Dataflow IV: Loop Optimizations, Exam 2 Review

EECS 483 – Lecture 26
University of Michigan
Wednesday, December 6, 2006

Announcements and Reading

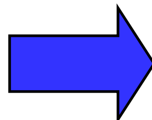
- Schedule
  - Wednes 12/6 – Optimizations, Exam 2 review
  - Mon 12/11 – Exam 2 in class
  - Wednes 12/13 – No class
- Extra office hours
  - Thurs: 4:30 – 5:30 (4633 CSE)
- Project 3 – 2 options
  - Due 12/13, Demos 12/14 (5% bonus on P3 if you turn it in early)
  - Due 12/20, Demos 12/21

Class Problem From Last Time

Optimize this applying
1. constant prop
2. constant folding
3. strength reduction
4. dead code elim
5. forward copy prop
6. backward copy prop
7. CSE

```
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)
```

Const prop
Dead code elim



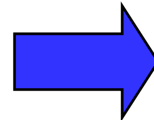
```
r2 = r3 * 4
r8 = r2 + 0
r9 = r3
r7 = load(r2)
r5 = r9 * 4
r3 = load(r2)

store (r8, r7)
r11 = r2
r12 = load(r11)
store(r12, r3)
```

Class Problem From Last Time (cont)

Optimize this applying
1. constant prop
2. constant folding
3. strength reduction
4. dead code elim
5. forward copy prop
6. backward copy prop
7. CSE

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

Str reduction
Const folding
Forw copy prop
Dead code elim



```
r2 = r3 << 2


r7 = load(r2)
r5 = r3 << 2
r3 = load(r2)
r10 = r3 >> 4
store (r2, r7)

r12 = load(r2)
store(r12, r3)
```

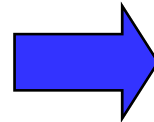
Class Problem From Last Time (cont)

Optimize this applying

1. constant prop
2. constant folding
3. strength reduction
4. dead code elim
5. forward copy prop
6. backward copy prop
7. CSE

```
r2 = r3 << 2


r7 = load(r2)
r5 = r3 << 2
r3 = load(r2)
r10 = r3 >> 4
store (r2, r7)

r12 = load(r2)
store(r12, r3)
```

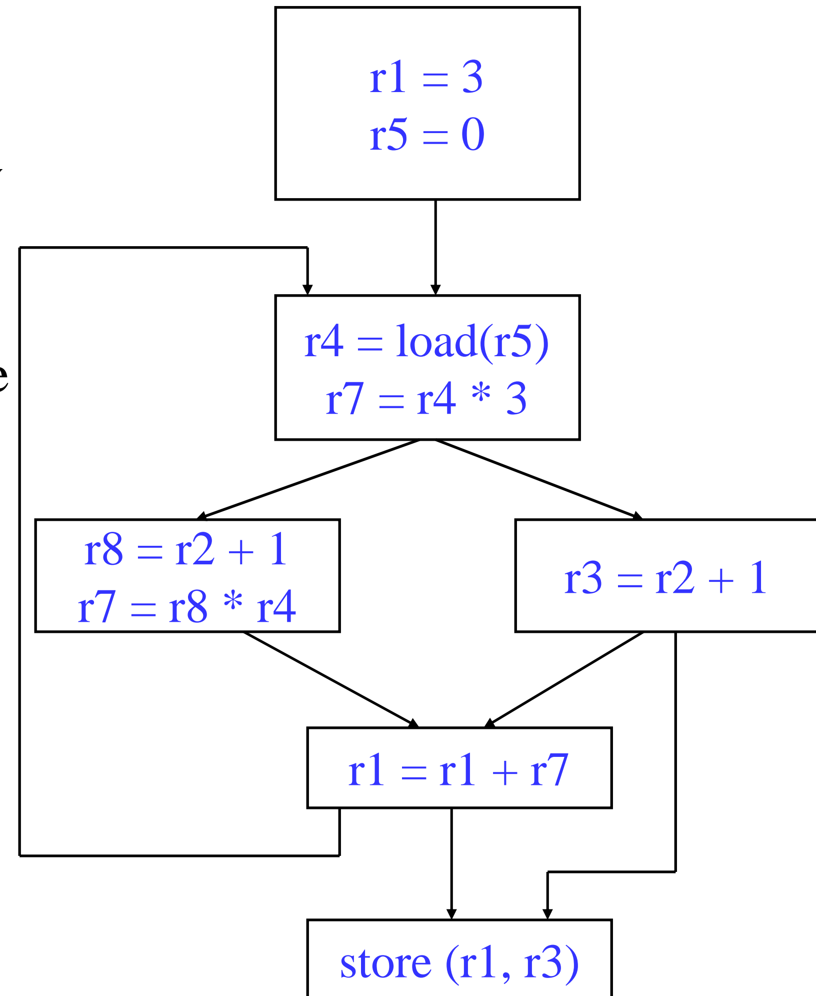
CSE
Forw copy prop
Dead code elim



```
r2 = r3 << 2


r7 = load(r2)



store (r2, r7)

r12 = load(r2)
store(r12, r7)
```

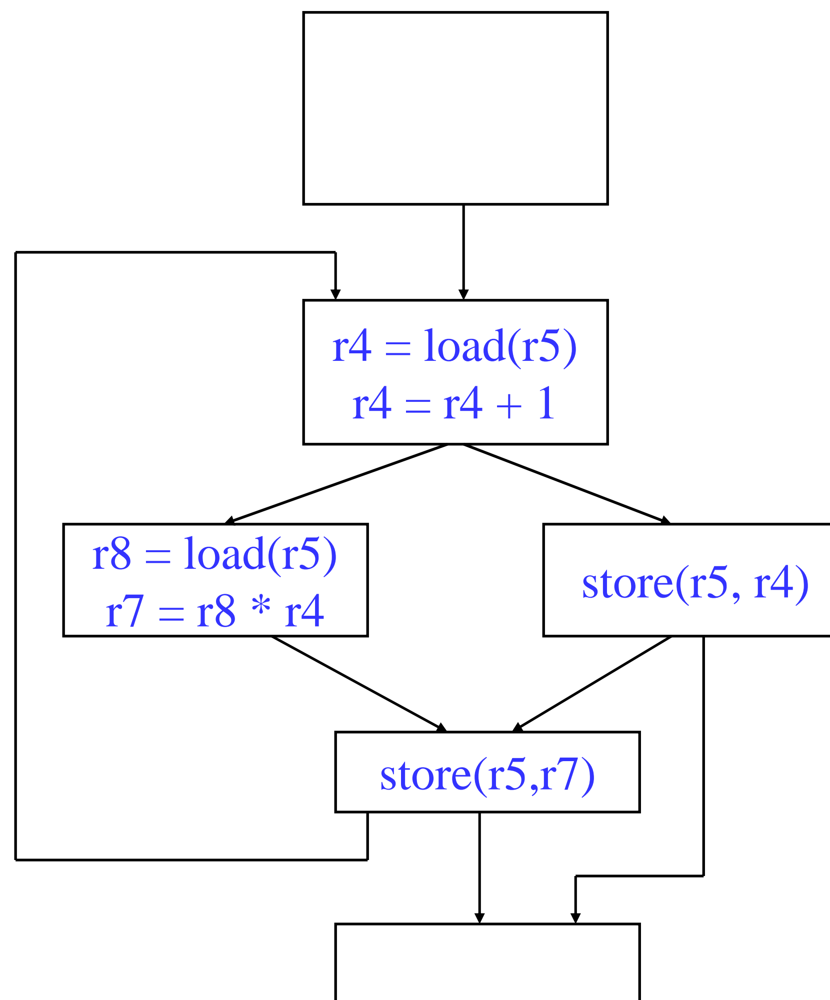
Loop Invariant Code Motion (LICM)

- 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



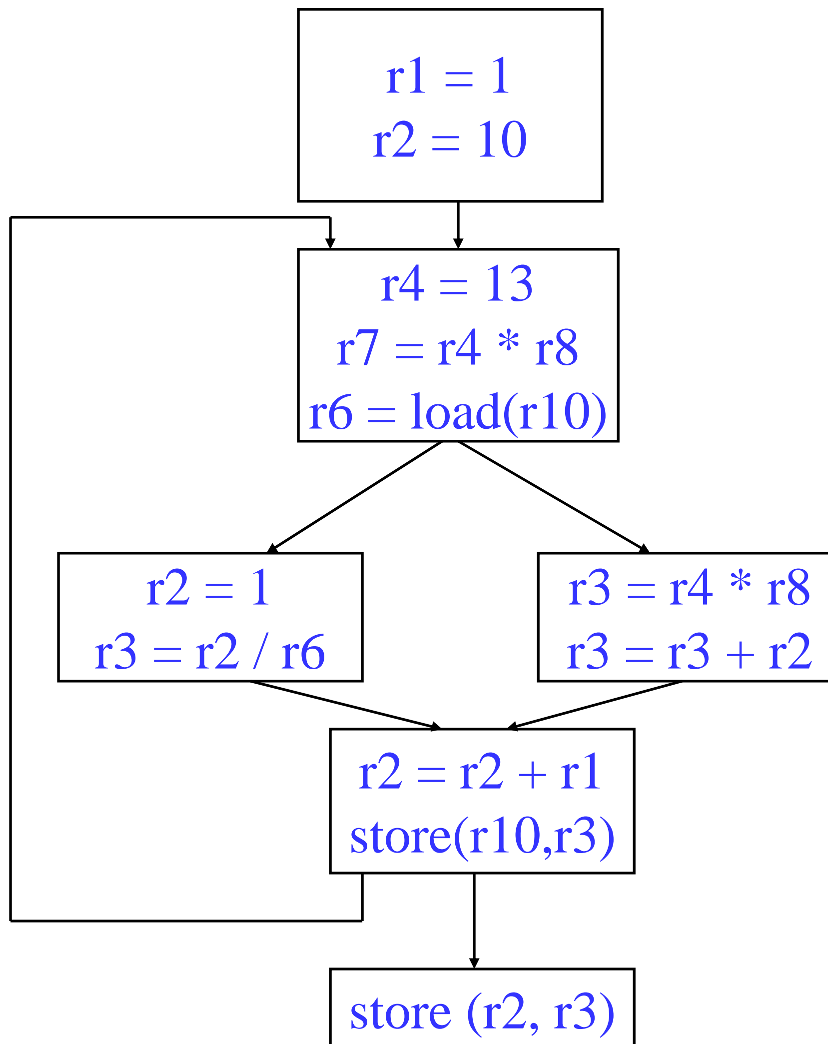
Global Variable Migration

- 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



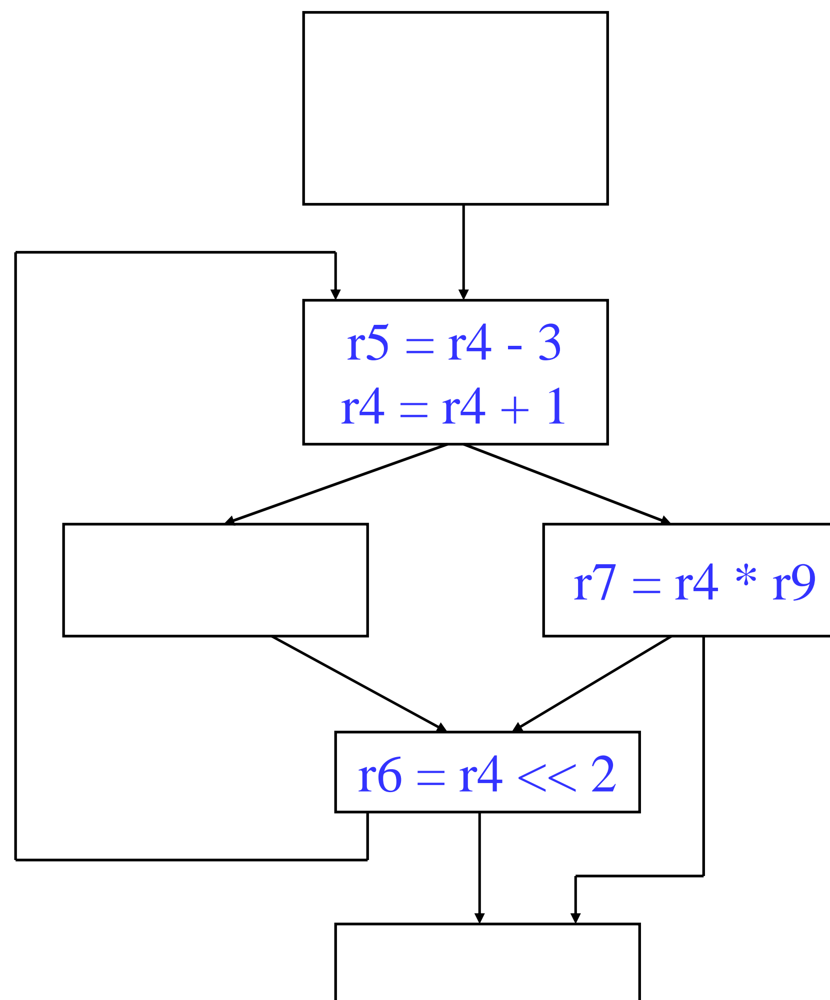
Class Problem

Apply global variable migration to the following program segment



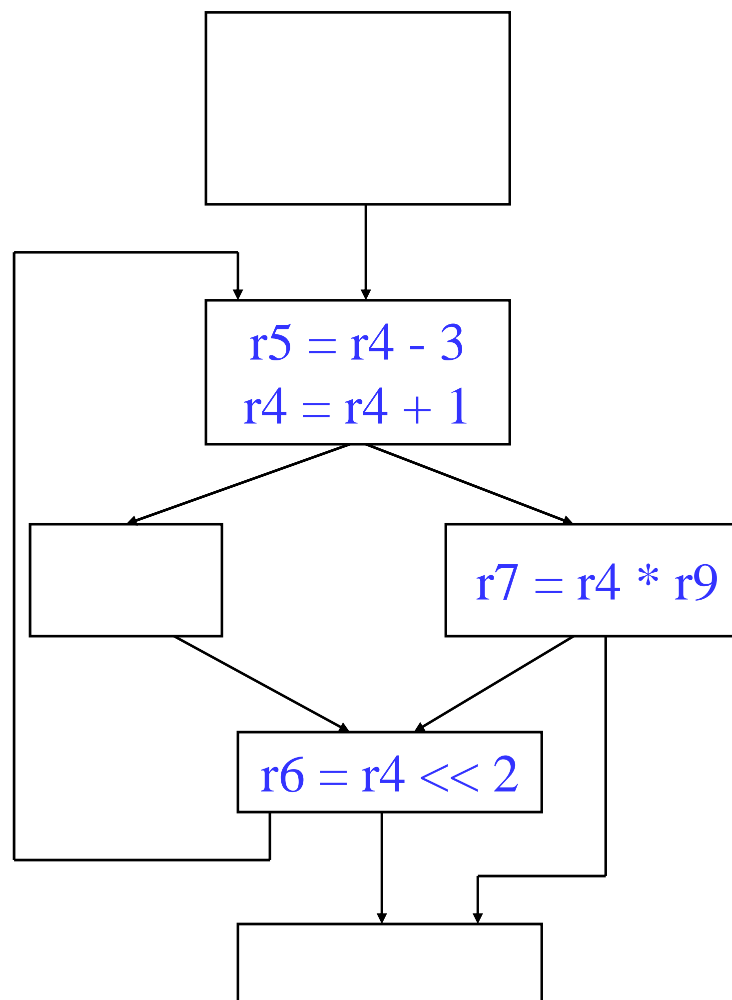
Induction Variable Strength Reduction

- Create basic induction variables from derived induction variables
- Rules
  - » X is a *, <<, + 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



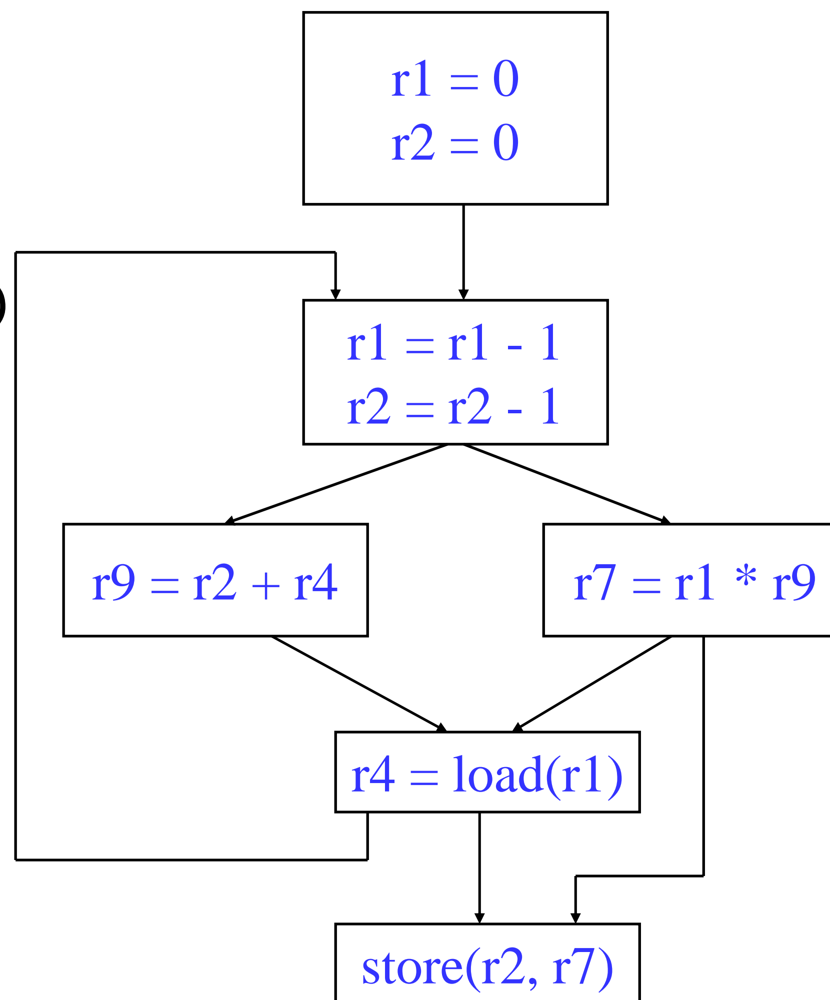
Induction Variable Strength Reduction (2)

- Transformation
  - Insert the following into the bottom of 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



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



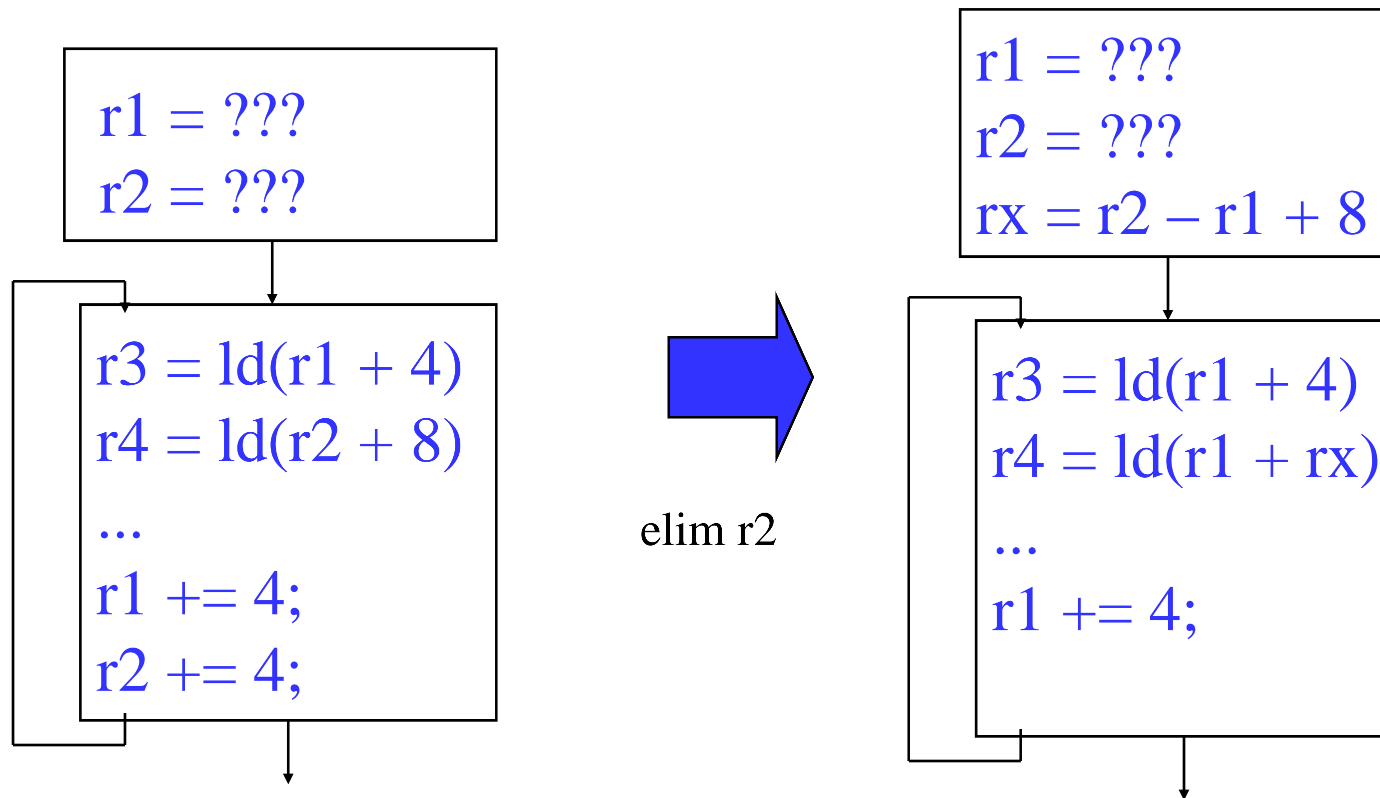
Induction Variable Elimination (2)

- 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

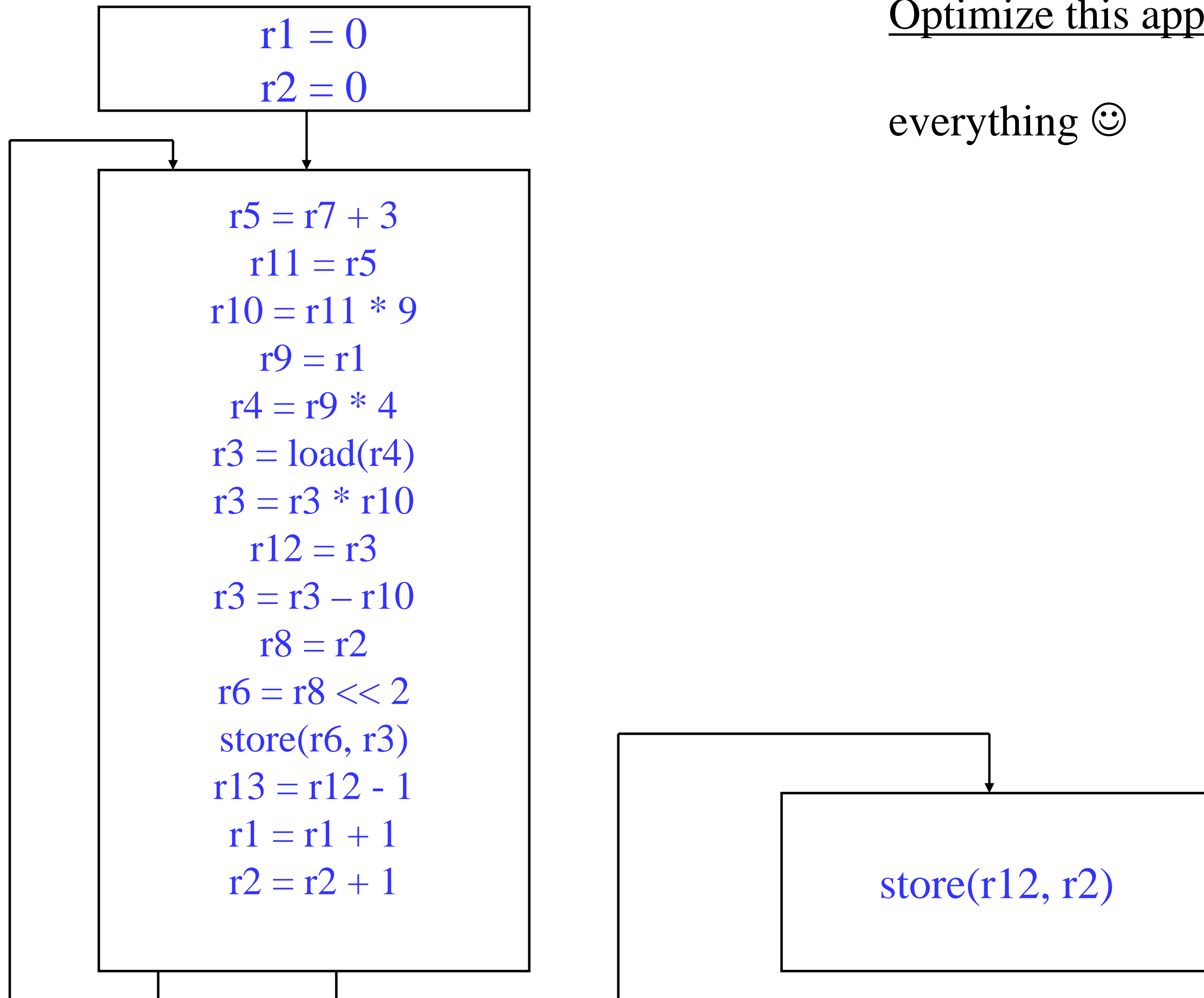
For the ind var you are eliminating, look at each non-increment use, need to regenerate the same sequence of values as before. If you can do that w/o adding any ops to the loop body, the apply xform



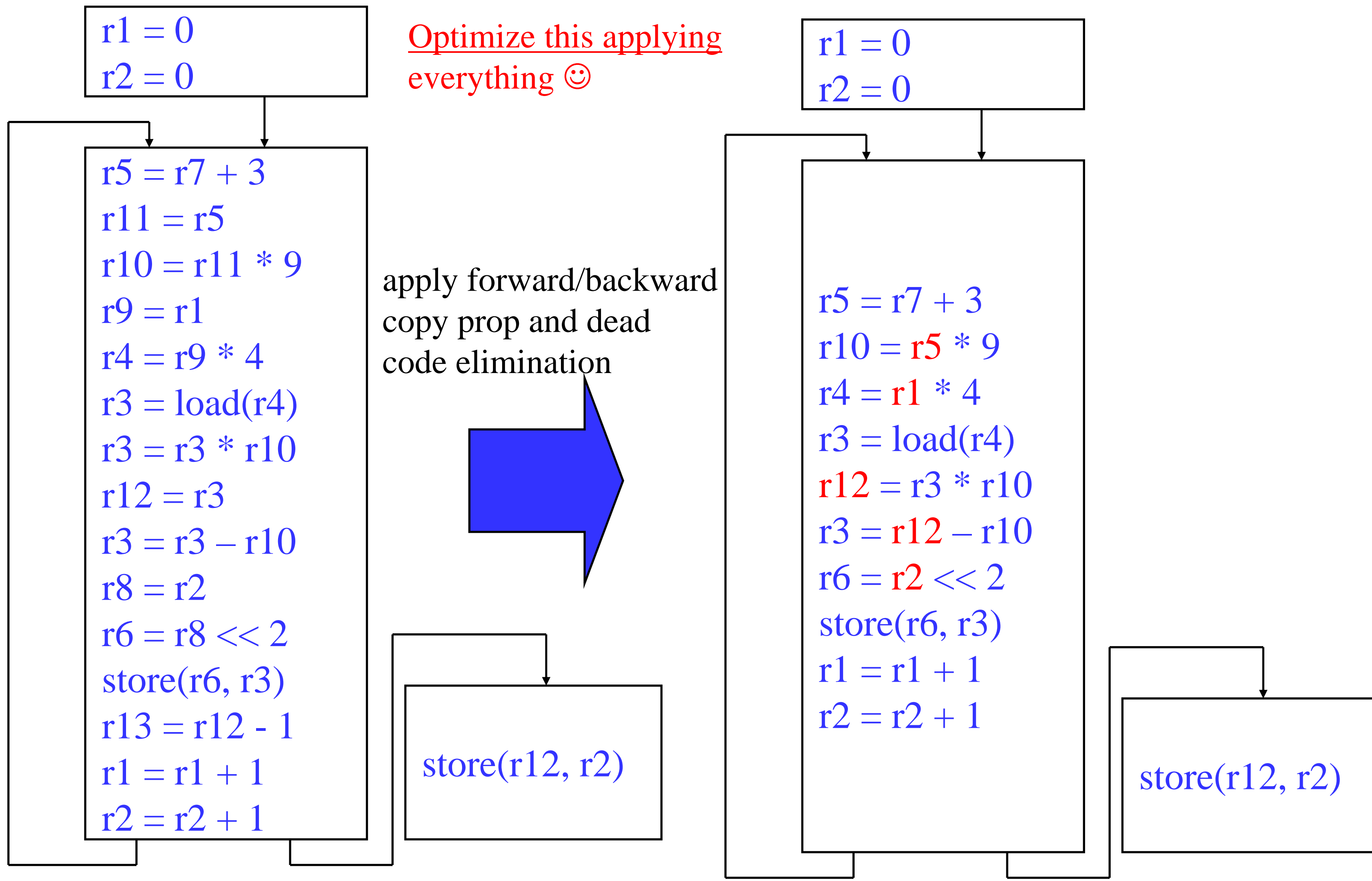
Class Problem

Optimize this applying

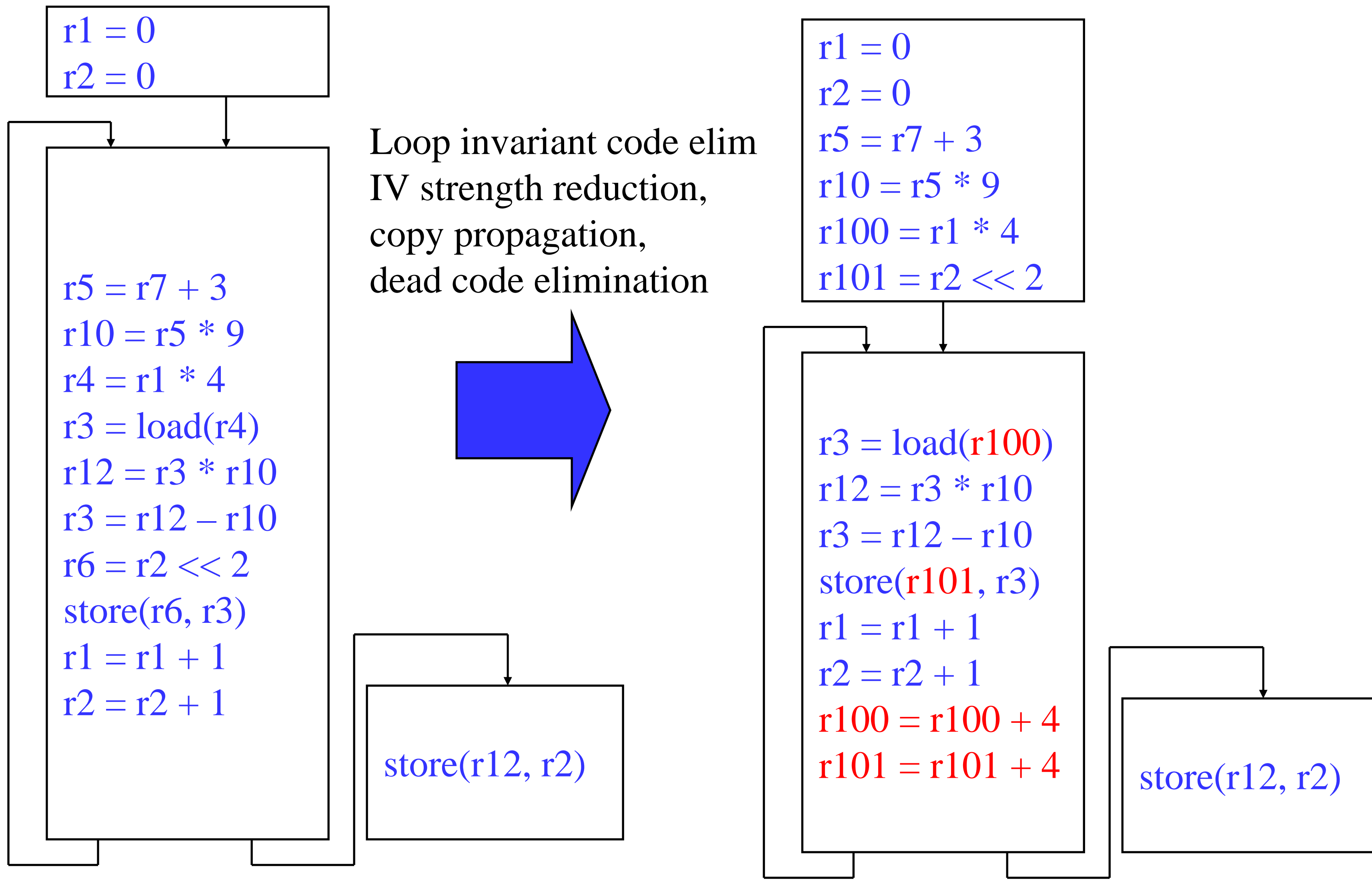
everything ☺



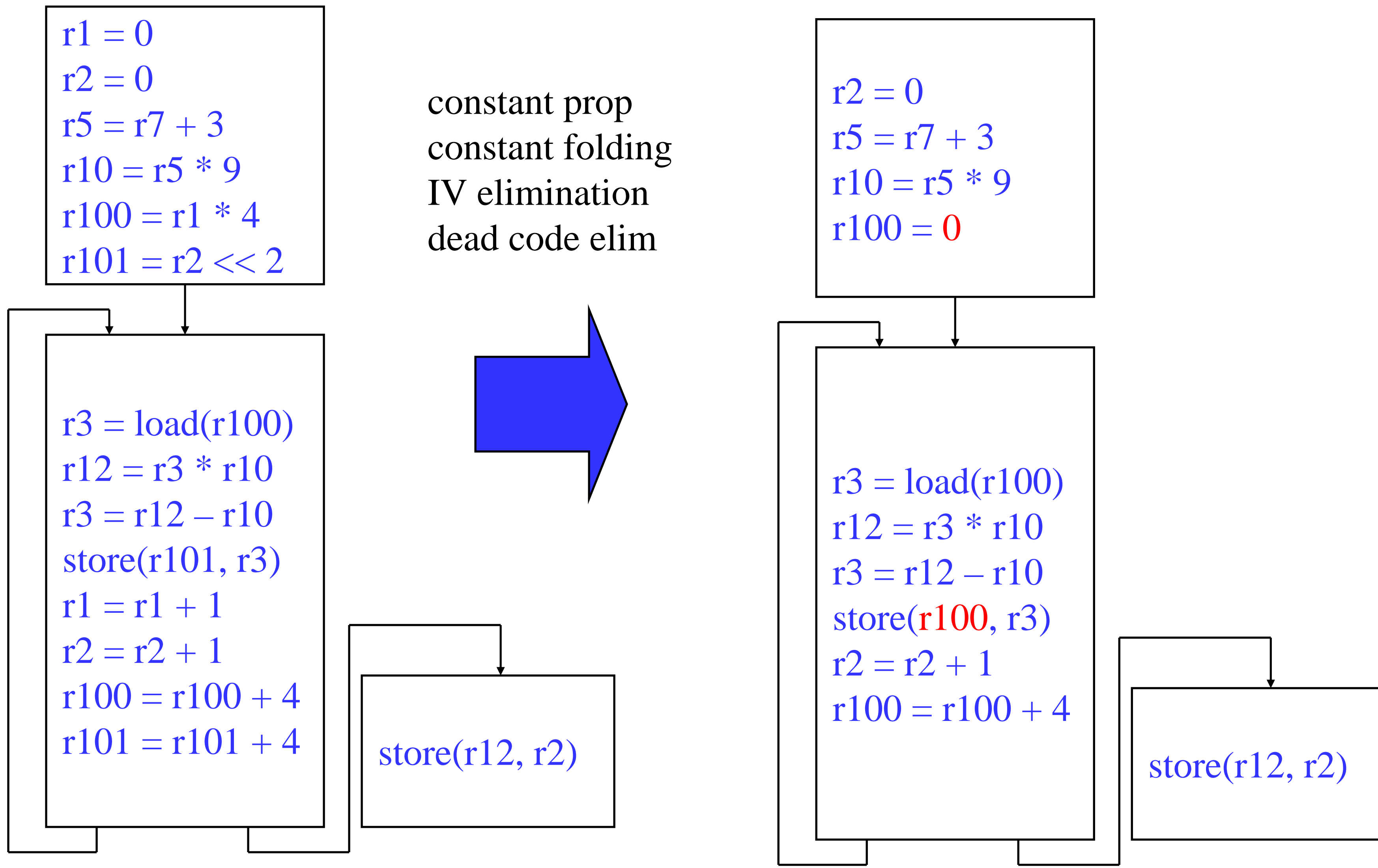
Class Problem – Answer (1)



Class Problem – Answer (2)



Class Problem – Answer (3)



Last Topic – Register Allocation

- Through optimization, assume an infinite number of virtual registers
  - » Now, must allocate these infinite virtual registers to a limited supply of hardware registers
  - » Want most frequently accessed variables in registers
    - Speed, registers much faster than memory
    - Direct access as an operand
  - » Any VR that cannot be mapped into a physical register is said to be spilled
  - » If there are not enough physical registers, which virtual registers get spilled?

Questions to Answer

- What is the minimum number of registers needed to avoid spilling?
- Given n registers, is spilling necessary?
- Find an assignment of virtual registers to physical registers
- If there are not enough physical registers, which virtual registers get spilled?

For those interested in how this works, see supplementary lecture.
You are not responsible for register allocation on the exam.

Exam 2 Review

Logistics

- When, Where:
  - Monday, Dec 11, 10:40am – 12:40pm
  - Room: 1006 Dow
- Type:
  - Open book/note
- What to bring:
  - Text book, reference books, lecture notes
  - Pencils
  - No laptops or cell phones

Topics Covered (1)

- Intermediate representation
  - Translating high-level constructs to assembly
  - Storage management
    - Stack frame
    - Data layout
- Control flow analysis
  - CFGs, dominator / post dominator analysis, immediate dom/pdom
  - Loop detection, trip count, induction variables

Topics Covered (2)

- Dataflow analysis
  - GEN, KILL, IN, OUT, up/down, all/any paths
  - Liveness, reaching defs, DU, available exprs, ...
- Optimization
  - Control
    - Loop unrolling, acyclic optimizations (branch to branch, unreachable code elim, etc.)
  - Data
    - Local, global, loop optis (how they work, how to apply them to examples, formulate new optis)

Not Covered

- This is NOT a cumulative test
  - Exam 1 → frontend, This exam → backend
    - No parsing or type analysis
  - But some earlier topics that carry over you will be expected to be familiar with (ie AST)
- No MIRV/Openimpact specific stuff
- No SSA Form, No register allocation
  - These are covered on the F04 exam 2

Textbook

- What have we covered: Nominally Chs 7-10
  - 2nd half of class: more loosely followed book
- Things you should know / can ignore
  - Ch 7 – 7.2,7.3 is what we covered, ignore rest
  - Ch 8 – 8.1-8.5 is what we covered, but not that closely
  - Ch 9 – we covered 9.3, 9.4, 9.7, 9.9
    - Ignore all code generation from DAG stuff
  - Ch 10 – This is the most important of all the book chapters
    - Ignore: all of 10.8, interval graph stuff in 10.9, all of 10.10, 10.12, 10.13

Exam Format

- Similar to Exam 1
- Short answer: ~50%
  - Explain something
  - Short problems to work out
- Longer design problems: ~50%
  - E.g., compute reaching defn gen/kill/in/out sets
- Range of questions
  - Simple – Were you conscience in class?
  - Grind it out – Can you solve problems
  - Challenging – How well do you really understand things

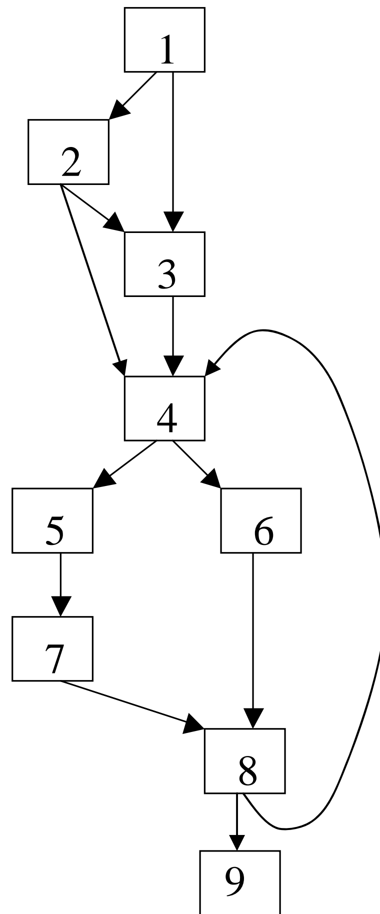
Intermediate Code

Convert the following C code segment into assembly format using the **do-while** style for translation. Assume that *x, y* are integers and that *A* is an array of integers. Note that you should make no assumptions about the value of *j*. You may use pseudo-assembly as was done in class, i.e., operators such as +, -, *, <, >, load, store, branch. Also, use virtual registers with the following mapping of register numbers to variables: r1 = i, r2 = j, r3 = x, r4 = y, r5 = starting address of A, r6 and above for temporaries.

```
for (i=0; i<j; i++) {
    x = A[i];
    if ((x ==0) || (x > 10))
        y++
}
```

Control Flow Analysis

Compute the post dominators (PDOM set) for each basic block in the following control flow graph.



Control Flow Optimization

Assuming that you wanted to apply the most aggressive form of loop unrolling to unroll the loop twice, which technique could be applied to the following loop segment? Briefly explain.

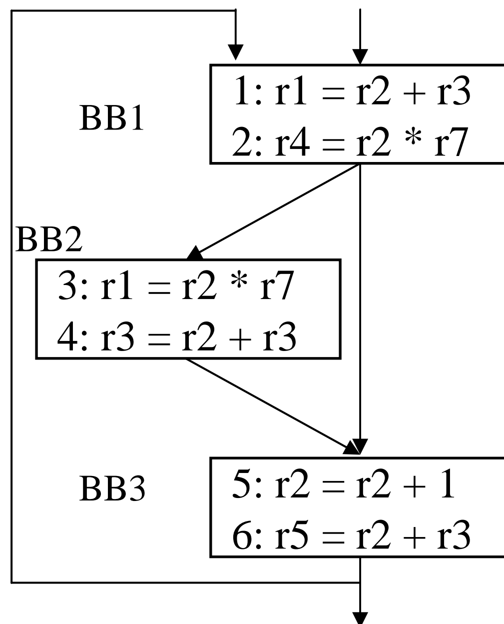
```
x = *ptr;
for (j = 0; j<x; j++) {
    if (ptr == NULL)   continue;
    ptr = ptr->next;
}
```

Dataflow Analysis (simple)

A. In one sentence, what's the primary difference between a reaching definition and an available definition? Give a small example to illustrate a definition that is reaching but not available.

Dataflow Analysis

Compute the available expression GEN, KILL, IN, OUT for each basic block in the following code segment.



Classical Optimization (simple)

Explain why induction variable strength reduction is only applicable with the opcodes +, -, *, and << applied to a basic induction variable. In other words, why is it limited to these opcodes?

Classical Optimization

Consider applying loop invariant code motion (LICM) to the following loop segment. In applying the optimization, you are not allowed to change any operands nor introduce any temporaries. For each instruction 1-5, state whether it can be removed from the loop via LICM. If the answer is no, provide **one** reason why it cannot be hoisted.