EECS 583 – Class 8
Classic Optimization

University of Michigan

September 22, 2025

Announcements & Reading Material

❖ HW2 – Get busy on it ASAP!

❖ Talk about course projects next class – Start thinking about forming/joining a group (3-5 students)

❖ 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 next Monday

   » "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!)

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, done at IR level)

- Reducing operation count (redundancy elimination)
- Simplifying operations
- Generally good for any kind of machine

» 2. Machine specific (done in llc)

- Peephole optimizations
- Take advantage of specialized hardware features

» 3. Parallelism enhancing (IR level often, but sometimes llc)

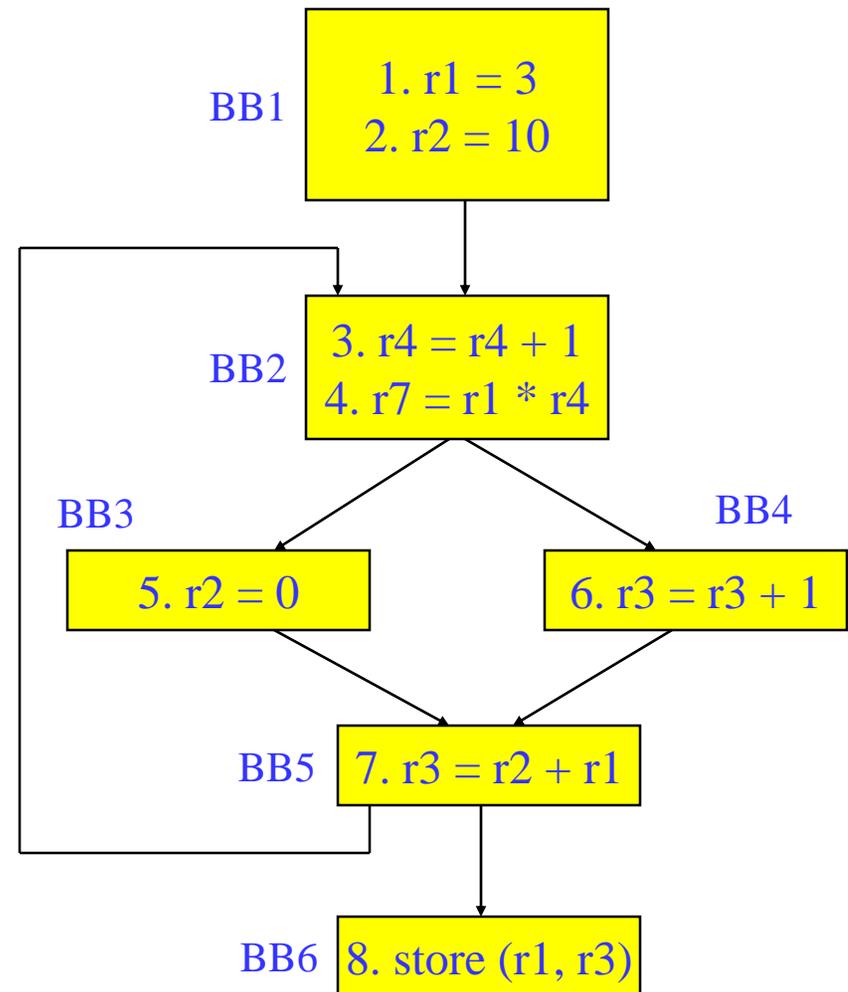- Increasing parallelism (ILP or TLP)
- Possibly increase instructions

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

BB1
```
1. r1 = 3
2. r2 = 10
```

BB2
```
3. r4 = r4 + 1
4. r7 = r1 * r4
```

BB3
```
5. r2 = 0
```

BB4
```
6. r3 = r3 + 1
```

BB5
```
7. r3 = r2 + r1
```

BB6
```
8. store (r1, r3)
```

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 (local available defs)

BB1

1. r1 = 5
2. r2 = 3
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)) (global available defs)
  - » 5. Dest(x) is not modified between the top of BB(Y) and Y (local available defs)

BB1
$$1.\ r1 = 5$$
$$2.\ r2 = 10$$

BB2
$$3.\ r1 = r1 + r2$$

BB3
$$4.\ r7 = r1 - r2$$

BB4
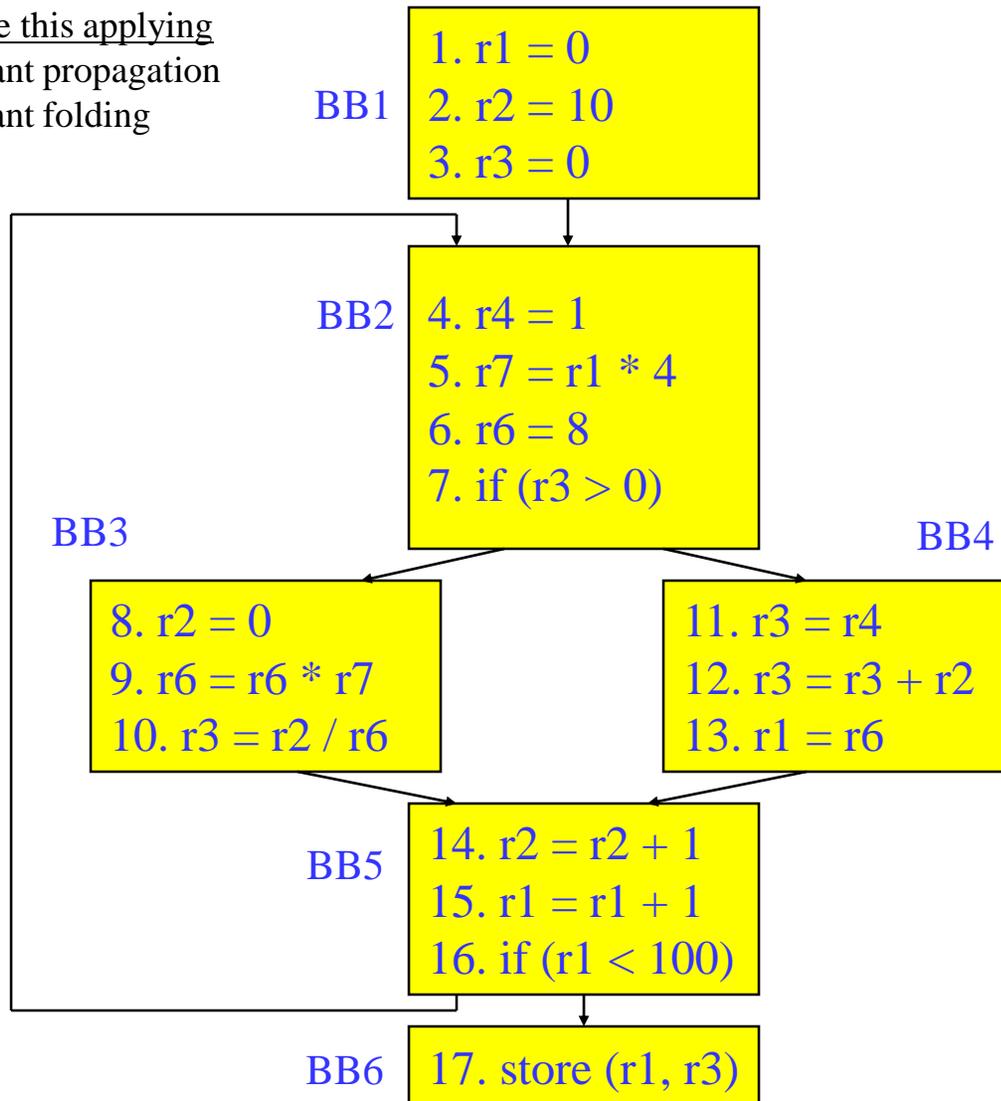$$5.\ r8 = r1 * r2$$

BB5
$$6.\ r9 = r1 + r2$$
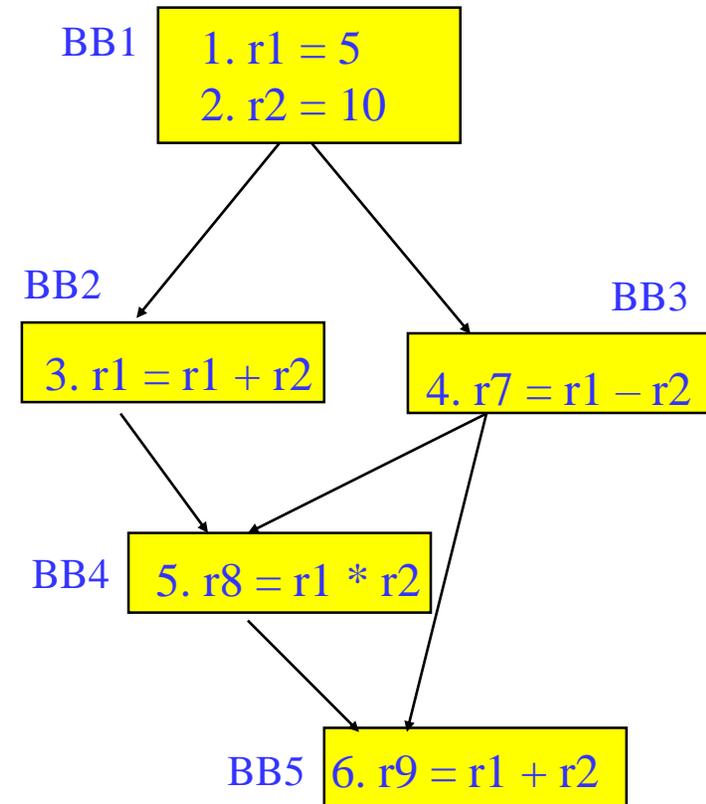
Constant Folding

❖ 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

BB1
1. r1 = 0
2. r2 = 10
3. r3 = 0

BB2
4. r4 = 1
5. r7 = r1 * 4
6. r6 = 8
7. if (r3 > 0)

BB3
8. r2 = 0
9. r6 = r6 * r7
10. r3 = r2 / r6

BB4
11. r3 = r4
12. r3 = r3 + r2
13. r1 = r6

BB5
14. r2 = r2 + 1
15. r1 = r1 + 1
16. if (r1 < 100)

BB6
17. store (r1, r3)

Class Problem - Solution

Optimize this applying
1. constant propagation
2. constant folding

BB1
```
1. r1 = 0
2. r2 = 10
3. r3 = 0
```

BB2
```
4. r4 = 1
5. r7 = r1 * 4
6. r6 = 8
7. if (r3 > 0)
```

BB3
```
8. r2 = 0
9. r6 = r6 * r7
10. r3 = r2 / r6
```

BB4
```
11. r3 = r4
12. r3 = r3 + r2
13. r1 = r6
```

BB5
```
14. r2 = r2 + 1
15. r1 = r1 + 1
16. if (r1 < 100)
```

BB6
```
17. store (r1, r3)
```

BB1
```
1. r1 = 0
2. r2 = 10
3. r3 = 0
```

BB2
```
4. r4 = 1
5. r7 = r1 * 4
6. r6 = 8
7. if (r3 > 0)
```

BB3
```
8. r2 = 0
9. r6 = r7 << 3
10. r3 = 0
```

BB4
```
11. r3 = 1
12. r3 = 1 + r2
13. r1 = 8
```

BB5
```
14. r2 = r2 + 1
15. r1 = r1 + 1
16. if (r1 < 100)
```

BB6
```
17. store (r1, r3)
```

Red instructions can be deleted by
subsequent dead code elimination
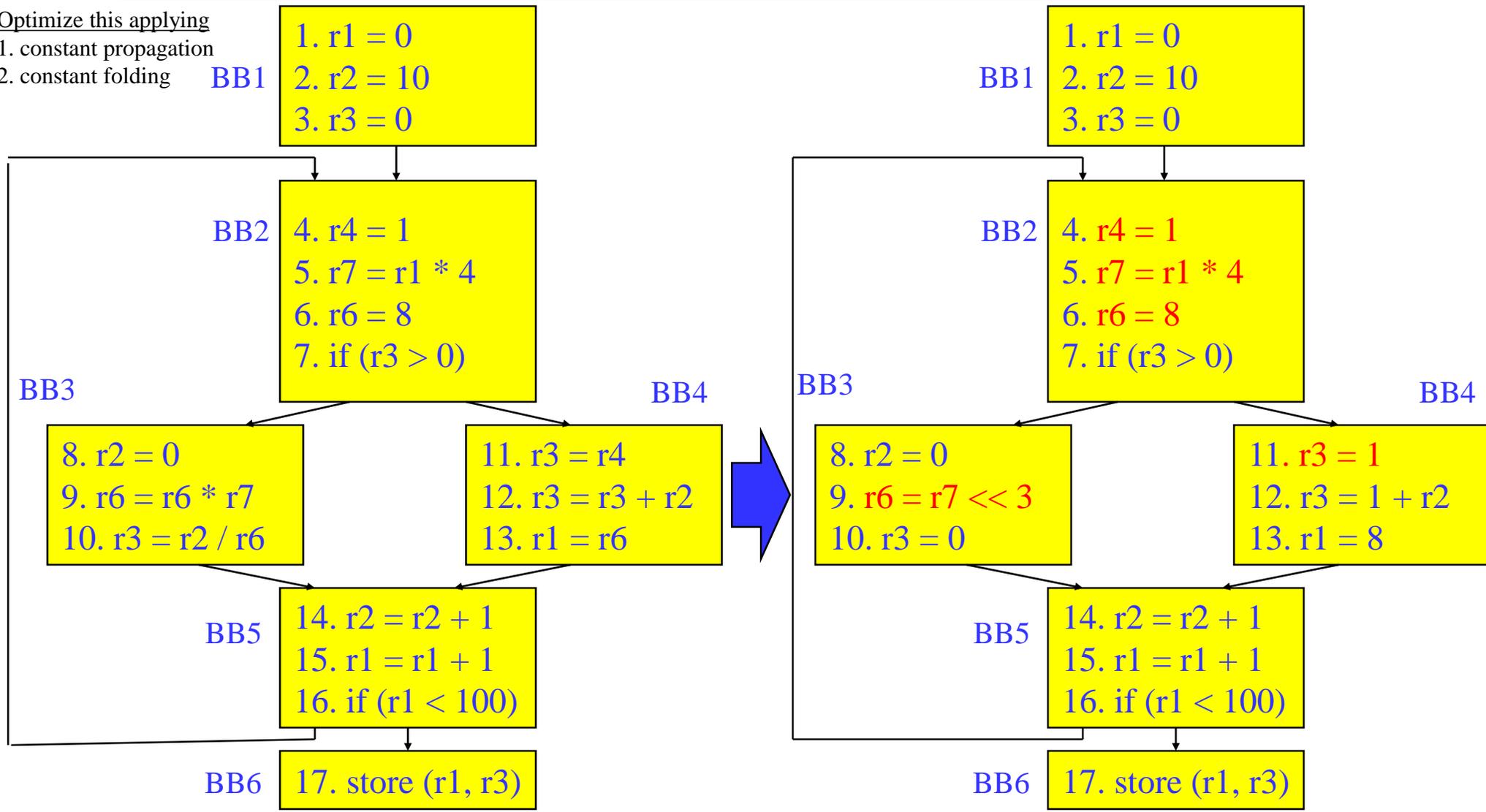
Forward Copy Propagation

❖ Forward propagation of the RHS of moves

   » r1 = r2

   » …

   » r4 = r1 + 1 ➔ r4 = r2 + 1

❖ Benefits

   » Reduce chain of dependences
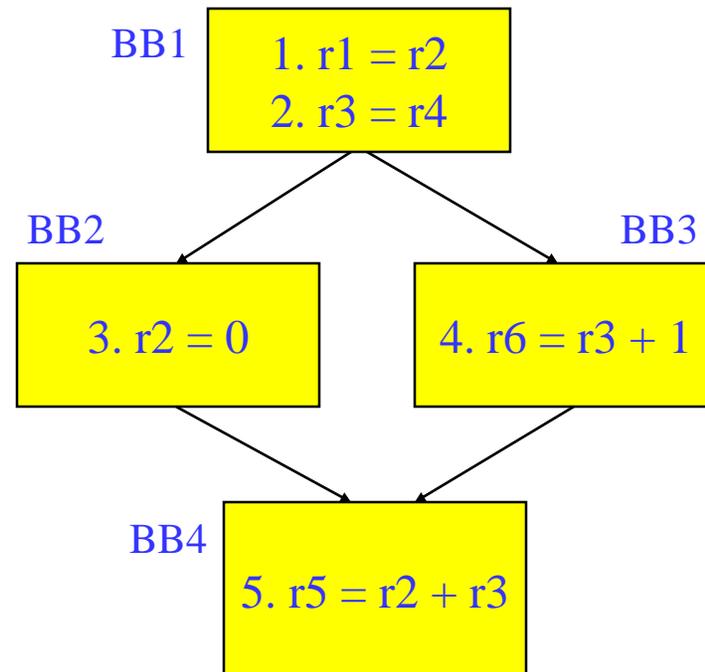
   » Eliminate the move

❖ 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

BB1
```
1. r1 = r2
2. r3 = r4
```

BB2
```
3. r2 = 0
```

BB3
```
4. r6 = r3 + 1
```

BB4
```
5. r5 = r2 + r3
```

CSE – Common Subexpression Elimination

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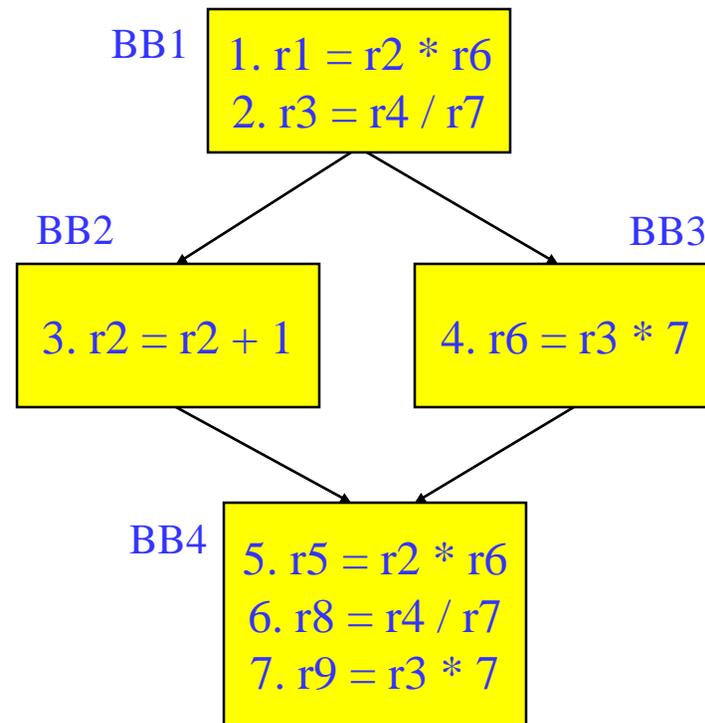 - » → r100 = r1
  - » …
  - » r4 = r2 * r3 → r4 = r100
- ❖ Benefits
  - » Reduce work
  - » Moves can get copy propagated
- ❖ Rules (ops X and Y)
  - » X and Y have the same opcode
  - » src(X) = src(Y), for all srcs
  - » expr(X) is available at Y
  - » if X is a load, then there is no store that may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

BB1
$$1.\ r1 = r2 * r6$$
$$2.\ r3 = r4 / r7$$

BB2
$$3.\ r2 = r2 + 1$$

BB3
$$4.\ r6 = r3 * 7$$

BB4
$$5.\ r5 = r2 * r6$$
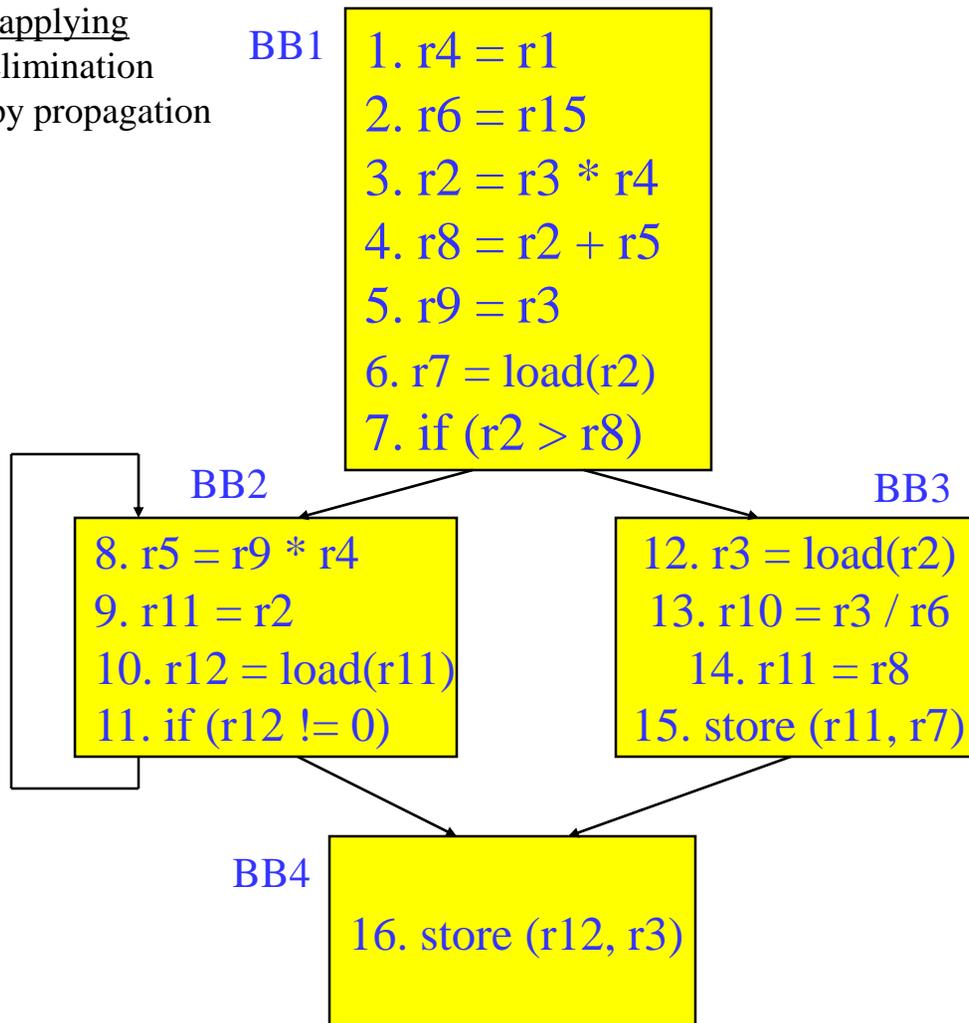$$6.\ r8 = r4 / r7$$
$$7.\ r9 = r3 * 7$$

if op is a load, call it redundant load elimination rather than CSE, Redundant store elimination also possible, but applied a bit differently – think about it!
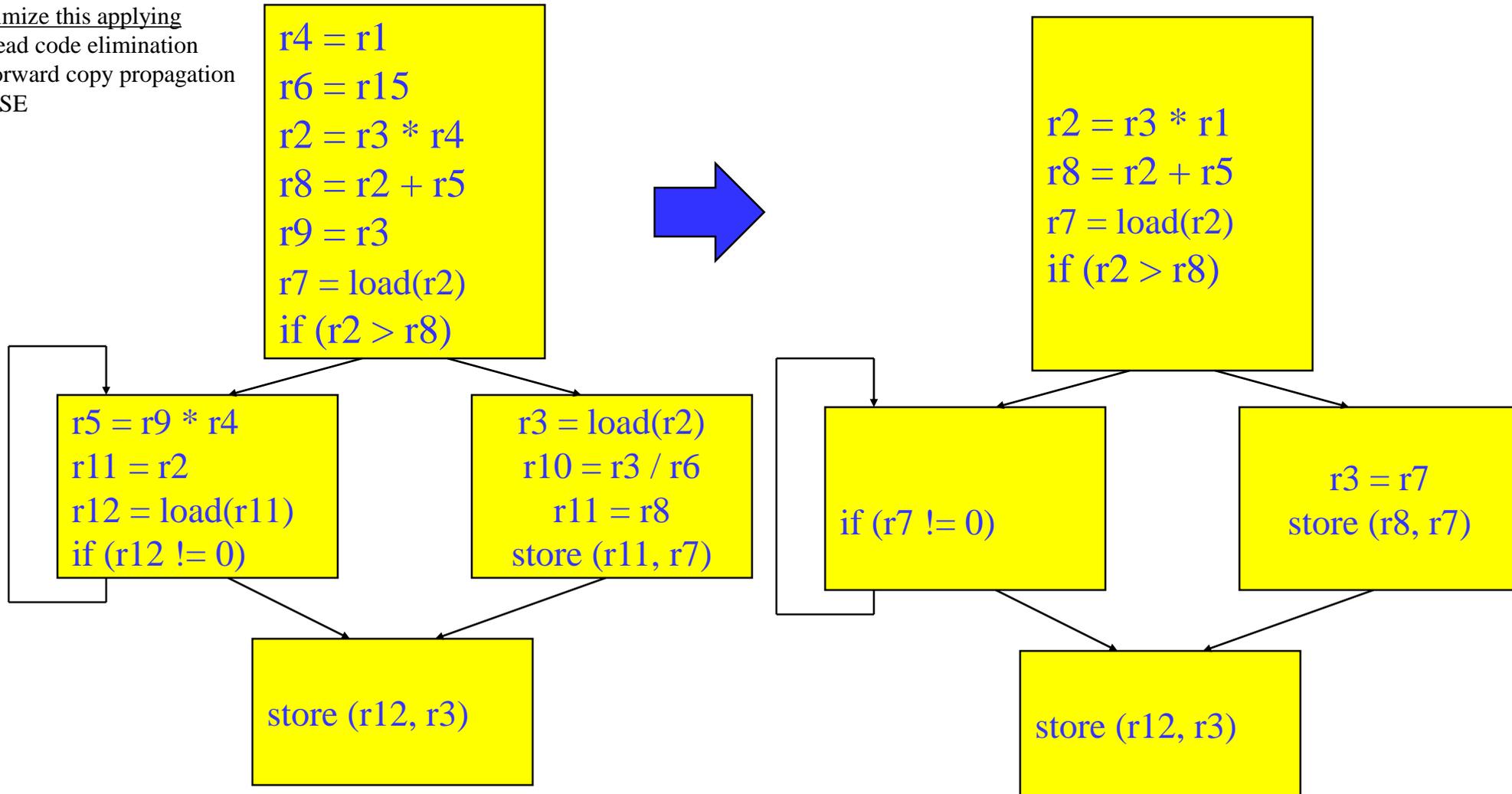
Class Problem

1. dead code elimination
2. forward copy propagation
3. CSE

BB1

```
1. r4 = r1
2. r6 = r15
3. r2 = r3 * r4
4. r8 = r2 + r5
5. r9 = r3
6. r7 = load(r2)
7. if (r2 > r8)
```

BB2

```
8. r5 = r9 * r4
9. r11 = r2
10. r12 = load(r11)
11. if (r12 != 0)
```

BB3

```
12. r3 = load(r2)
13. r10 = r3 / r6
14. r11 = r8
15. store (r11, r7)
```

BB4

```
16. store (r12, r3)
```

Class Problem Solution

r4 = r1
r6 = r15
r2 = r3 * r4
r8 = r2 + r5
r9 = r3
r7 = load(r2)
if (r2 > r8)

r5 = r9 * r4
r11 = r2
r12 = load(r11)
if (r12 != 0)

r3 = load(r2)
r10 = r3 / r6
r11 = r8
store (r11, r7)

store (r12, r3)

r2 = r3 * r1
r8 = r2 + r5
r7 = load(r2)
if (r2 > r8)

if (r7 != 0)

r3 = r7
store (r8, r7)

store (r12, r3)
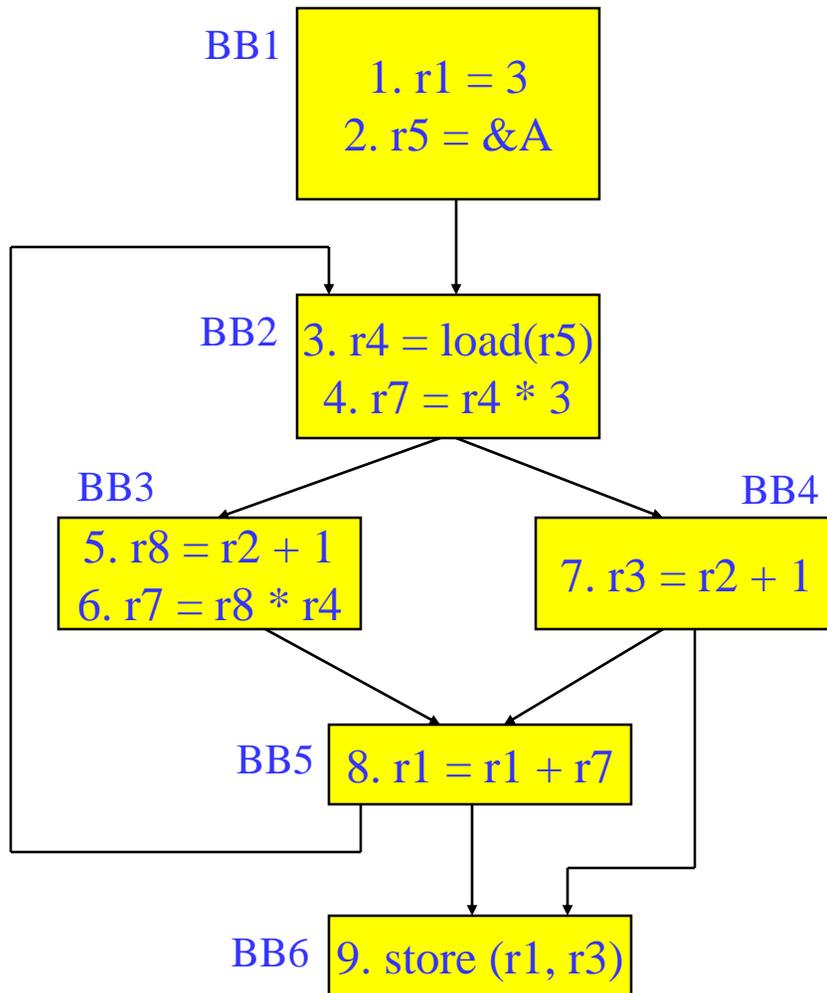
- 13 -

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
- ❖ 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

BB1

BB2

BB3

BB4

BB5

BB6

LICM Example

BB1
```
1. r1 = 3
2. r5 = &A
```

BB2
```
3. r4 = load(r5)
4. r7 = r4 * 3
```

BB3
```
5. r8 = r2 + 1
6. r7 = r8 * r4
```

BB4
```
7. r3 = r2 + 1
```

BB5
```
8. r1 = r1 + r7
```

BB6
```
9. store (r1, r3)
```
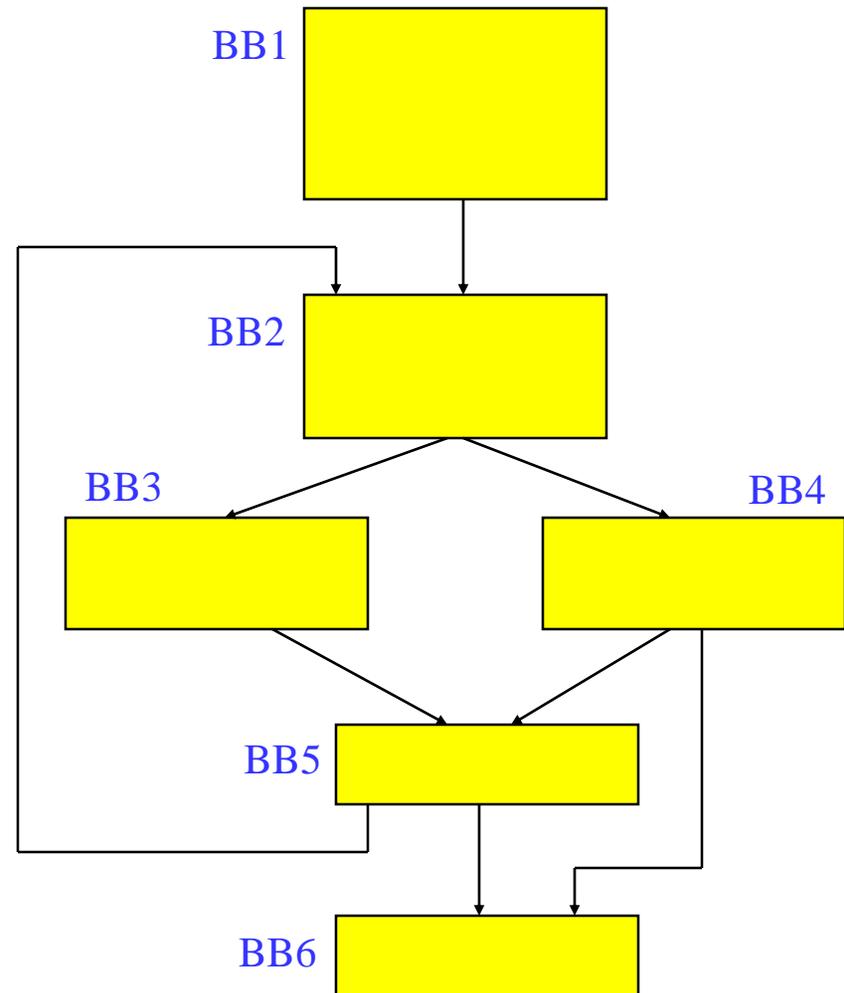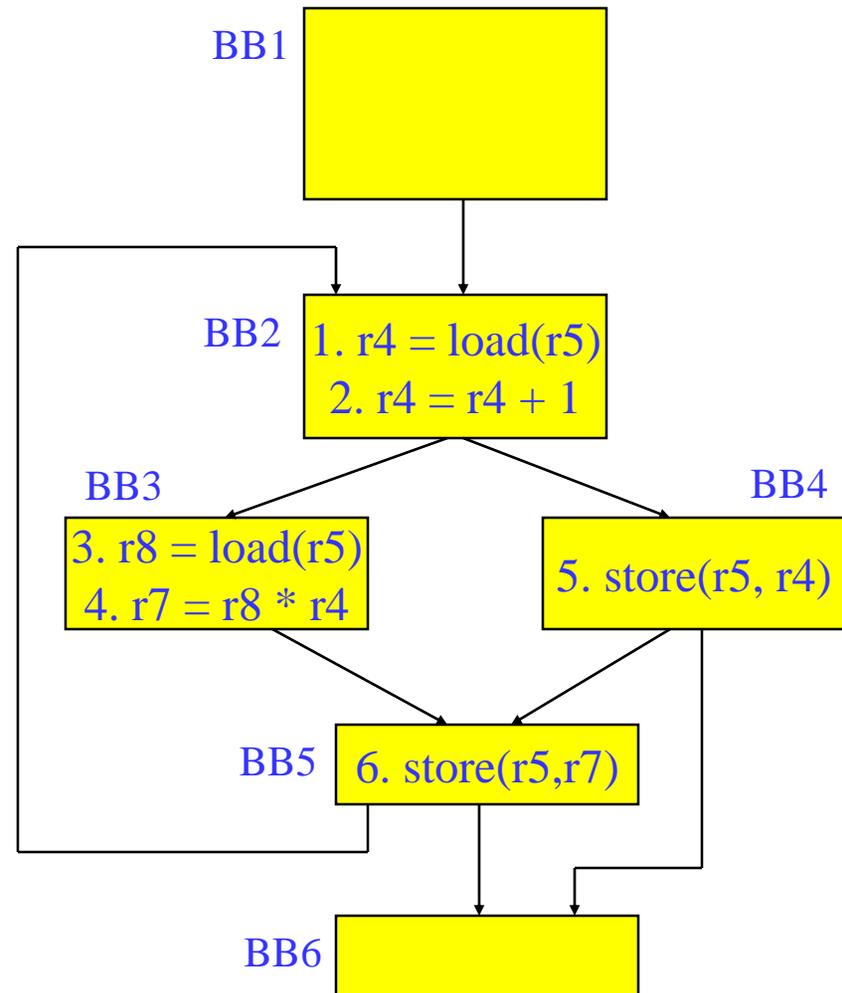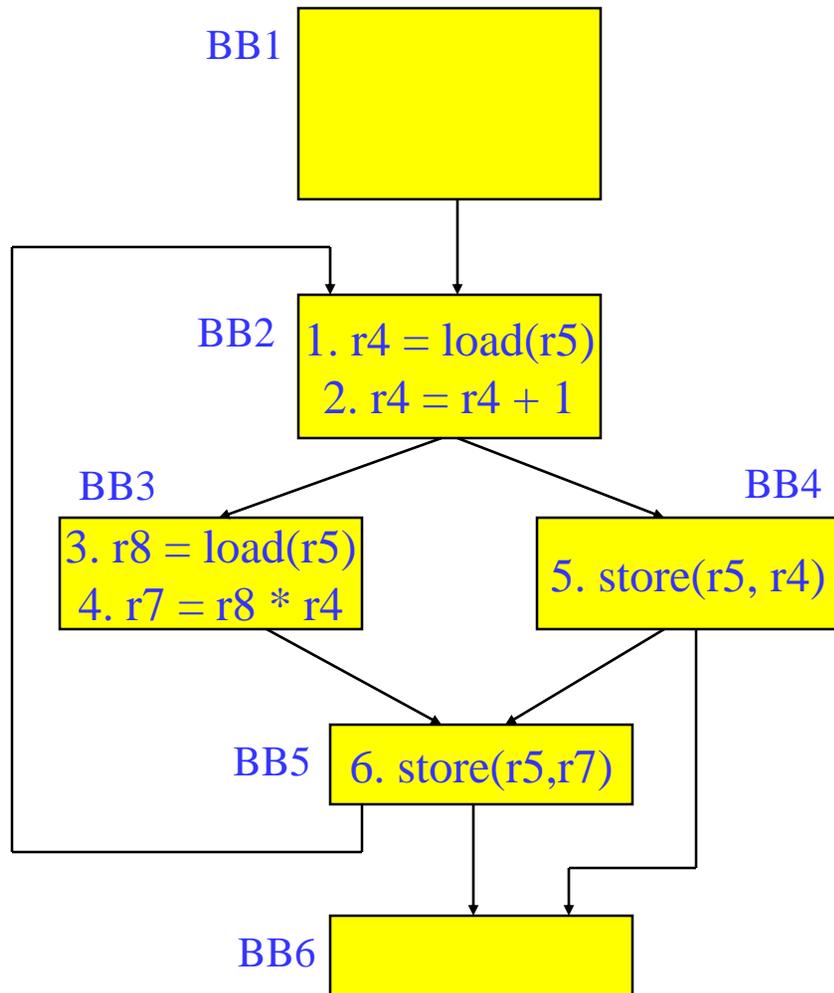
Global Variable Migration

- 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

BB1

BB2
1. r4 = load(r5)
2. r4 = r4 + 1

BB3
3. r8 = load(r5)
4. r7 = r8 * r4

BB4
5. store(r5, r4)

BB5
6. store(r5,r7)

BB6

Global Variable Migration Example

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

BB1

BB2 1. r5 = r4 - 3
     2. r4 = r4 + 1

BB3   BB4

                             3. r7 = r4 * r9

BB5 4. r6 = r4 << 2

BB6
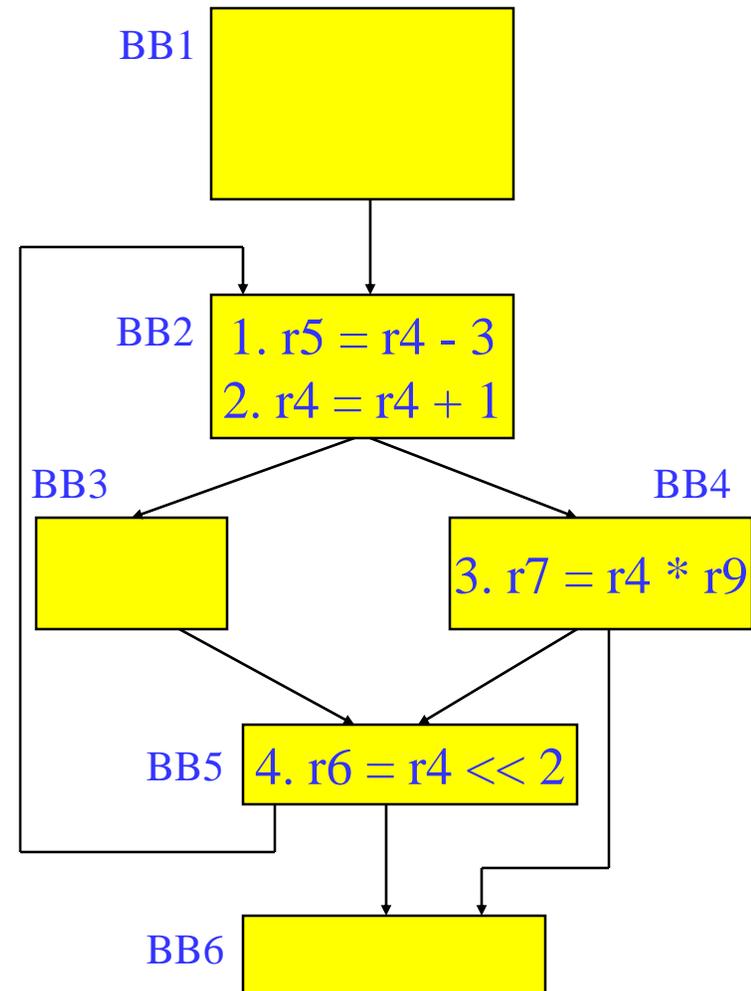
Induction Variable Strength Reduction (2)

- ❖ 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

BB1

BB2
1. r5 = r4 - 3
2. r4 = r4 + 1

BB3

BB4
3. r7 = r4 * r9

BB5
4. r6 = r4 << 2

BB6
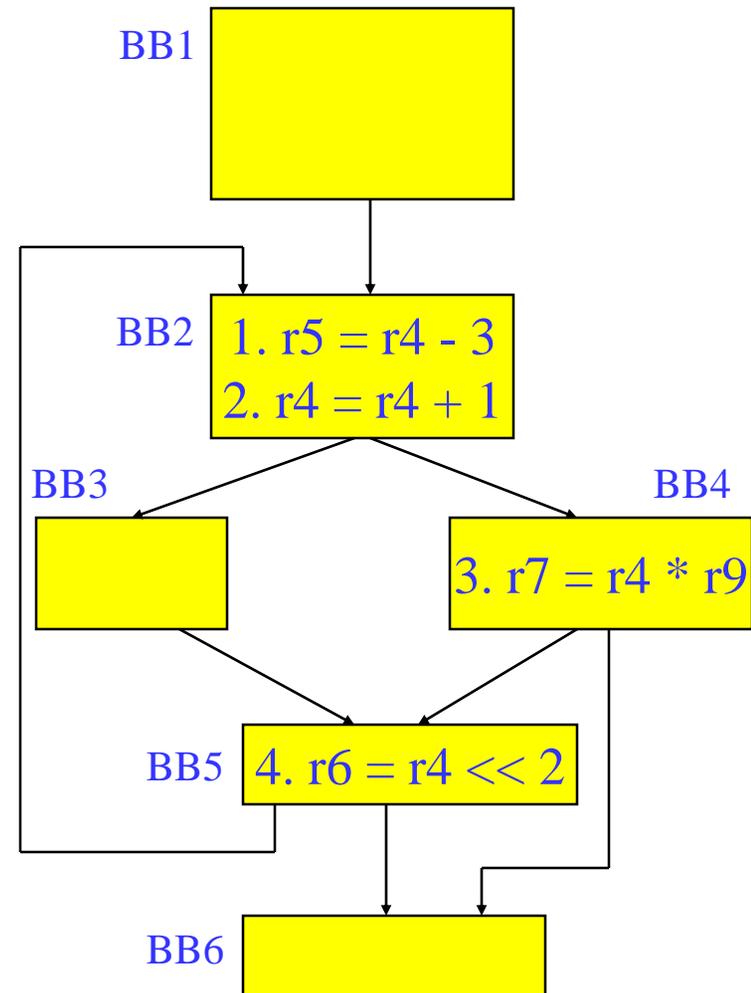
Induction Variable Strength Reduction - Example

BB1

BB2
1. r5 = r4 - 3
2. r4 = r4 + 1

BB3

BB4
3. r7 = r4 * r9

BB5
4. r6 = r4 << 2

BB6

Class Problem

Optimize this applying
induction var str
reduction

BB1

1. r1 = 0
2. r2 = 0

BB2

3. r5 = r5 + 1
4. r11 = r5 * 2
5. r10 = r11 + 2
6. r12 = load (r10+0)
7. r9 = r1 << 1
8. r4 = r9 - 10
9. r3 = load(r4+4)
10. r3 = r3 + 1
11. store(r4+0, r3)
12. r7 = r3 << 2
13. r6 = load(r7+0)
14. r13 = r2 - 1
15. r1 = r1 + 1
16. r2 = r2 + 1

BB3   r13, r12, r6, r10
            liveout

Class Problem Solution

Optimize this applying
induction var str reduction

Left box (top):

r1 = 0
r2 = 0

Left box (main):

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

Left box (bottom):

r13, r12, r6, r10
liveout

Right box (top):

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

Right box (main):

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

Right box (bottom):

r13, r12, r6, r10
liveout

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