Dataflow II: Finish Dataflow Analysis, Start on Classical Optimizations

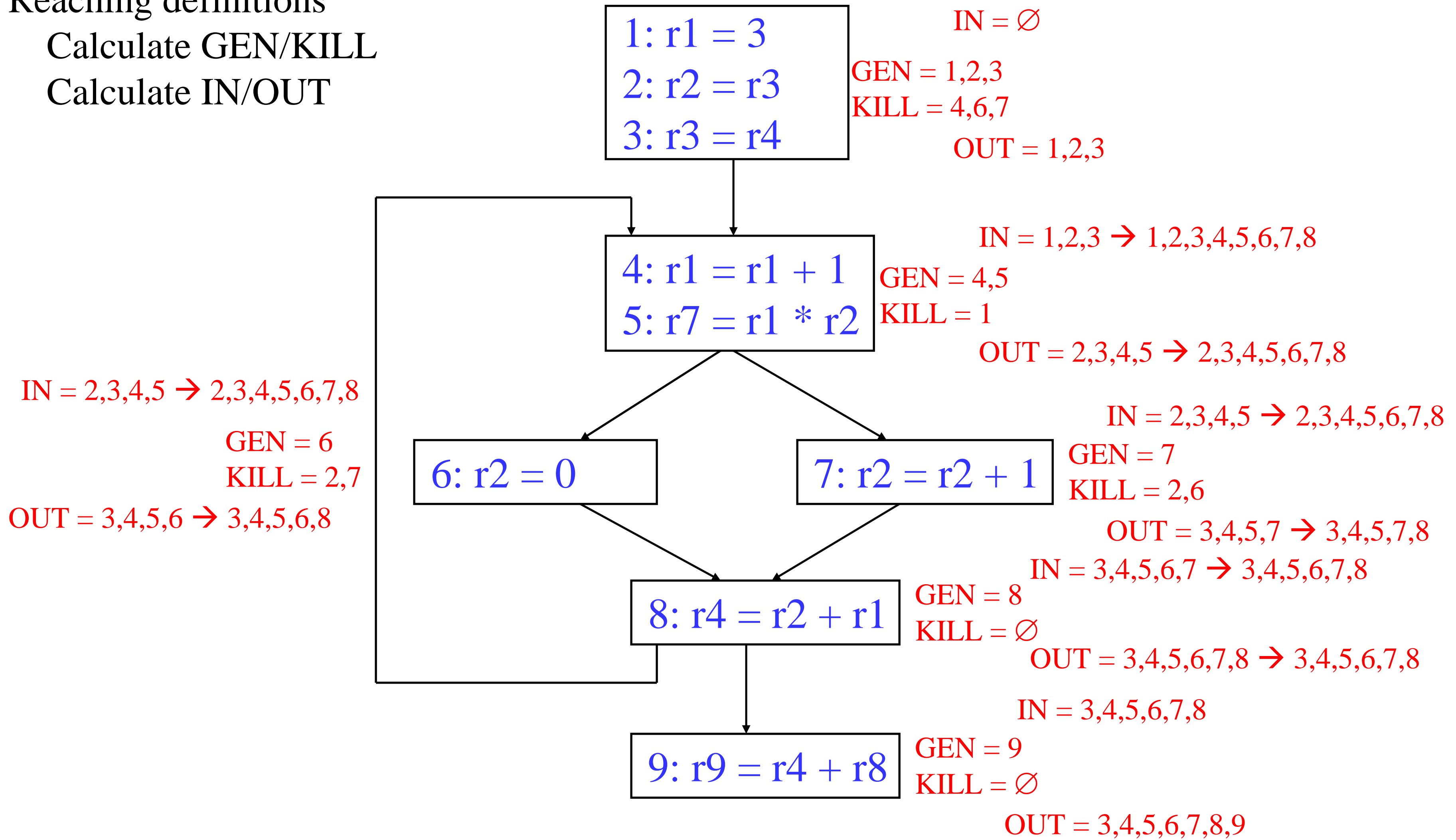
EECS 483 – Lecture 24
University of Michigan
Wednesday, November 29, 2006

Announcements and Reading

- Project 3 – should have started work on this
- Schedule for the rest of the semester
  - Today – Dataflow analysis
  - Wednes 11/29 – Finish dataflow, optimizations
  - Mon 12/4 – Optimizations, start on register allocation
  - 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
  - 10.5, 10.6. 10.10, 10.11

Class Problem – From Last Time

Reaching definitions
- Calculate GEN/KILL
- Calculate IN/OUT



Some Things to Think About

- Liveness and reaching defs are basically the same thing!!!!!!!!!!!!!!!!!!
  - » All dataflow is basically the same with a few parameters
    - Meaning of gen/kill (use/def)
    - Backward / Forward
    - All paths / some paths (must/may)
      - So far, we have looked at may analysis algorithms
      - How do you adjust to do must algorithms?
- Dataflow can be slow
  - » How to implement it efficiently? (Block traversal order can speed things up)
  - » How to represent the info? (Bitvectors)

Generalizing Dataflow Analysis

- Transfer function
  - How information is changed by “something” (BB)
  - OUT = GEN + (IN – KILL)   forward analysis
  - IN = GEN + (OUT – KILL)   backward analysis
- Meet function
  - How information from multiple paths is combined
  - IN = Union(OUT(predecessors)) forward analysis
  - OUT = Union(IN(successors)) backward analysis
  - Note, this is only for “any path

Generalized Dataflow Algorithm

- while (change)
  - change = false
  - for each BB
    - apply meet function
    - apply transfer function
    - if any changes → change = true

Liveness Using GEN/KILL

- Liveness = upward exposed uses

```
for each basic block in the procedure, X, do
    up_use_GEN(X) = 0
    up_use_KILL(X) = 0
    for each operation in reverse sequential order in X, op, do
        for each destination operand of op, dest, do
            up_use_GEN(X) -= dest
            up_use_KILL(X)  += dest
        endfor
        for each source operand of op, src, do
            up_use_GEN(X) += src
            up_use_KILL(X) -= src
        endfor
    endfor
endfor
```

Example - Liveness with GEN/KILL

meet: OUT = Union(IN(succs))
xfer: IN = GEN + (OUT – KILL)

BB1

r1 = MEM[r2+0]
r2 = r2 + 1
r3 = r1 * r4

up_use_GEN(1) = r2,r4
up_use_KILL(1) = r1,r3

up_use_GEN(2) = r1,r5
up_use_KILL(2) = r3,r7

BB2

r1 = r1 + 5
r3 = r5 – r1
r7 = r3 * 2

BB3

r2 = 0
r7 = 23
r1 = 4

up_use_GEN(3) = 0
up_use_KILL(3) = r1, r2, r7

BB4

r3 = r3 + r7
r1 = r3 – r8
r3 = r1 * 2

up_use_GEN(4.3) = r3,r7,r8
up_use_GEN(4.2) = r3,r8
up_use_GEN(4.1) = r1

up_use_KILL(4.3) = r1
up_use_KILL(4.2) = r1
up_use_KILL(4.1) = r3

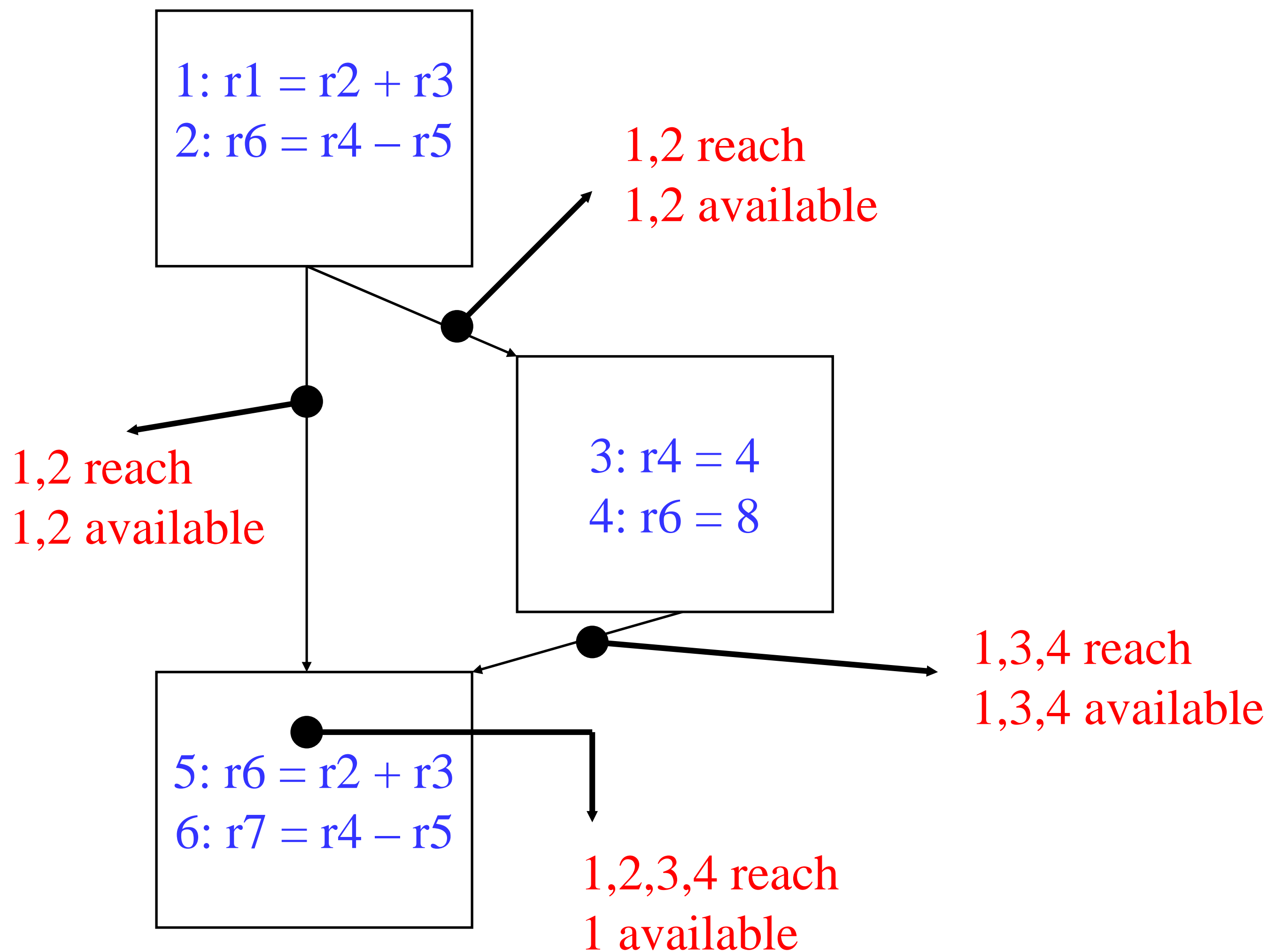
Beyond Liveness (Upward Exposed Uses)

- Upward exposed defs
  - IN = GEN + (OUT – KILL)
  - OUT = Union(IN(successors))
  - Walk ops reverse order
    - GEN += dest; KILL += dest
- Downward exposed uses
  - IN = Union(OUT(predecessors))
  - OUT = GEN + (IN-KILL)
  - Walk ops forward order
    - GEN += src; KILL -= src;
    - GEN -= dest; KILL += dest;
- Downward exposed defs
  - IN = Union(OUT(predecessors))
  - OUT = GEN + (IN-KILL)
  - Walk ops forward order
    - GEN += dest; KILL += dest;

What About All Path Problems?

- Up to this point
  - Any path problems (maybe relations)
    - Definition reaches along some path
    - Some sequence of branches in which def reaches
    - Lots of defs of the same variable may reach a point
  - Use of Union operator in meet function
- All-path: Definition guaranteed to reach
  - Regardless of sequence of branches taken, def reaches
  - Can always count on this
  - Only 1 def can be guaranteed to reach
  - Availability (as opposed to reaching)
    - Available definitions
    - Available expressions (could also have reaching expressions, but not that useful)

Reaching vs Available Definitions



Available Definition Analysis (Adefs)

- 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
  - GEN/KILL/IN/OUT similar to reaching defs
    - Initialization of IN/OUT is the tricky part

Compute Adef GEN/KILL Sets

Exactly the same as reaching defs !!!!!!!

```
for each basic block in the procedure, X, do
    GEN(X) = 0
    KILL(X) = 0
    for each operation in sequential order in X, op, do
        for each destination operand of op, dest, do
            G = op
            K = {all ops which define dest – op}
            GEN(X) = G + (GEN(X) – K)
            KILL(X) = K + (KILL(X) – G)
        endfor
    endfor
endfor
```

Compute Adef IN/OUT Sets

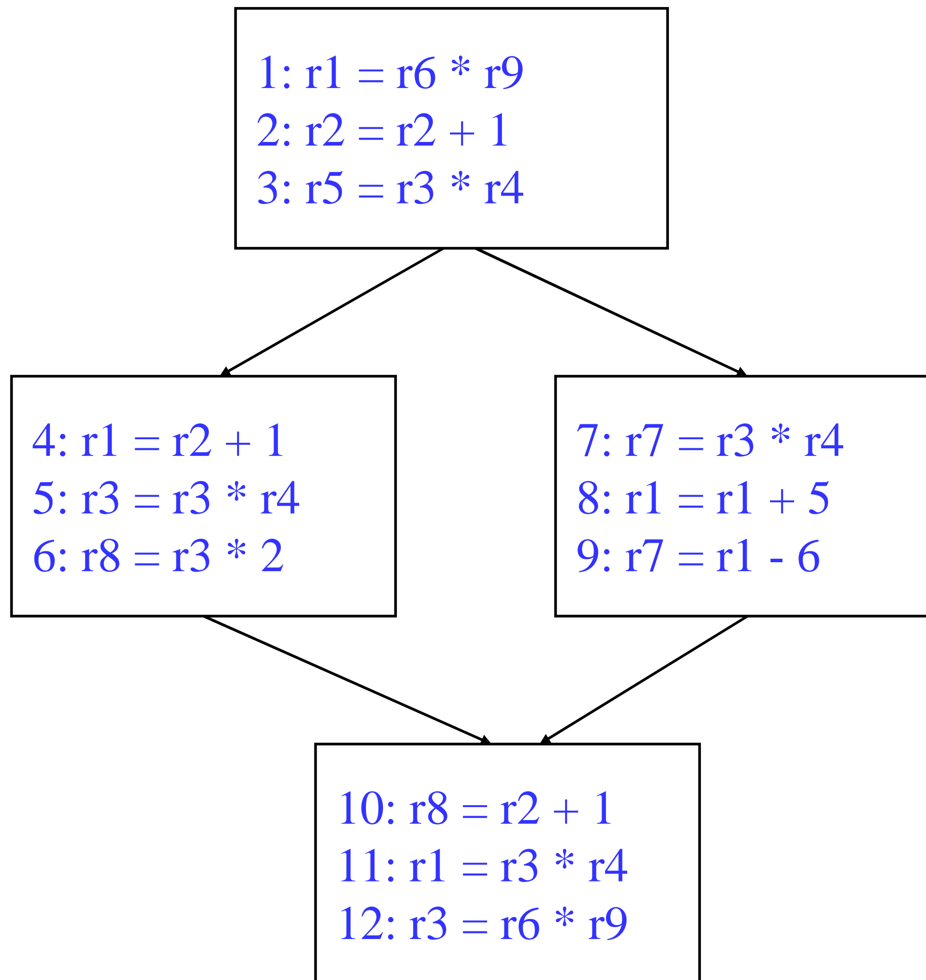
```
U = universal set of all operations in the Procedure
IN(0) = 0
OUT(0) = GEN(0)
for each basic block in procedure, W, (W != 0), do
    IN(W) = 0
    OUT(W) = U – KILL(W)

change = 1
while (change) do
    change = 0
    for each basic block in procedure, X, do
        old_OUT = OUT(X)
        IN(X) = Intersect(OUT(Y)) for all predecessors Y of X
        OUT(X) = GEN(X) + (IN(X) – KILL(X))
        if (old_OUT != OUT(X)) then
            change = 1
        endif
    endfor
endfor
```

Available Expression Analysis (Aexprs)

- An expression is a RHS of an operation
  - r2 = r3 + r4, r3+r4 is an expression
- An expression e is available at a point p if along all paths from e to p, e is not killed
- An expression is killed between 2 points when one of its source operands are redefined
  - r1 = r2 + r3 kills all expressions involving r1
- Algorithm
  - Forward dataflow analysis
  - Use the Intersect function as the meet operator to guarantee the all-path requirement
  - Looks exactly like adefs, except GEN/KILL/IN/OUT are the RHS's of operations rather than the LHS's

Class Problem - Aexprs Calculation

Compute the Aexpr IN/OUT sets for each BB



Optimization – Put Dataflow To Work!

- Make the code run faster on the target processor
  - Anything goes
    - Look at benchmark kernels, what's the bottleneck??
    - Invent your own optis
- Classes of optimization
  - 1. Classical (machine independent)
    - Reducing operation count (redundancy elimination)
    - Simplifying operations
  - 2. Machine specific
    - Peephole optimizations
    - Take advantage of specialized hardware features
  - 3. ILP enhancing
    - Increasing parallelism
    - Possibly increase instructions

Types of Classical Optimizations

- Operation-level – 1 operation in isolation
  - Constant folding, strength reduction
  - Dead code elimination (global, but 1 op at a time)
- Local – Pairs of operations in same BB
  - May or may not use dataflow analysis
- Global – Again pairs of operations
  - But, operations in different BBs
  - Dataflow analysis necessary here
- Loop – Body of a loop

Caveat

- Traditional compiler class
  - Fancy implementations of optimizations, efficient algorithms
  - Bla bla bla
  - Spend entire class on 1 optimization
- For this class – Go over concepts of each optimization
  - What it is
  - When can it be applied (set of conditions that must be satisfied)

Constant Folding

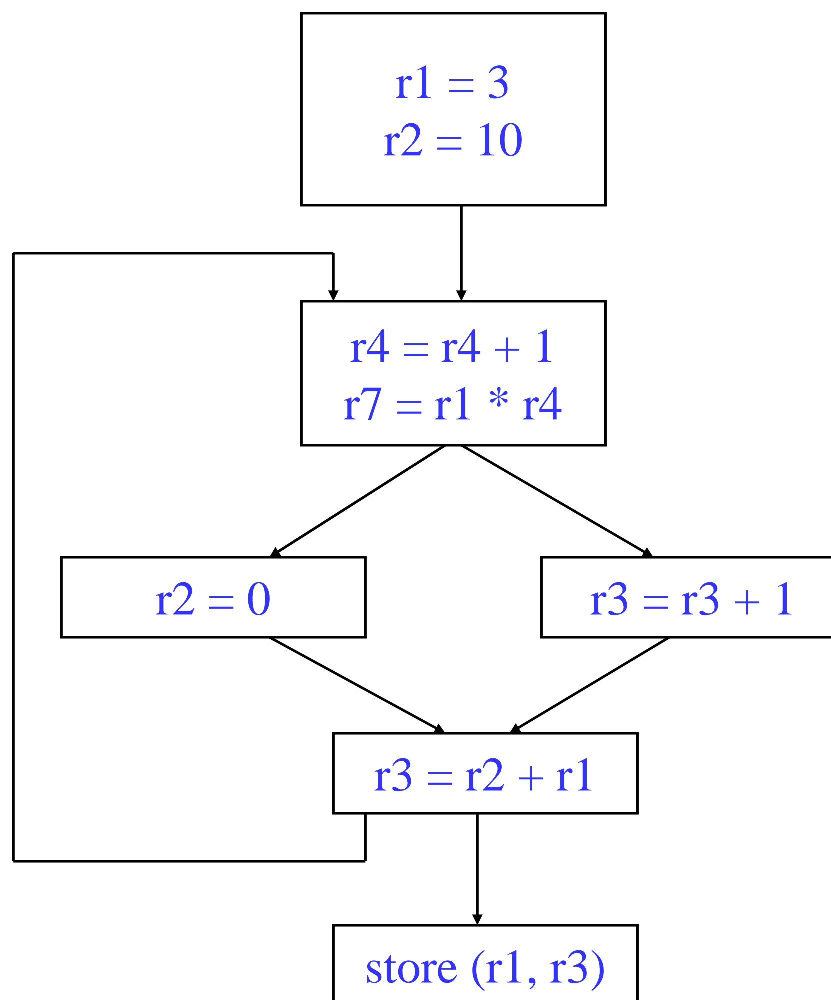
- Simplify operation based on values of src operands
  - » Constant propagation creates opportunities for this
- All constant operands
  - » Evaluate the op, replace with a move
    - r1 = 3 * 4 → r1 = 12
    - r1 = 3 / 0 → ??? Don't evaluate excepting ops !, what about FP?
  - » Evaluate conditional 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

Strength Reduction

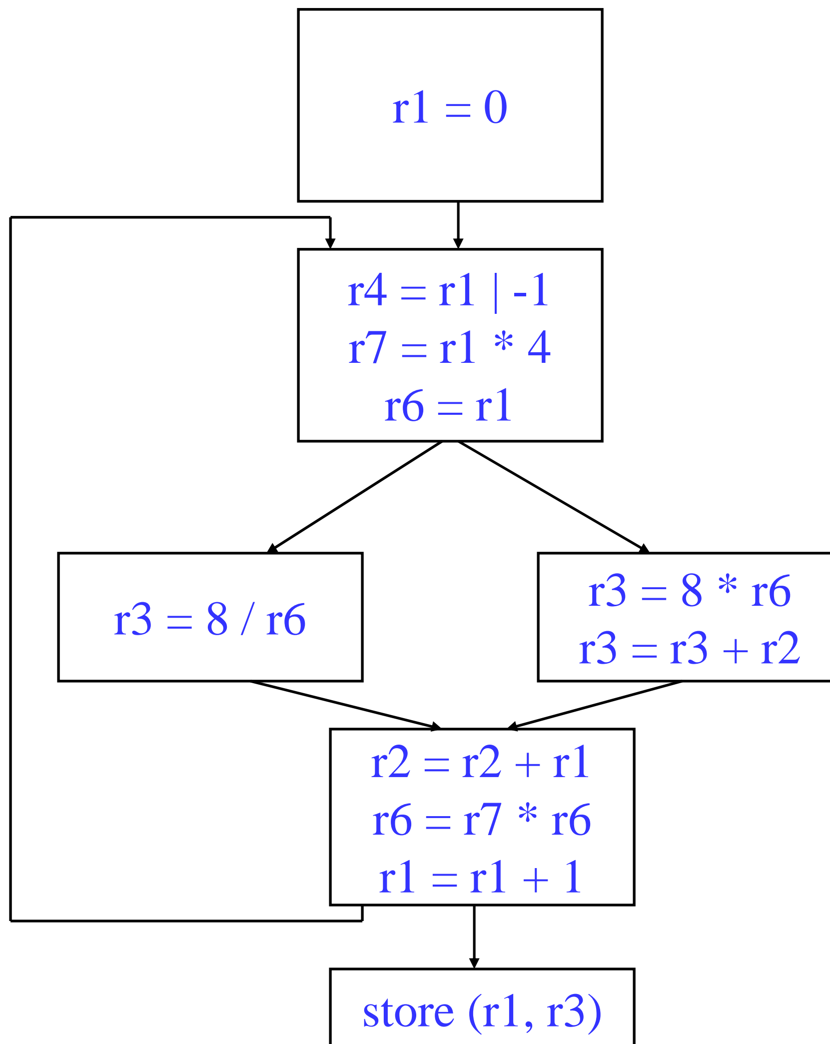
- Replace expensive ops with cheaper ones
  - » Constant propagation creates opportunities for this
- Power of 2 constants
  - » Mpy by power of 2: r1 = r2 * 8 → r1 = r2 << 3
  - » Div by power of 2: r1 = r2 / 4 → r1 = r2 >> 2
  - » Rem by power of 2: r1 = r2 REM 16 → r1 = r2 & 15
- More exotic
  - » Replace multiply by constant by sequence of shift and adds/subs
    - r1 = r2 * 6
      - r100 = r2 << 2; r101 = r2 << 1; r1 = r100 + r101
    - r1 = r2 * 7
      - r100 = r2 << 3; r1 = r100 – r2

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 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

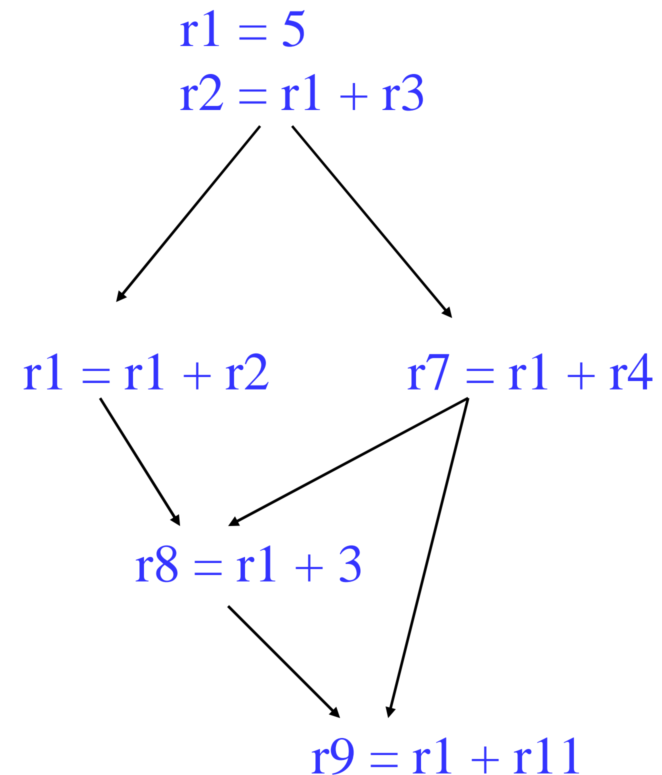


Optimize this applying

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



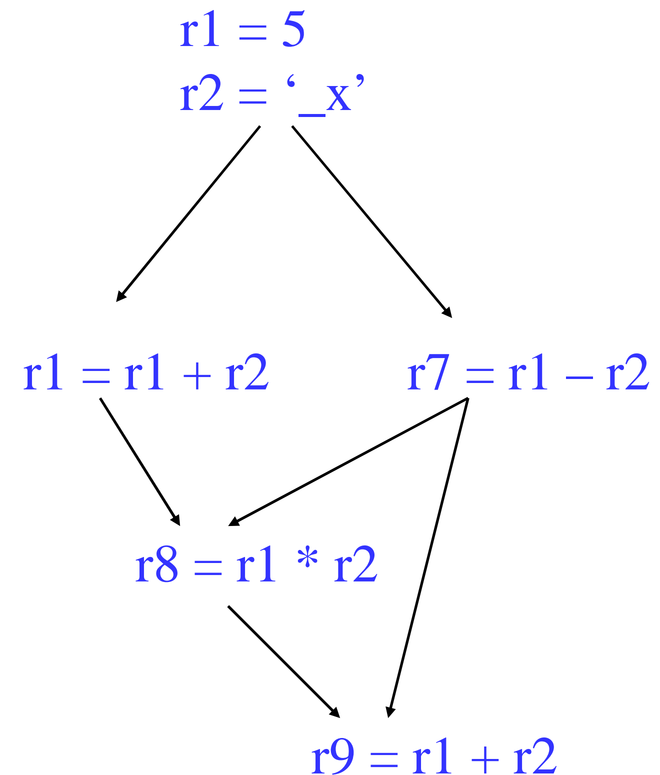
Local Constant Propagation

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

```
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
```

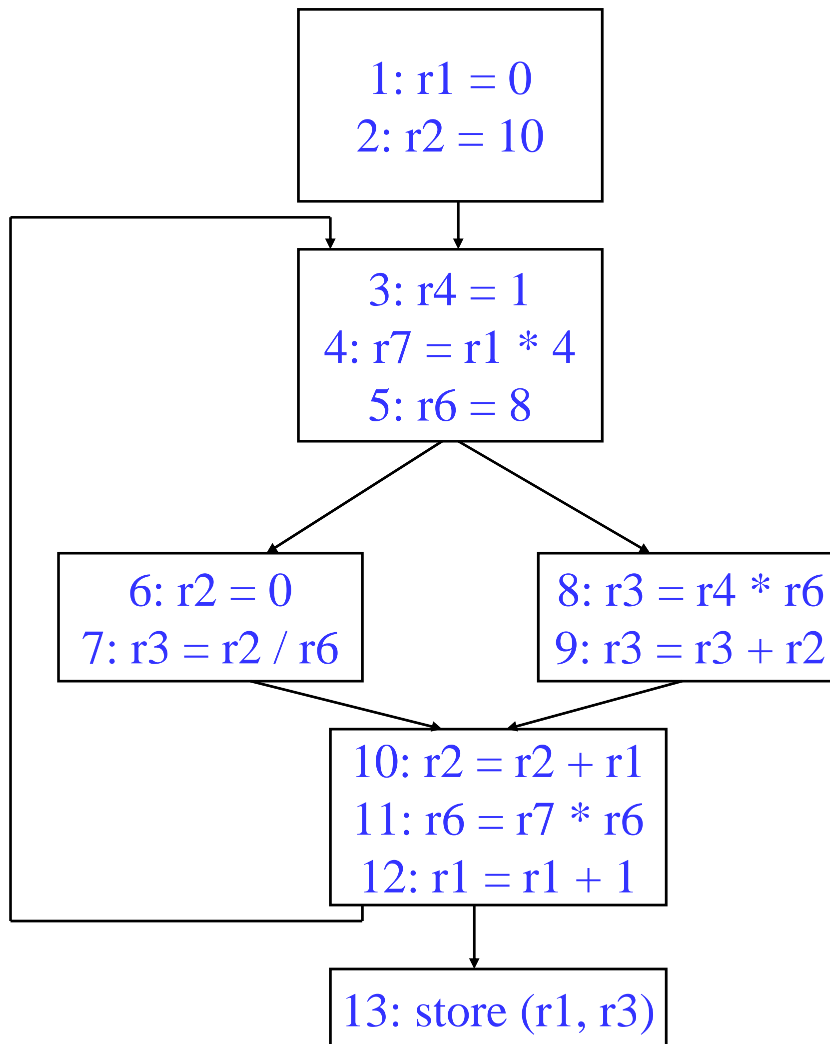
Global Constant Propagation

- 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 adef_IN(BB(Y))
  - » 5. dest(X) is not modified between the top of BB(Y) and Y
    - Rules 4/5 guarantee X is available
  - » 6. If 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

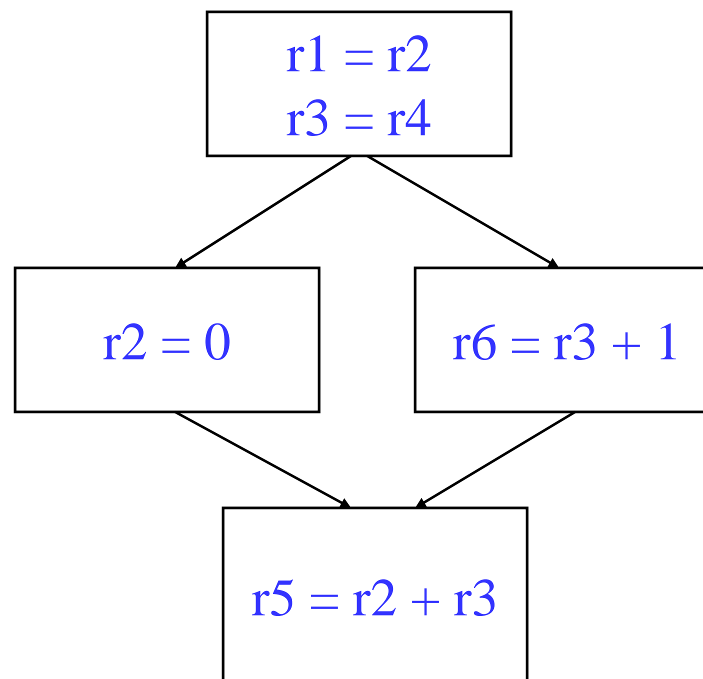


Optimize this applying

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

Forward Copy Propagation

- Forward propagation of the RHS of moves
  - X: r1 = r2
  - …
  - Y: r4 = r1 + 1 → r4 = r2 + 1
- Benefits
  - Reduce chain of dependences
  - Possibly 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



Backward Copy Propagation

- Backward prop. of the LHS of moves
  - X: r1 = r2 + r3 → r4 = r2 + r3
  - …
  - r5 = r1 + r6 → r5 = r4 + r6
  - …
  - Y: r4 = r1 → noop
- Rules (ops X and Y in same BB)
  - dest(X) is a register
  - dest(X) not live out of BB(X)
  - Y is a move
  - dest(Y) is a register
  - Y consumes dest(X)
  - dest(Y) not consumed in (X…Y)
  - dest(Y) not defined in (X…Y)
  - There are no uses of dest(X) after the first redefinition of 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
```