    - Peephole optimizations
    - Take advantage of specialized hardware features
  - 3. Parallelism enhancing
    - Increasing parallelism (ILP or TLP)
    - Possibly increase instructions

A Tour Through the Classical Optimizations

- For this class – Go over concepts of a small subset of the optimizations
  - What it is, why its useful
  - When can it be applied (set of conditions that must be satisfied)
  - How it works
  - Give you the flavor but don't want to beat you over the head
- Challenges
  - Register pressure?
  - Parallelism verses operation count

Dead Code Elimination

- Remove any operation who's result is never consumed
- Rules
  - X can be deleted
    - no stores or branches
  - DU chain empty or dest register not live
- This misses some dead code!!
  - Especially in loops
- Better Algorithm
  - 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



Local Constant Propagation

- Forward propagation of moves of the form
  - rx = L (where L is a literal)
  - Maximally propagate
- Consider 2 ops, X and Y in a BB, X is before Y
  - 1. X is a move
  - 2. src1(X) is a literal
  - 3. Y consumes dest(X)
  - 4. There is no definition of dest(X) between X and Y

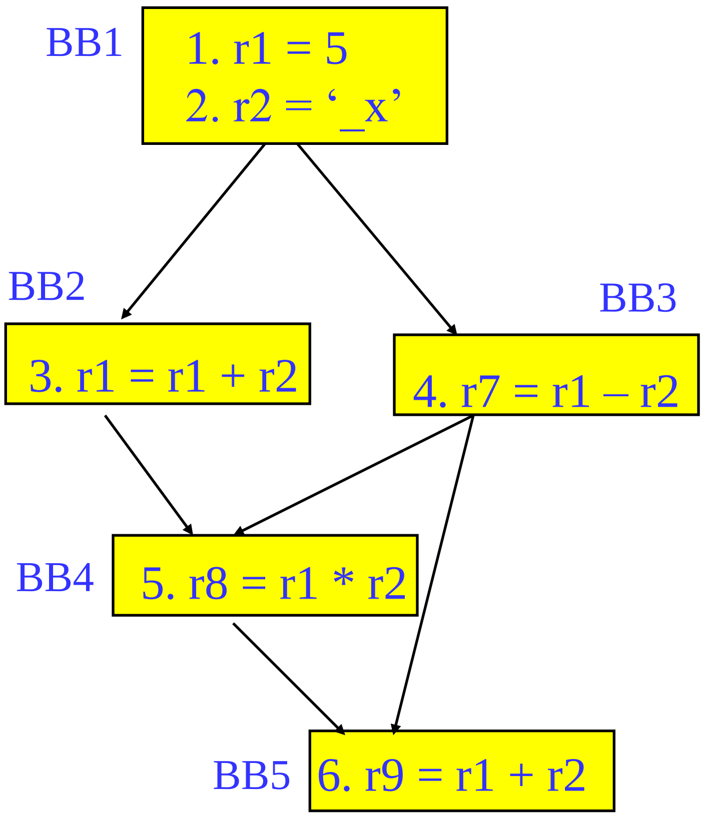
BB1

1. r1 = 5
2. r2 = '_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

Note, ignore operation format issues, so all operations can have literals in either operand position

Global Constant Propagation

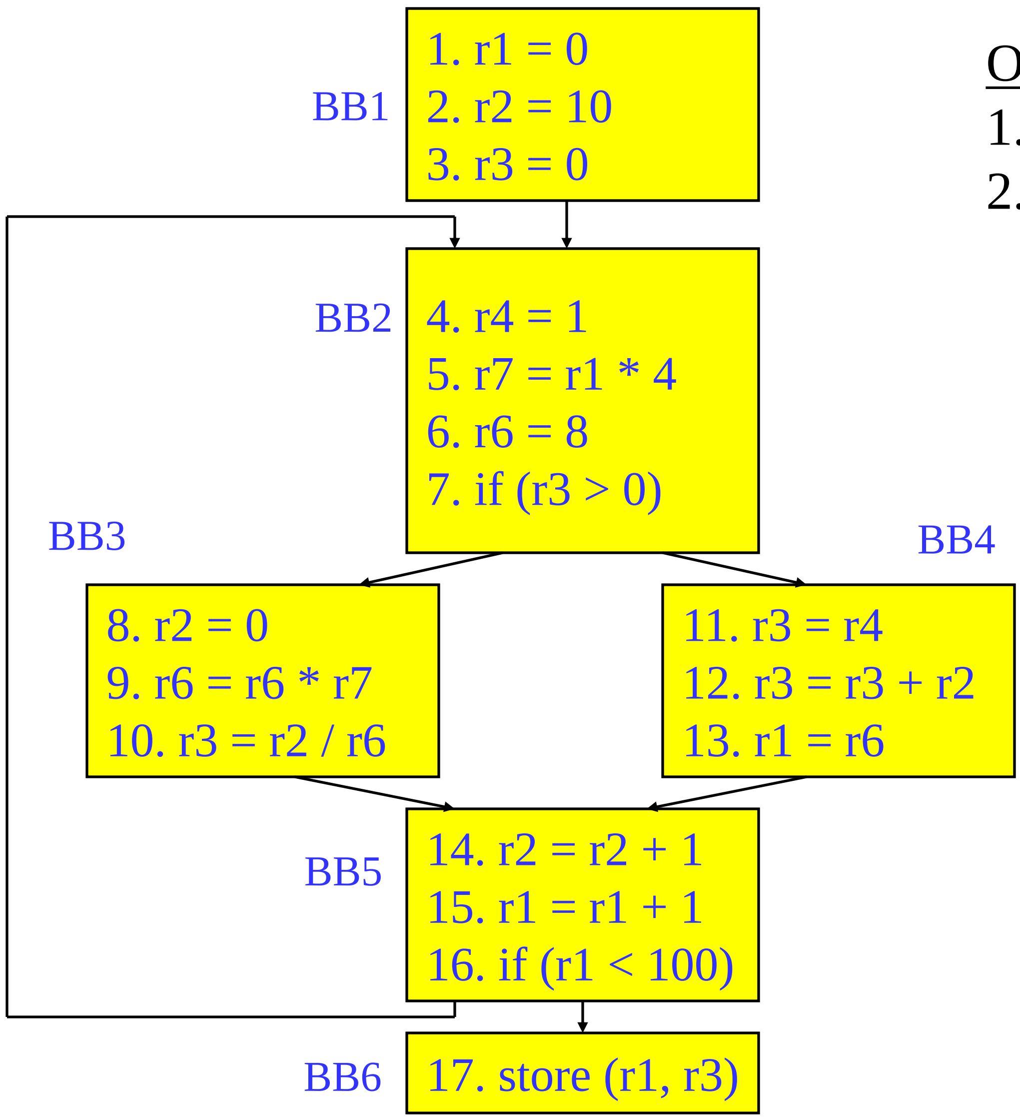
- Consider 2 ops, X and Y in different BBs
  - » 1. X is a move
  - » 2. src1(X) is a literal
  - » 3. Y consumes dest(X)
  - » 4. X is in a_in(BB(Y))
  - » 5. Dest(x) is not modified between the top of BB(Y) and Y



Constant Folding

- Simplify 1 operation based on values of src operands
  - Constant propagation creates opportunities for this
- All constant operands
  - Evaluate the op, replace with a move
    - r1 = 3 * 4 → r1 = 12
    - r1 = 3 / 0 → ??? Don't evaluate excepting ops !, what about floating-point?
  - Evaluate conditional branch, replace with BRU or noop
    - if (1 < 2) goto BB2 → BRU BB2
    - if (1 > 2) goto BB2 → convert to a noop
- Algebraic identities
  - r1 = r2 + 0, r2 – 0, r2 | 0, r2 ^ 0, r2 << 0, r2 >> 0
    - r1 = r2
  - r1 = 0 * r2, 0 / r2, 0 & r2
    - r1 = 0
  - r1 = r2 * 1, r2 / 1
    - r1 = r2

Class Problem

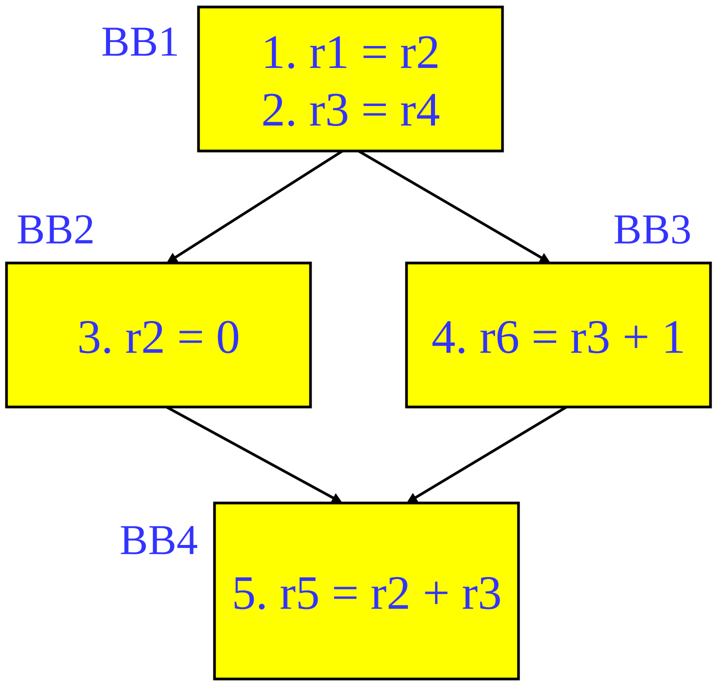


Optimize this applying

1. constant propagation
2. constant folding

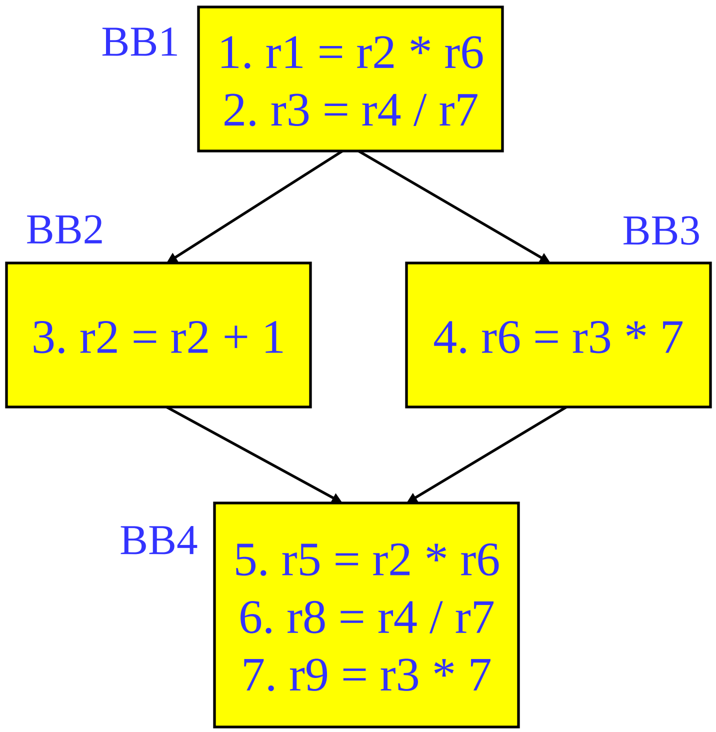
Forward Copy Propagation

- Forward propagation of the RHS of moves
  - r1 = r2
  - …
  - r4 = r1 + 1 → r4 = r2 + 1
- Benefits
  - Reduce chain of dependences
  - Eliminate the move
- Rules (ops X and Y)
  - X is a move
  - src1(X) is a register
  - Y consumes dest(X)
  - X.dest is an available def at Y
  - X.src1 is an available expr at Y



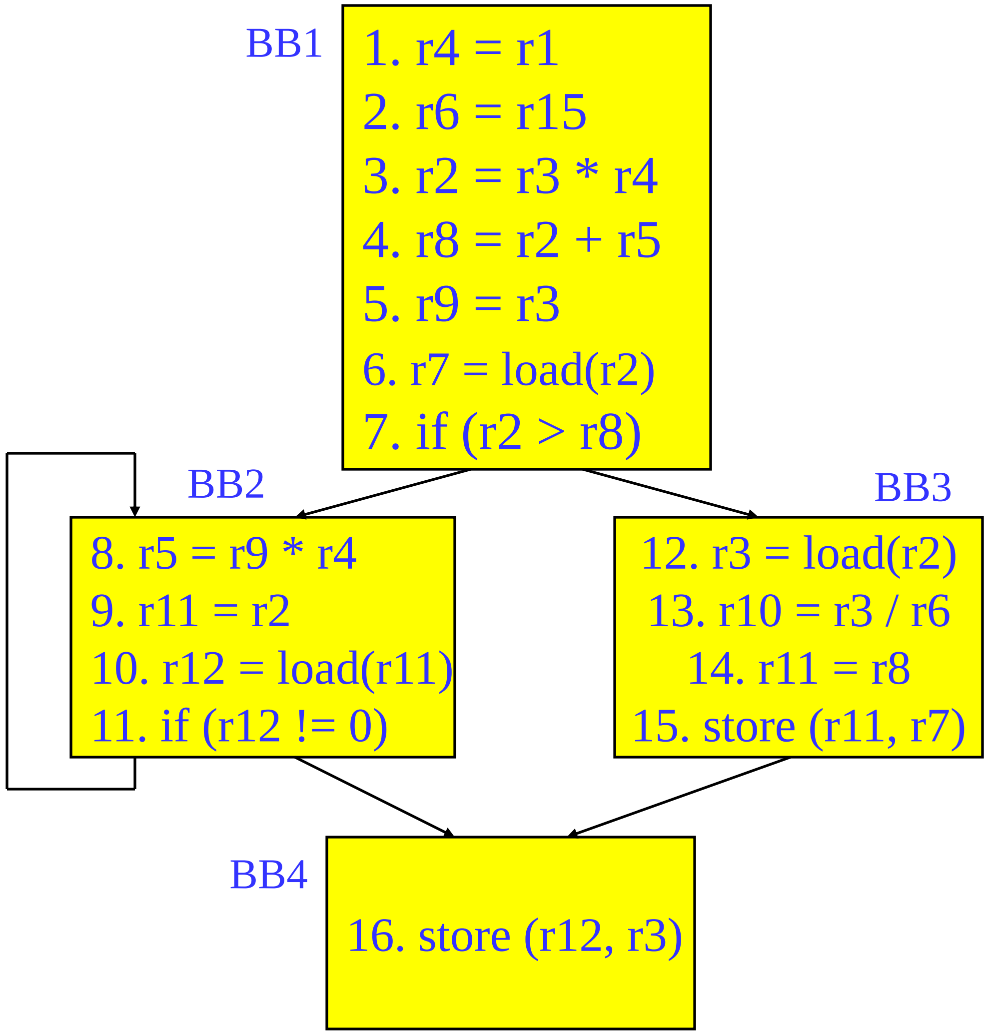
CSE – Common Subexpression Elimination

- Eliminate recomputation of an expression by reusing the previous result
  - r1 = r2 * r3
  - → r100 = r1
  - …
  - r4 = r2 * r3 → r4 = r100
- Benefits
  - Reduce work
  - Moves can get copy propagated
- Rules (ops X and Y)
  - X and Y have the same opcode
  - src(X) = src(Y), for all srcs
  - expr(X) is available at Y
  - if X is a load, then there is no store that may write to address(X) along any path between X and Y



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

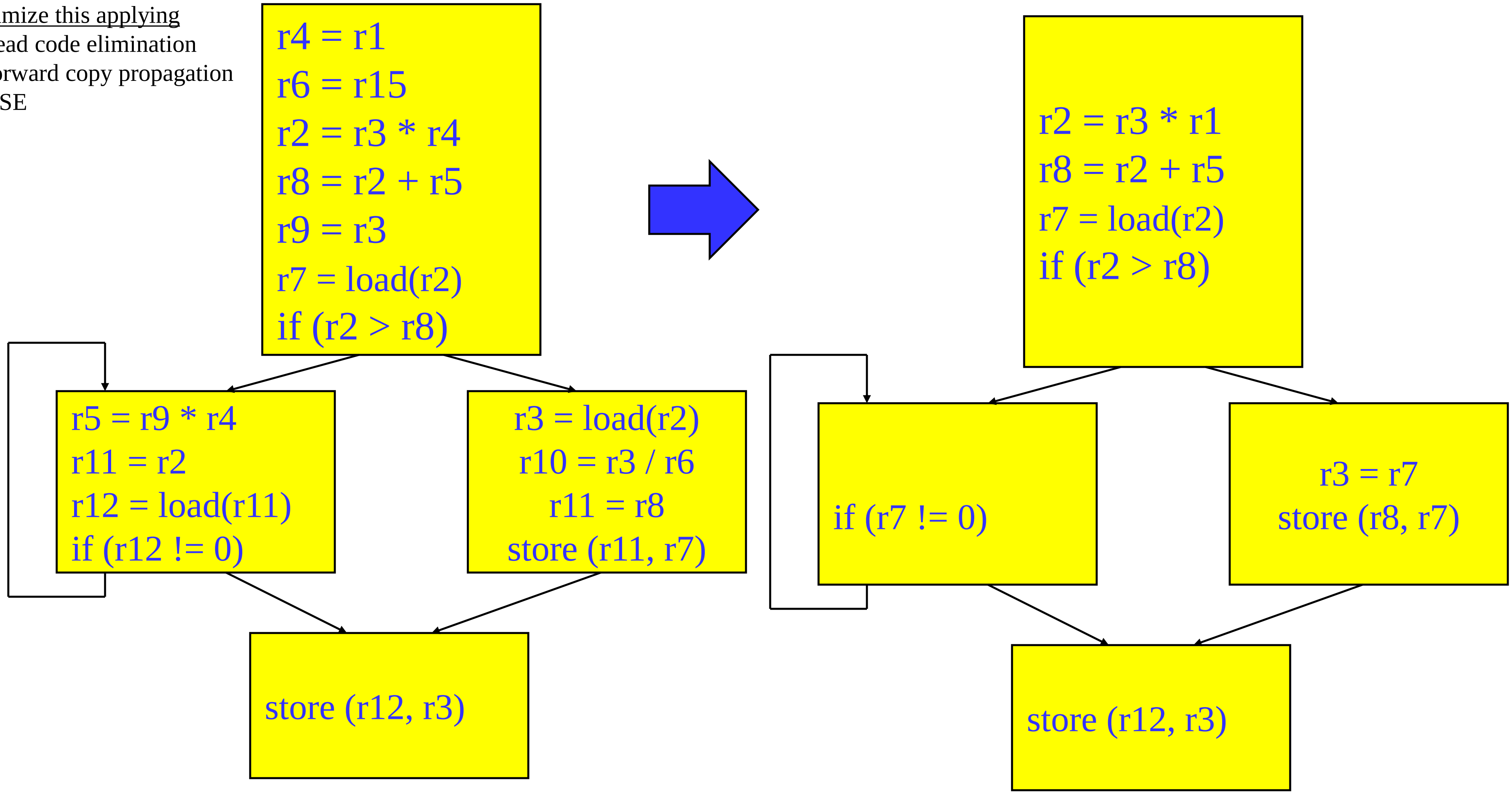
Class Problem



Optimize this applying
1. dead code elimination
2. forward copy propagation
3. CSE

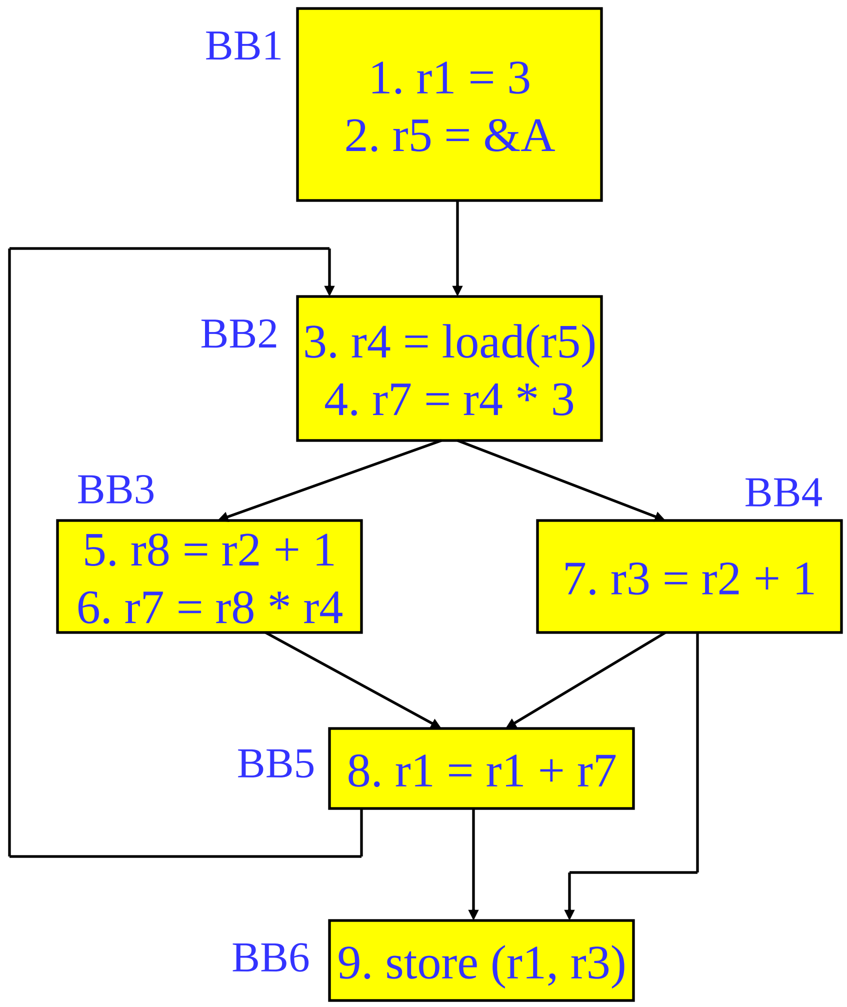
Class Problem Solution

Optimize this applying
1. dead code elimination
2. forward copy propagation
3. CSE



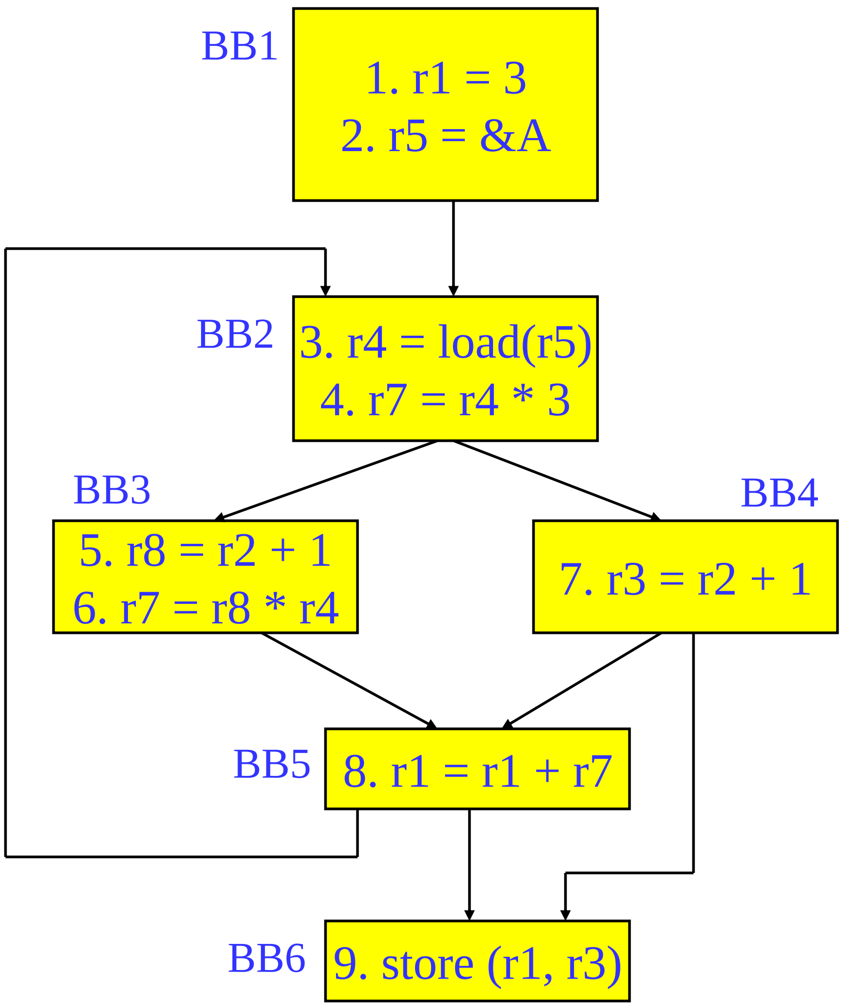
Loop Invariant Code Motion (LICM)

- 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



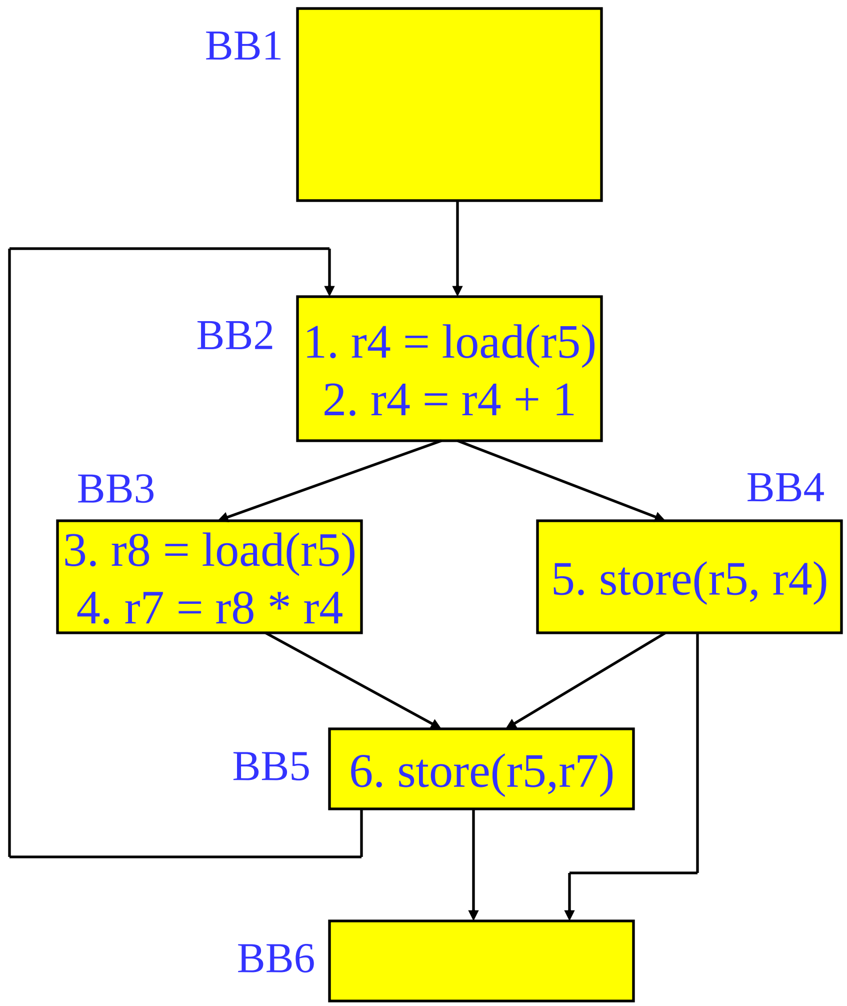
LICM (2)

- 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



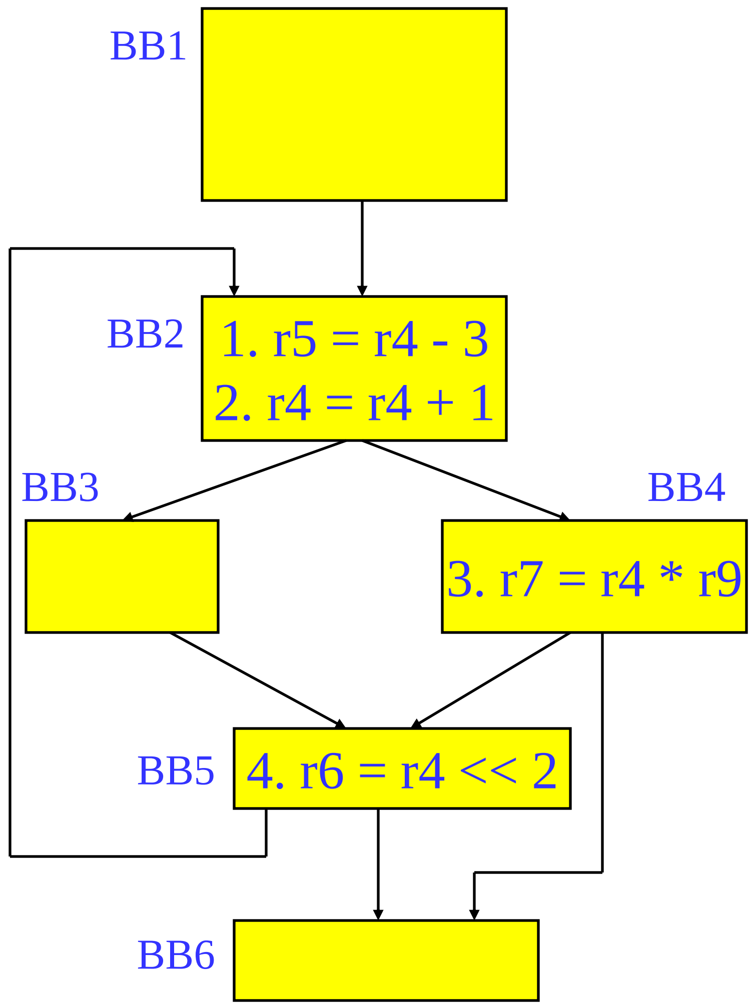
Global Variable Migration

- 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



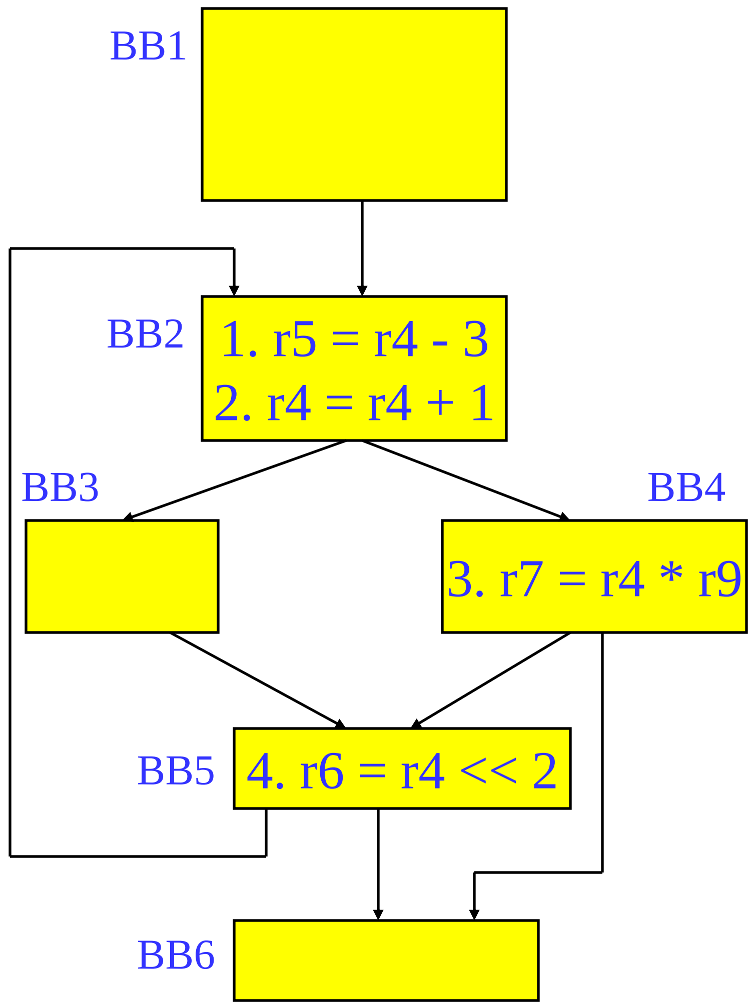
Induction Variable Strength Reduction

- Create basic induction variables from derived induction variables
- Induction variable
  - » BIV (i++)
    - 0,1,2,3,4,...
  - » DIV (j = i * 4)
    - 0, 4, 8, 12, 16, ...
  - » DIV can be converted into a BIV that is incremented by 4
- Issues
  - » Initial and increment vals
  - » Where to place increments

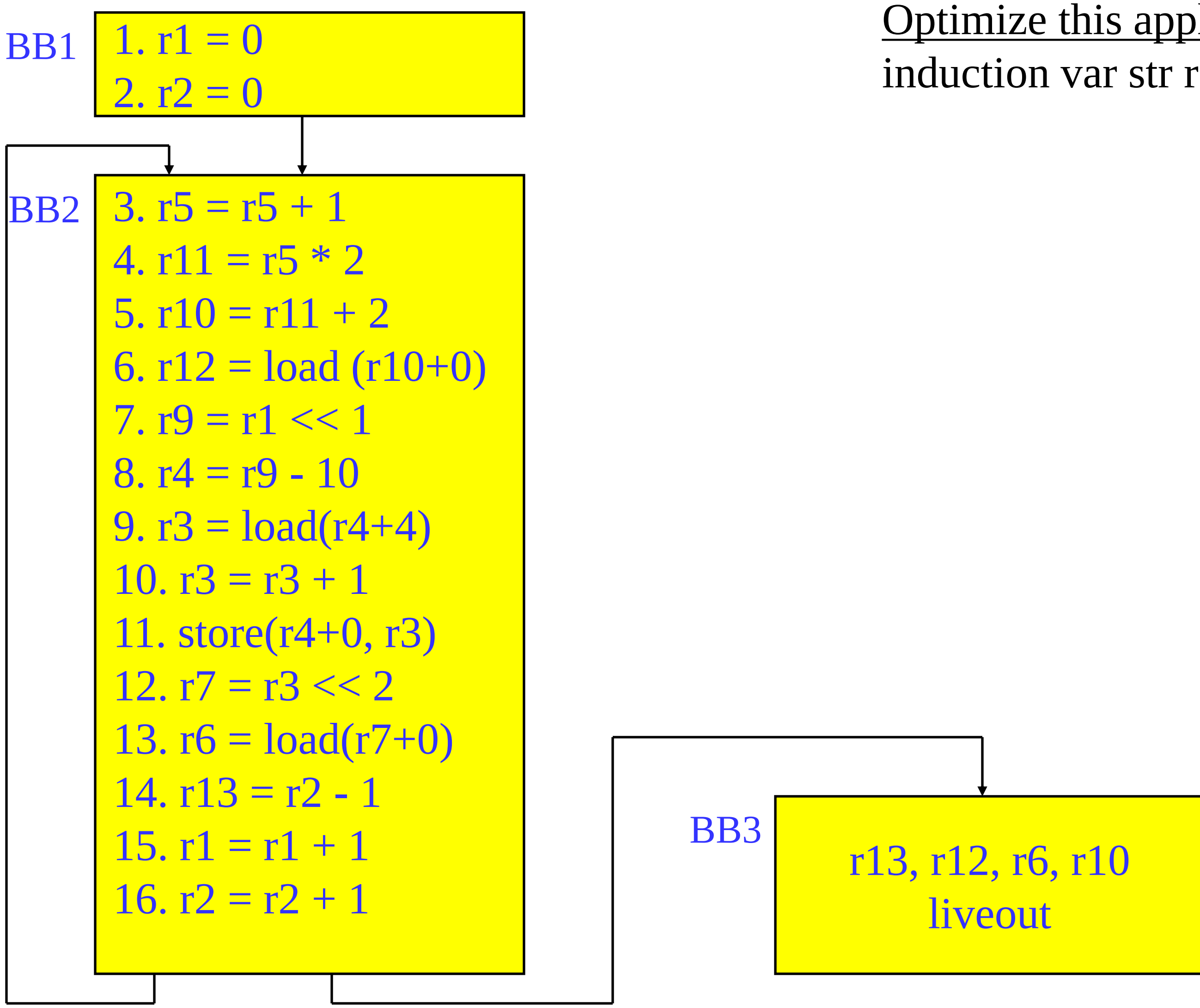


Induction Variable Strength Reduction (2)

- 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
- Transformation
  - Insert the following into the preheader
    - new_reg = RHS(X)
  - If opcode(X) is not add/sub, insert to the bottom of the preheader
    - new_inc = inc(src1(X)) opcode(X) src2(X)
  - else
    - new_inc = inc(src1(X))
  - Insert the following at each update of src1(X)
    - new_reg += new_inc
  - Change X → dest(X) = new_reg



Class Problem



Optimize this applying induction var str reduction

Class Problem Solution

Optimize this applying induction var str reduction

Original:

```
r1 = 0
r2 = 0
```

```
r5 = r5 + 1
r11 = r5 * 2
r10 = r11 + 2
r12 = load (r10+0)
r9 = r1 << 1
r4 = r9 - 10
r3 = load(r4+4)
r3 = r3 + 1
store(r4+0, r3)
r7 = r3 << 2
r6 = load(r7+0)
r13 = r2 - 1
r1 = r1 + 1
r2 = r2 + 1
```

```
r13, r12, r6, r10
liveout
```

Optimized:

```
r1 = 0
r2 = 0
r111 = r5 * 2
r109 = r1 << 1
r113 = r2 -1
```

```
r5 = r5 + 1
r111 = r111 + 2
r11 = r111
r10 = r11 + 2
r12 = load (r10+0)
r9 = r109
r4 = r9 - 10
r3 = load(r4+4)
r3 = r3 + 1
store(r4+0, r3)
r7 = r3 << 2
r6 = load(r7+0)
r13 = r113
r1 = r1 + 1
r109 = r109 + 2
r2 = r2 + 1
r113 = r113 + 1
```

```
r13, r12, r6, r10
liveout
```

Note, after copy propagation, r10 and r4 can be strength reduced as well.