Basic Optimizations

Compilers for High Performance Architectures

The compiler back End

- Takes intermediate code and generates machine-dependent code
- Performs multiple optimization steps
  - Machine independent optimizations
  - Machine dependent optimizations

Basic optimizations

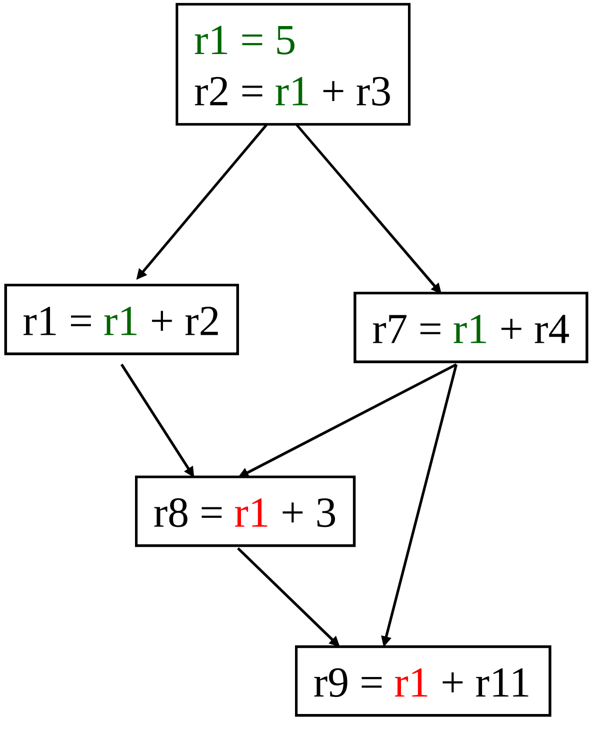
- Constant folding
- Constant propagation
- Constant combining
- Operation folding
- Copy propagation
- Common subexpression elimination
- Algebraic simplification
- Dead code removal
- Tree height reduction

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

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



Constant Combining

- Combine 2 dependent ops into 1 by combining the literals
  - r1 = r2 + 4
  - ...
  - r5 = r1 - 9 → r5 = r2 – 5
- First op often becomes dead
- Rules (ops X and Y in same BB)
  - X is of the form rx +- K
  - dest(X) != src1(X)
  - Y is of the form ry +- K (comparison also ok)
  - Y consumes dest(X)
  - src1(X) not modified in (X…Y)

```
r1 = r2 + 4
r3 = r1 < 0
r2 = r3 + 6
r7 = r1 – 3
r8 = r7 + 5
```

Operation Folding

- Combine 2 dependent ops into 1 complex op
- First op often becomes dead
- Actually an ISA dependent optimization
- Rules (ops X and Y in same BB)
  - X is an arithmetic operation
  - dest(X) != any src(X)
  - Y is an arithmetic operation
  - Y consumes dest(X)
  - X and Y can be merged
  - src(X) not modified in (X...Y)

- Multiply & Add

  r1 = r2 * r3

  r6 = r1 * r4

  r5 = r1 + r4 → r5 = r2 * r3 + r4

- Shift and add (PA –RISC)

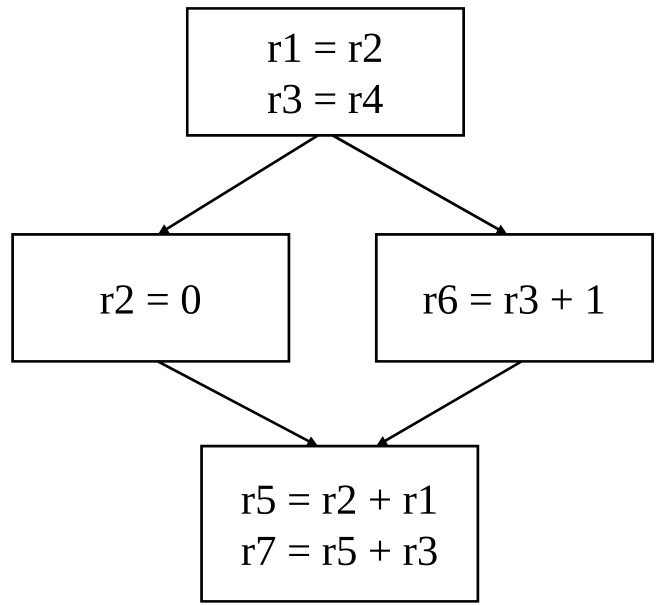
  Multiply by 5

  r2 = r1 << 2 → dead !!!

  r2 = r2 + r1 → r2 = r1 << 2 + r1

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



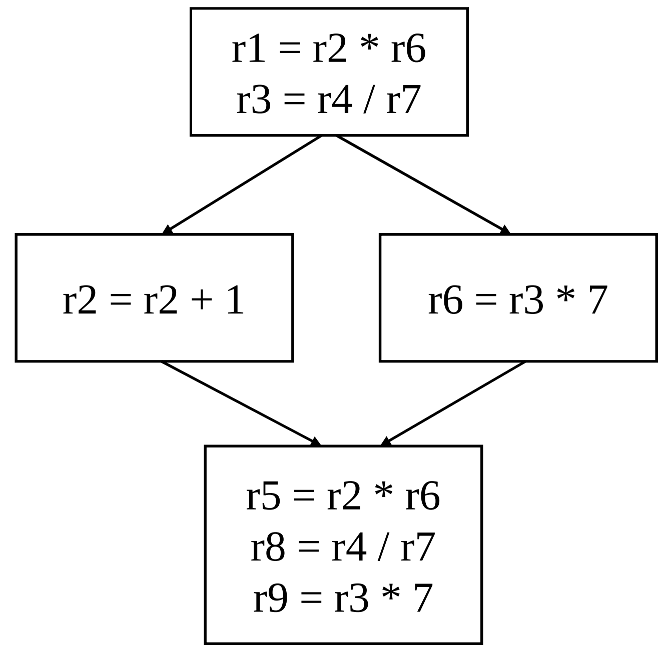
Backward Copy Propagation

- Backward propagation of the LHS of moves
  - r1 = r2 + r3 → r4 = r2 + r3
  - …
  - r5 = r1 + r6 → r5 = r4 + r6
  - …
  - 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
```

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

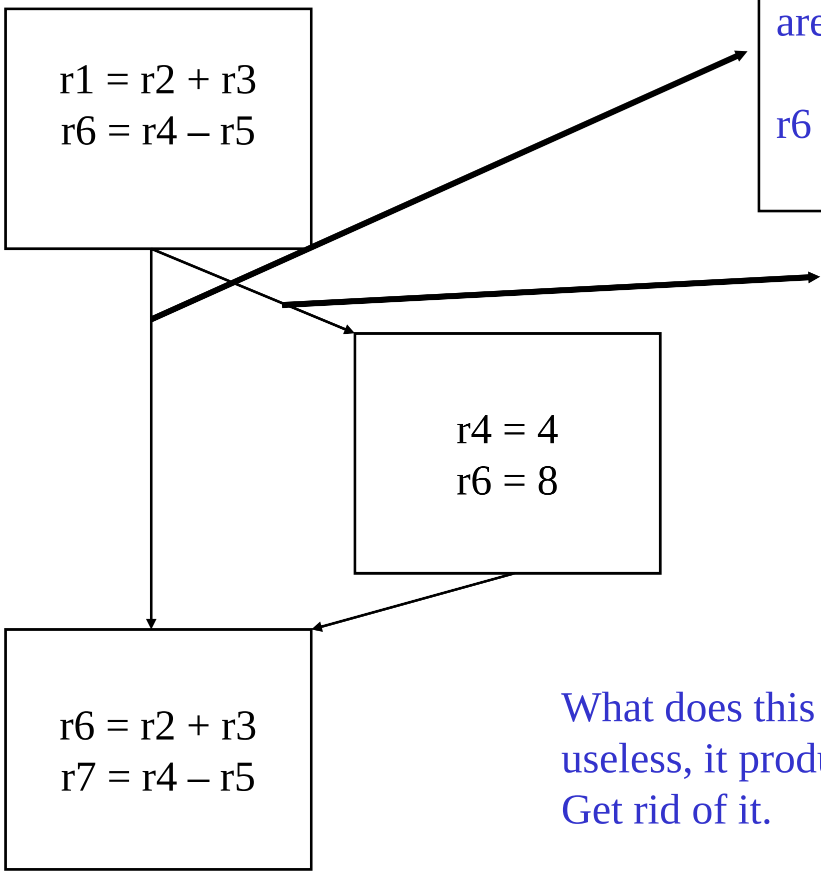
Algebraic simplification

- Also known as strength reduction
- Replace expensive ops with cheaper ones
  - Constant propagation creates opportunities for this
- Power of 2 constants
  - Multiply by power of 2, replace with left shift
    - r1 = r2 * 8 → r1 = r2 << 3
  - Divide by power of 2, replace with right shift
    - r1 = r2 / 4 → r1 = r2 >> 2
  - Remainder by power of 2, replace with logical and
    - 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

Live Variable (Liveness) Analysis

- Algorithm sketch
  - For each BB, y is live if it is used before defined in the BB or it is live leaving the block
  - Backward dataflow analysis as propagation occurs from uses upwards to defs
- 4 sets
  - USE = set of external variables consumed in the BB
  - DEF = set of variables defined in the BB
  - IN = set of variables that are live at the entry point of a BB
  - OUT = set of variables that are live at the exit point of a BB

Liveness Example



r2, r3, r4, r5 are all live as they are consumed later

r6 is dead as it is redefined later

r4 is dead, as it is redefined.
r6 is dead as it is redefined later

r2, r3, r5 are live

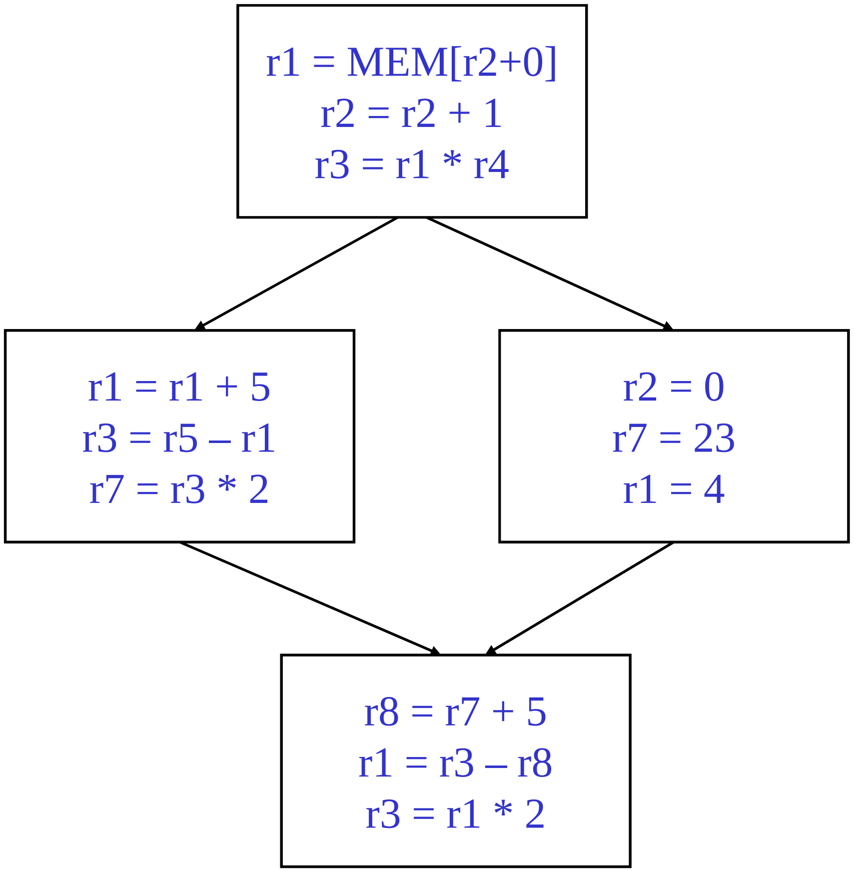
What does this mean? r6 = r4 – r5 is useless, it produces a dead value !!
Get rid of it.

Compute USE/DEF Sets for each BB

def is the union of all the LHS's
use is all the VRs that are used before defined

```
for each basic block in the procedure, X, do
    DEF(X) = 0
    USE(X) = 0
    for each operation in sequential order in X, op, do
        for each source operand of op, src, do
            if (src not in DEF(X)) then
                USE(X) += src
            endif
        endfor
        for each destination operand of op, dest, do
            DEF(X) += dest
        endfor
    endfor
endfor
```

Class Problem: USE/DEF Calculation

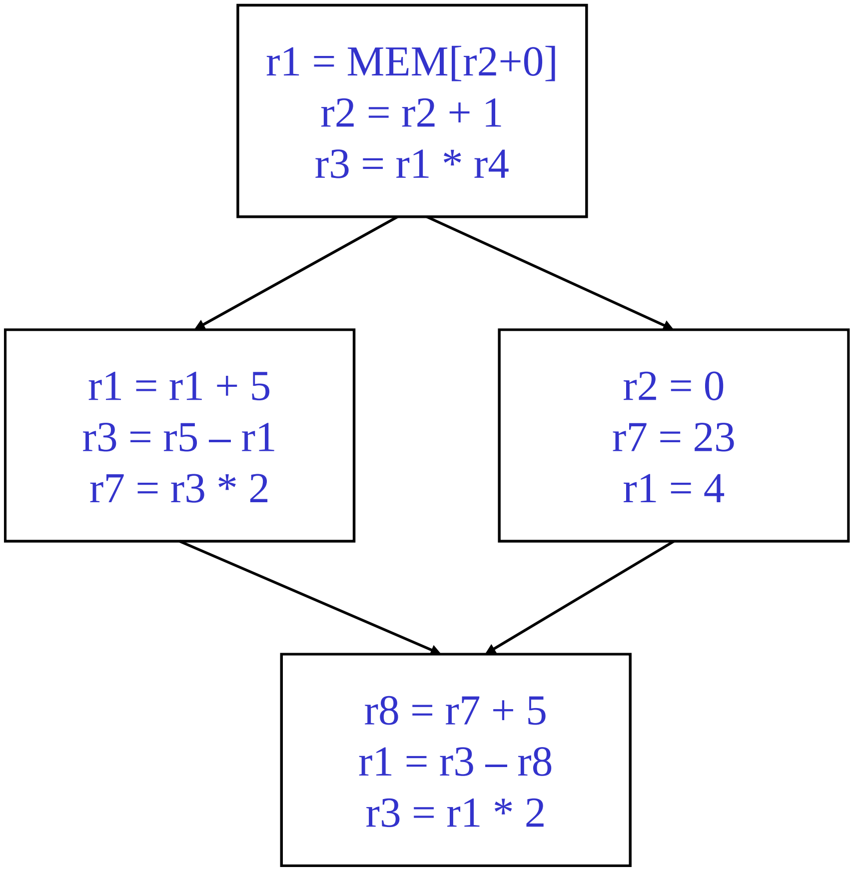


Compute IN/OUT Sets for all BBs

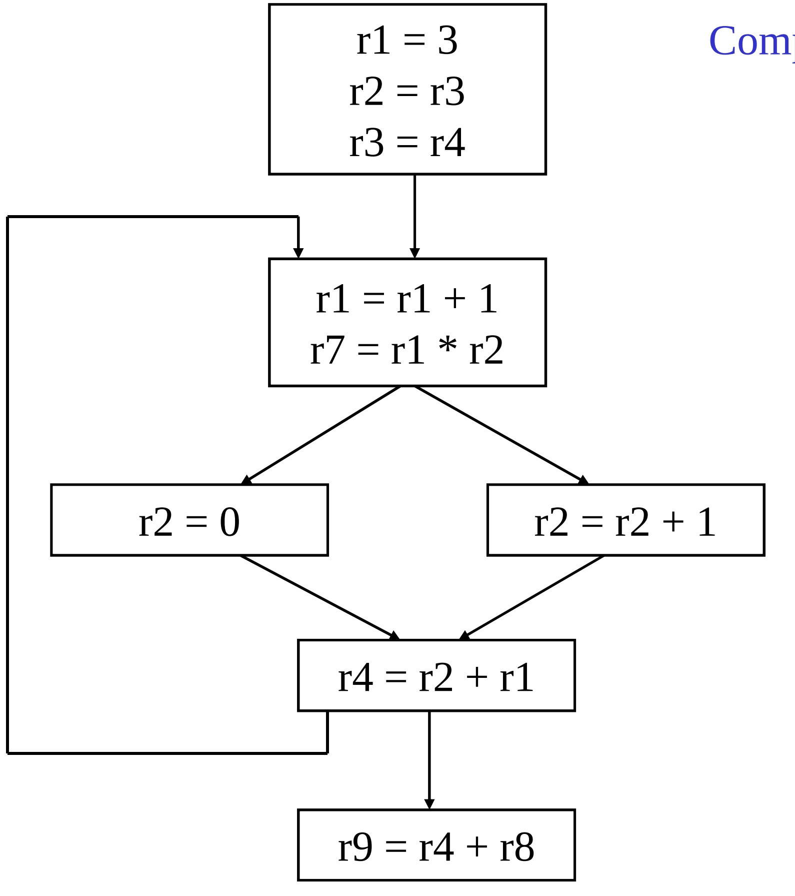
IN = set of variables that are live when the BB is entered
OUT = set of variables that are live when the BB is exited

```
initialize IN(X) to 0 for all basic blocks X
change = 1
while (change) do
    change = 0
    for each basic block in procedure, X, do
        old_IN = IN(X)
        OUT(X) = Union(IN(Y)) for all successors Y of X
        IN(X) = USE(X) + (OUT(X) – DEF(X))
        if (old_IN != IN(X)) then
            change = 1
        endif
    endfor
endfor
```

Class Problem: IN/OUT Calculation



Class Problem



Compute liveness

- Calculate USE/DEF for each BB
- Calculate IN/OUT for each BB

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 */
- Generalized dataflow algorithm
  - while (change)
    - change = false
    - for each BB
      - apply meet function
      - apply transfer functions
      - if any changes → change = true

DU/UD Chains

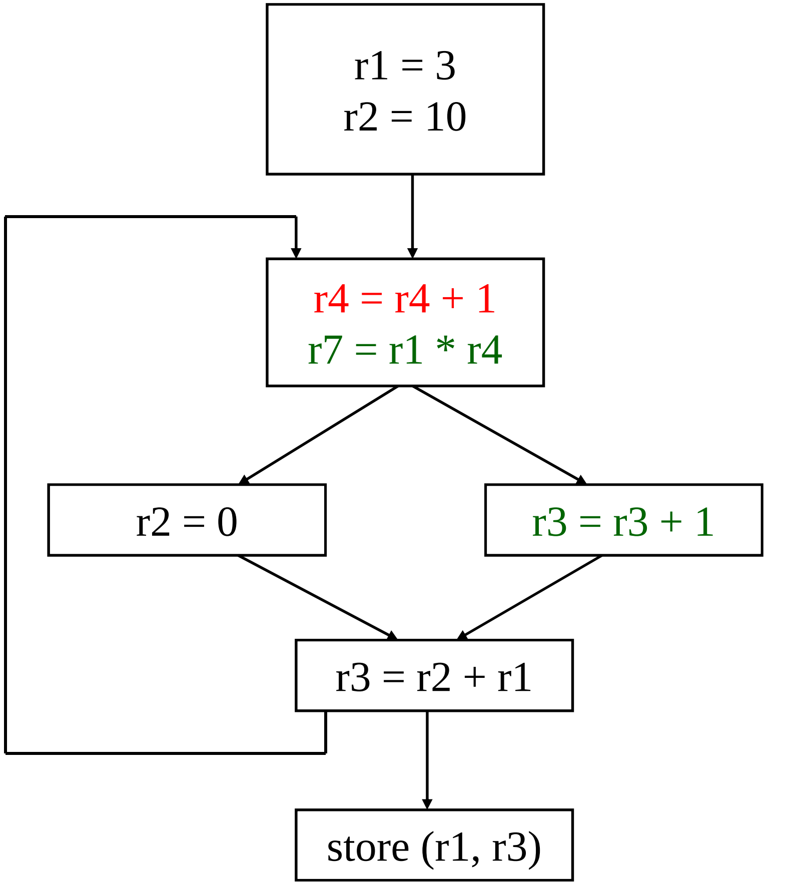
- Convenient way to access/use reaching defs info
- Def-Use chains
  - Given a def, what are all the possible consumers of the operand produced
  - Maybe consumer
- Use-Def chains
  - Given a use, what are all the possible producers of the operand consumed
  - Maybe producer

Dead Code Elimination

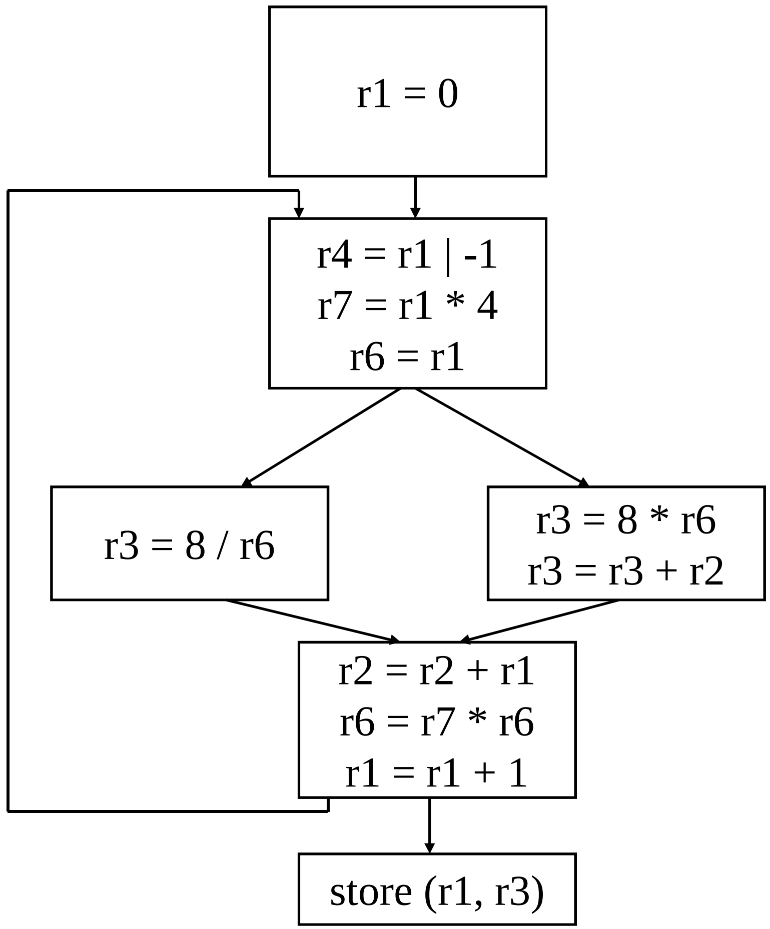
- Remove instructions producing never used results
- Previous optimizations help cause them
  - Constant combining
  - Constant folding
  - Operation folding
  - Copy propagation
  - Common subexpression elimination
  - Agebratic simplification

Dead Code Elimination

- Remove any operation who's result is never consumed
- Rules
  - X can be deleted
    - no stores or branches
  - dest(X) not used in BB
  - dest(X) not in Out(BB)
- This misses some dead code!!
  - Especially in loops
- Better code removal
  - Critical operation
    - store or branch operation
    - Any operation that does not directly or indirectly feed a critical operation is dead
  - Compute use-def chains
  - Trace use-def chains backwards from critical operations
  - Any op not visited is dead



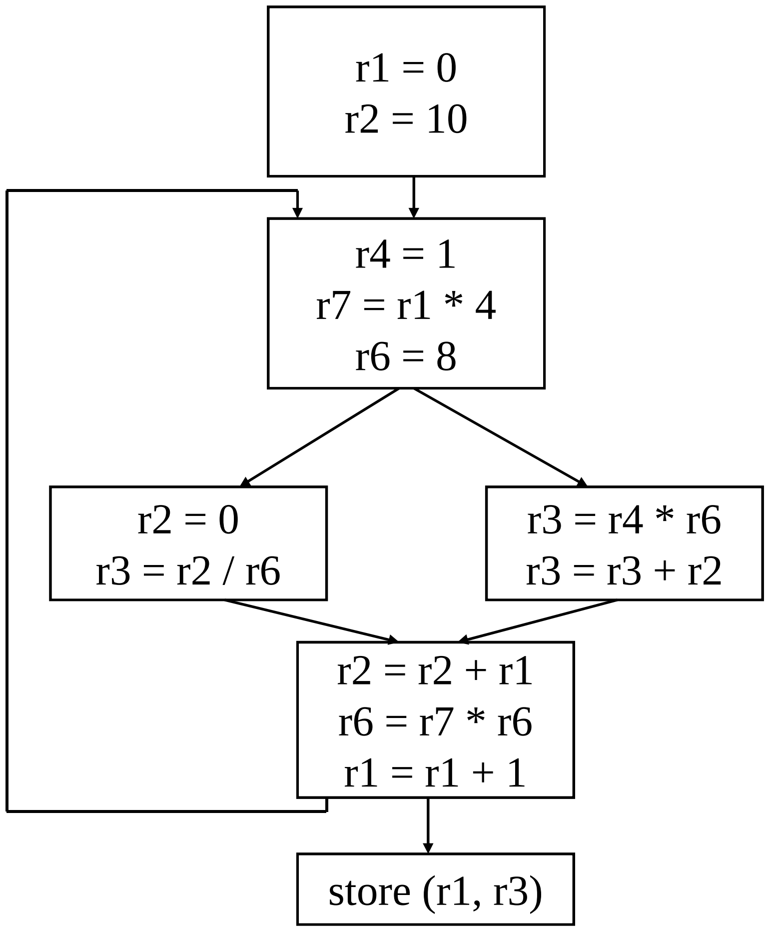
Class Problem



Optimize this applying

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

Class Problem



<u>Optimize this applying</u>

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

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

Back Substitution

- Generation of expressions by compiler front-ends is very sequential
  - Account for operator precedence
  - Apply left-to-right within same precedence
- Back substitution
  - Create larger expressions
    - Iteratively substitute RHS expression for LHS variable
  - Note – may correspond to multiple source statements
  - Enable subsequent opts
- Optimization
  - Re-compute expression in a more favorable manner

y = a + b + c – d + e – f;

r9 = r1 + r2
r10 = r9 + r3
r11 = r10 - r4
r12 = r11 + r5
r13 = r12 – r6

Subs r12:
r13 = r11 + r5 – r6
Subs r11:
r13 = r10 – r4 + r5 – r6
Subs r10
r13 = r9 + r3 – r4 + r5 – r6
Subs r9
r13 = r1 + r2 + r3 – r4 + r5 – r6

Tree Height Reduction

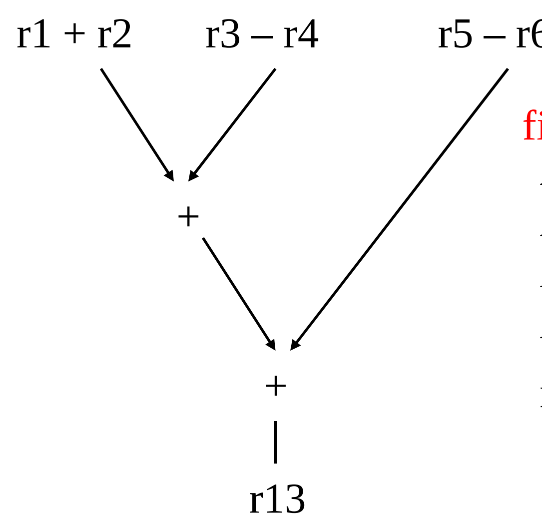
- Re-compute expression as a balanced binary tree
  - Obey precedence rules
  - Essentially re-parenthesize
- Effects
  - Height reduced (n terms)
    - n-1 (assuming unit latency)
    - ceil(log2(n))
  - Number of operations remains constant
  - Cost
    - Temporary registers “live” longer
  - Watch out for
    - Always ok for integer arithmetic
    - Floating-point – may not be!!

original:

r9 = r1 + r2
r10 = r9 + r3
r11 = r10 - r4
r12 = r11 + r5
r13 = r12 – r6

after back subs:

r13 = r1 + r2 + r3 – r4 + r5 – r6



final code:

t1 = r1 + r2
t2 = r3 – r4
t3 = r5 – r6
t4 = t1 + t2
r13 = t4 + t3

Fancier Tree Height Reduction

- Take advantage of literals
  - Reassociate to maximize opportunities for combining literals at compile time
  - Reduces amount of computation

after back subs:

r13 = r1 + 4 + r2 - 3 + r3 - 6

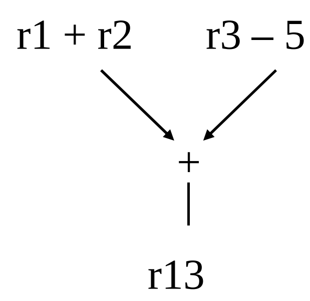
reassociate:

r13 = r1 + r2 + r3 + (4 - 3 – 6)

simplify:

r13 = r1 + r2 + r3 - 5

balance:

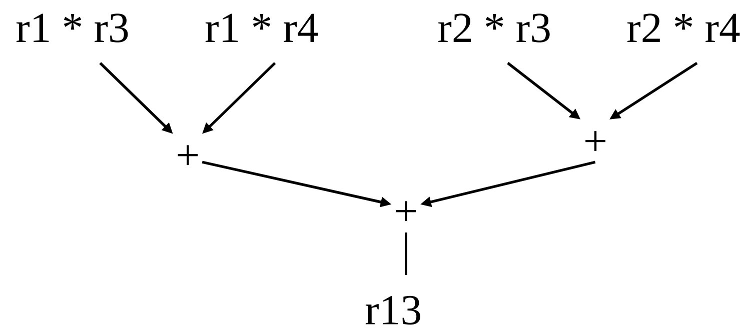


Fancier Tree Height Reduction (2)

- Apply distributive property
  - a(b+c) = ab + bc
  - Or the reverse
  - Danger
    - Generate more operations
    - Lots of possibilities
- Account for latency in balancing process
  - Want latencies balanced, not the number of operations
  - multiply = 3, add = 1
- Account for operand arrival time
  - Delay use of late arriving operands

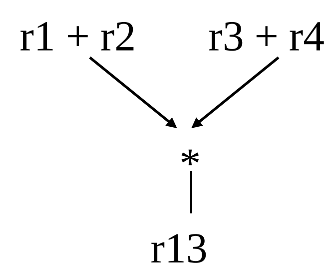
after back subs:

r13 = r1*r3 + r1*r4 + r2*r3 + r2*r4



or better yet:

r13 = (r1 + r2) * (r3 + r4)



Class Problem

Assume: + = 1, * = 3

operand	r1	r2	r3	r4	r5	r6
arrival times	0	0	0	1	2	0

```
r10 = r1 * r2
r11 = r10 + r3
r12 = r11 + r4
r13 = r12 – r5
r14 = r13 + r6
```

Back susbstitute

Re-express in tree-height reduced form

Account for latency and arrival times