EECS 583 – Class 9
Classic and ILP Optimization

University of Michigan

October 5, 2011

Announcements & Reading Material

- Look on Phorum for help with HW #2
  - Daya's post on adding/deleting instructions
  - Splitting a BB
- Today's class
  - "Compiler Code Transformations for Superscalar-Based High-Performance Systems," Scott Mahlke, William Chen, John Gyllenhaal, Wen-mei Hwu, Pohua Chang, and Tokuzo Kiyohara, *Proceedings of Supercomputing '92*, Nov. 1992, pp. 808-817
- Next class (instruction scheduling)
  - "Machine Description Driven Compilers for EPIC Processors", B. Rau, V. Kathail, and S. Aditya, HP Technical Report, HPL-98-40, 1998. (long paper but informative)

HW#2 – Best Times from Last Semester

- Times in seconds for each application

	no spec LICM	with spec LICM
Perf1	.008453	.000095
Perf2	.023796	.000133
Perf3	.012391	.000099
583wc	.002813	.001644

- This translates to: 89x, 178x and 125x and 1.7x speedup
- Note: These generated with an older LLVM
  - But, gives you an idea of what to expect.

Course Project – Time to Start Thinking About This

- Mission statement: Design and implement something "interesting" in a compiler
  - LLVM preferred, but others are fine
  - Groups of 2-3 people (1 or 4 persons is possible in some cases)
  - Extend existing research paper or go out on your own
- Topic areas
  - Automatic parallelization/SIMDization
  - Optimizing for GPUs
  - Targeting VLIW processors
  - Memory system optimization
  - Reliability
  - Energy
  - For the adventurous
    - Dynamic optimization
    - Streaming applications
    - Binary optimization

Course Projects – Timetable

- Now
  - Start thinking about potential topics, identify teammates
- Oct 24-28: Project proposals
  - No class that week
  - Daya/I will meet with each group
  - Informal proposal discussed at meeting
  - Written proposal (a paragraph or 2 plus some references) due Oct 31
- Last 2 wks of Nov: Project checkpoint
  - Class as usual
  - Daya/I will again meet with each group
  - Update on your progress, what left to do
- Dec 13-16: Project demos

Sample Project Ideas (in random order)

- Memory system
  - Cache profiler for LLVM IR – miss rates, stride determination
  - Data cache prefetching
  - Data layout for improved cache behavior
- Optimizing for GPUs
  - Dumb OpenCL/CUDA → smart OpenCL/CUDA – selection of threads/blocks and managing on-chip memory
  - Reducing uncoalesced memory accesses – measurement of uncoalesced accesses, code restructuring to reduce these
  - Matlab → CUDA/OpenCL
  - StreamIt to CUDA/OpenCL (single or multiple GPUs)
- Reliability
  - AVF profiling, reducing vulnerability to soft errors
  - Coarse-grain code duplication for soft error protection

More Project Ideas

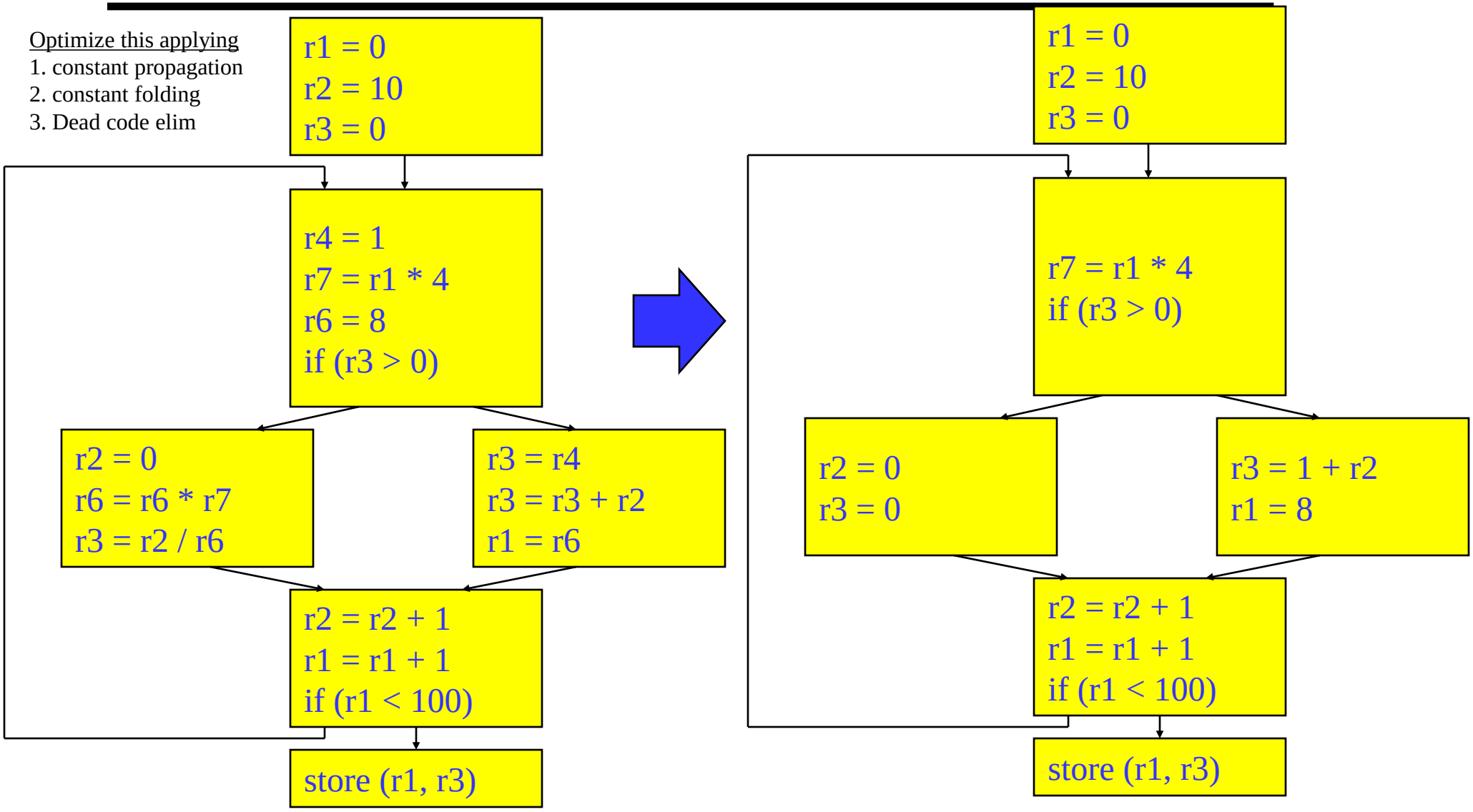
- Parallelization/SIMDization
  - CUDA/OpenCL/Matlab/OpenMP to x86+SSE or Arm+NEON
  - DOALL loop parallelization, dependence breaking transformations
  - Stream parallelism analysis tool - profiling tool to identify maximal program regions the exhibit stream (or DOALL) parallelism
  - Access-execute program decomposition - decompose program into memory access/compute threads
- Dynamic optimization (Dynamo, Dalvik VM)
  - Run-time DOALL loop parallelization
  - Run-time program analysis for security (taint tracking)
  - Run-time classic optimization (if they don't have it already!)
- Profiling tools
  - Distributed control flow/energy profiling (profiling + aggregation)

And More Project Ideas

- VLIW
  - Superblock formation in LLVM
  - Acyclic BB scheduler for LLVM
  - Modulo BB scheduler for LLVM
- Binary optimizer
  - Arm binary to LLVM IR, de-register allocation
  - X86 binary (clean) to LLVM IR
- Energy
  - Minimizing instruction bit flips
- Misc
  - Program distillation - create a subset program with equivalent memory/branch behavior
  - Code sharing analysis for mobile applications

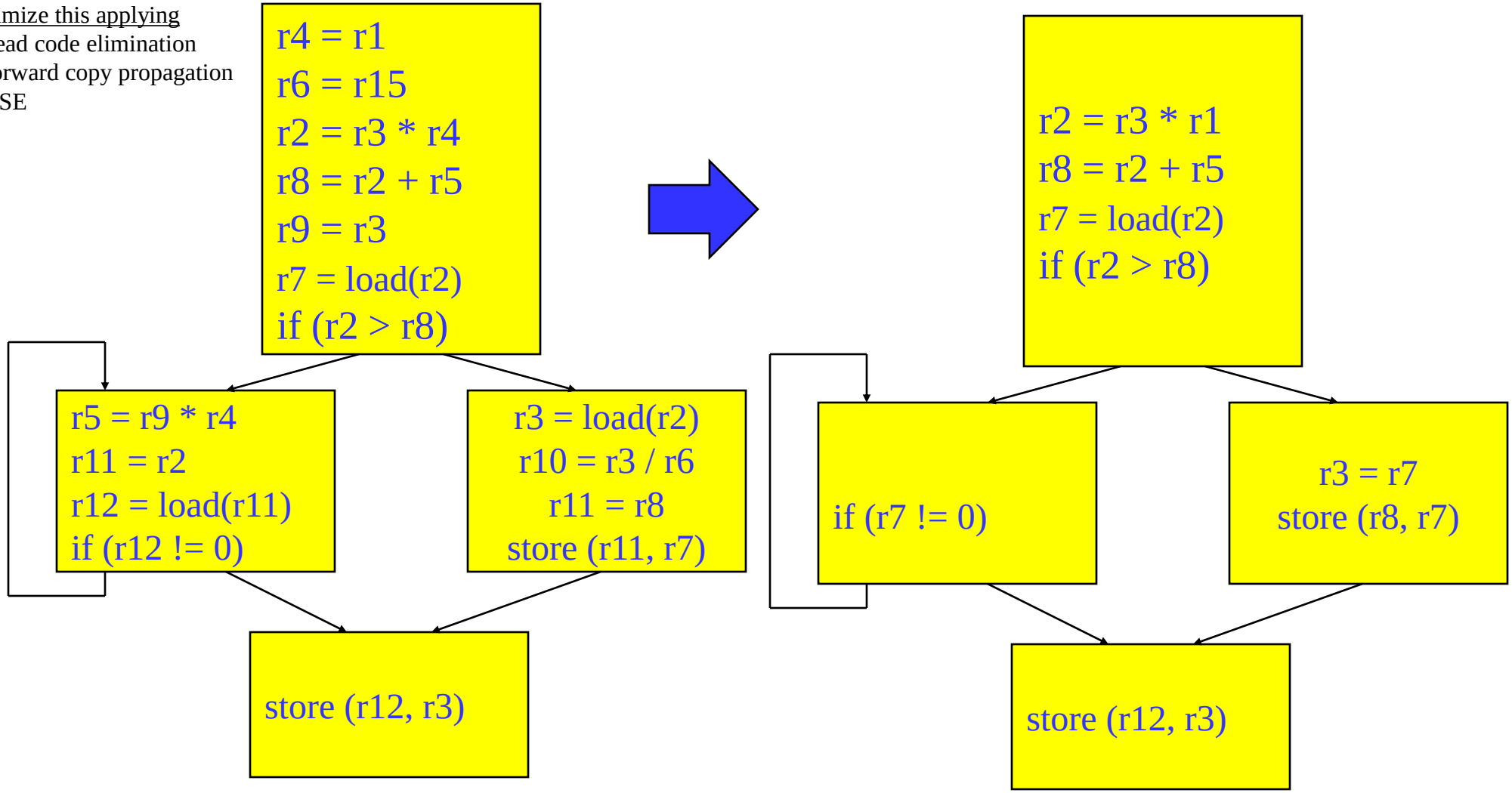
From Last Time: Class Problem Solution

Optimize this applying
1. constant propagation
2. constant folding
3. Dead code elim



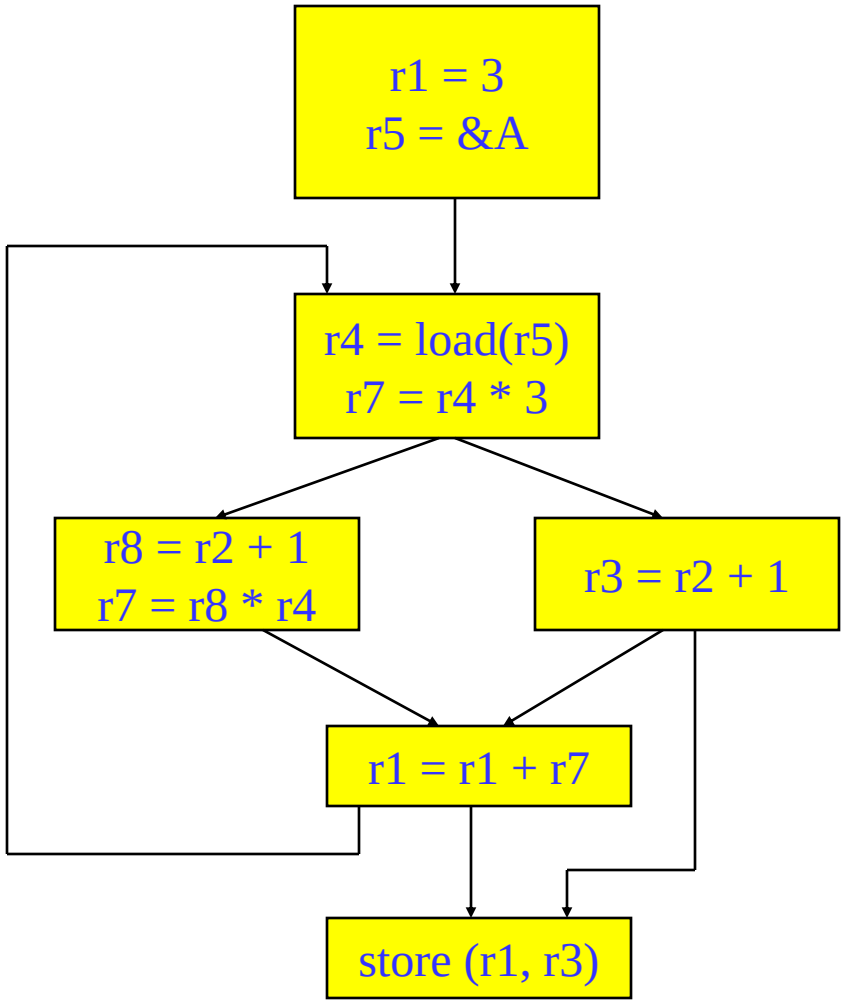
From Last Time: Class Problem Solution

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



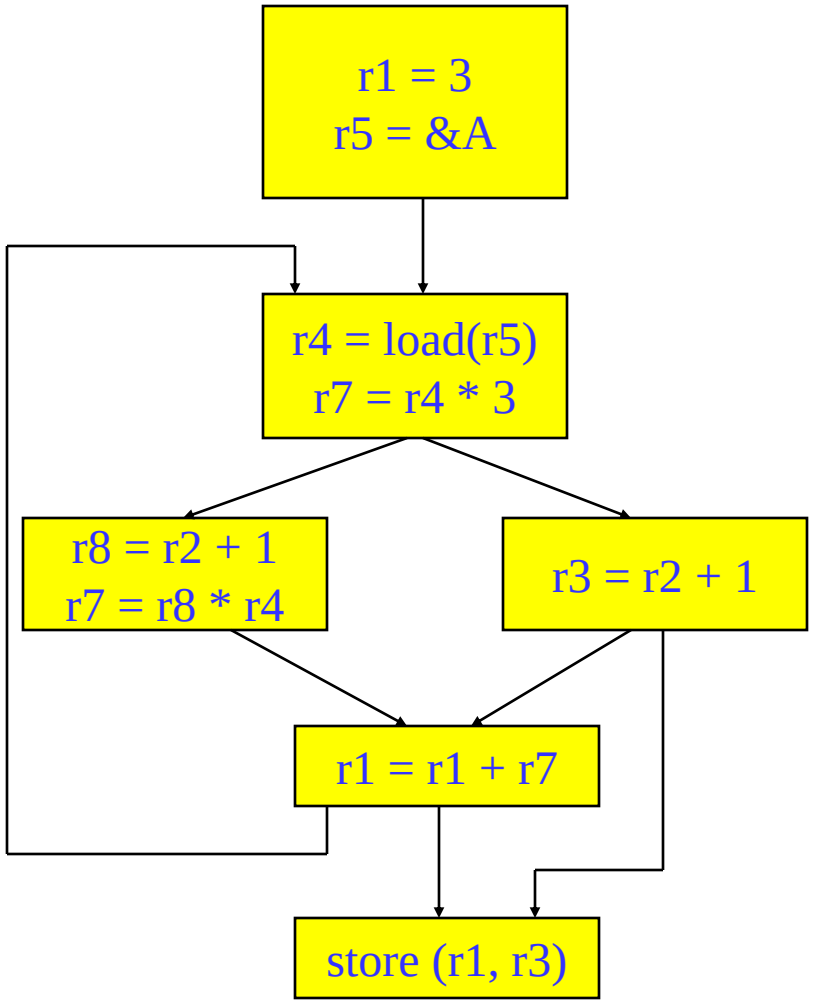
Loop Invariant Code Motion (LICM)

- 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



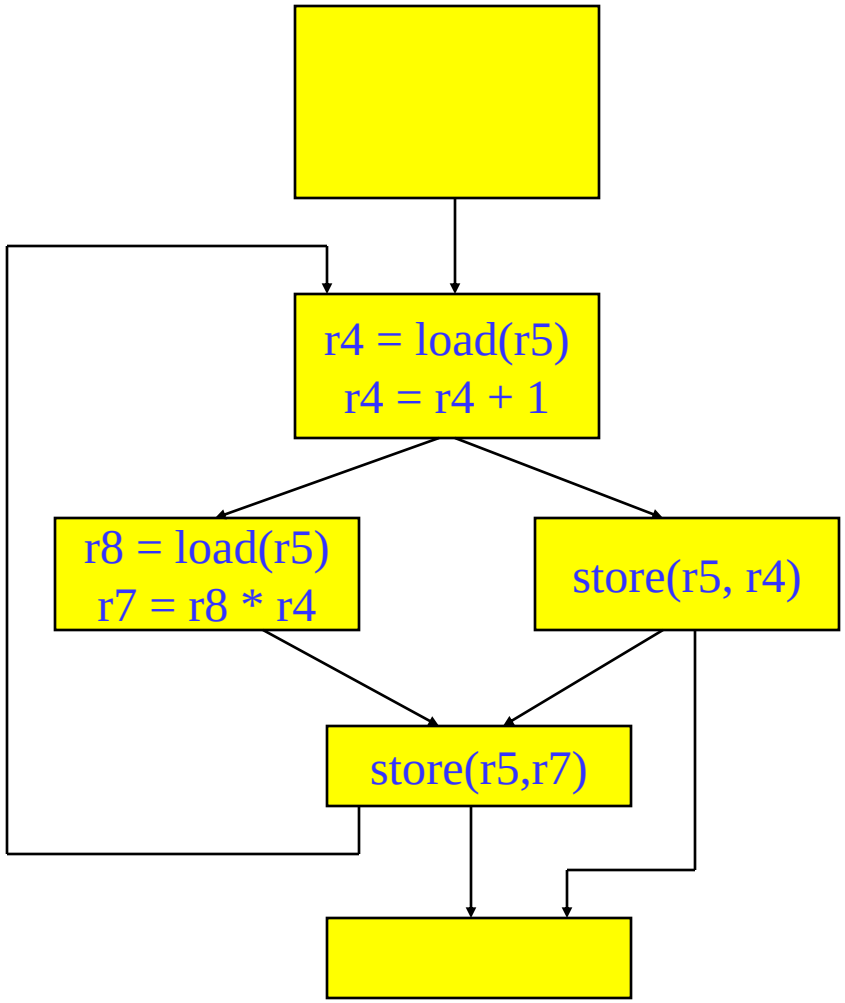
LICM (2)

- 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



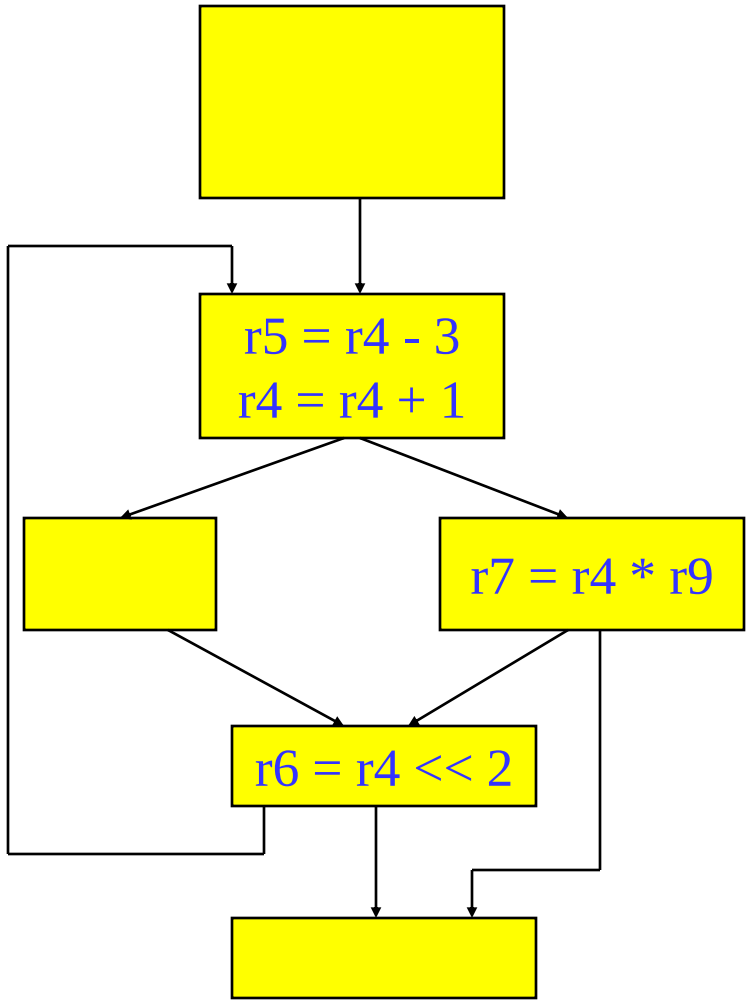
Global Variable Migration

- 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



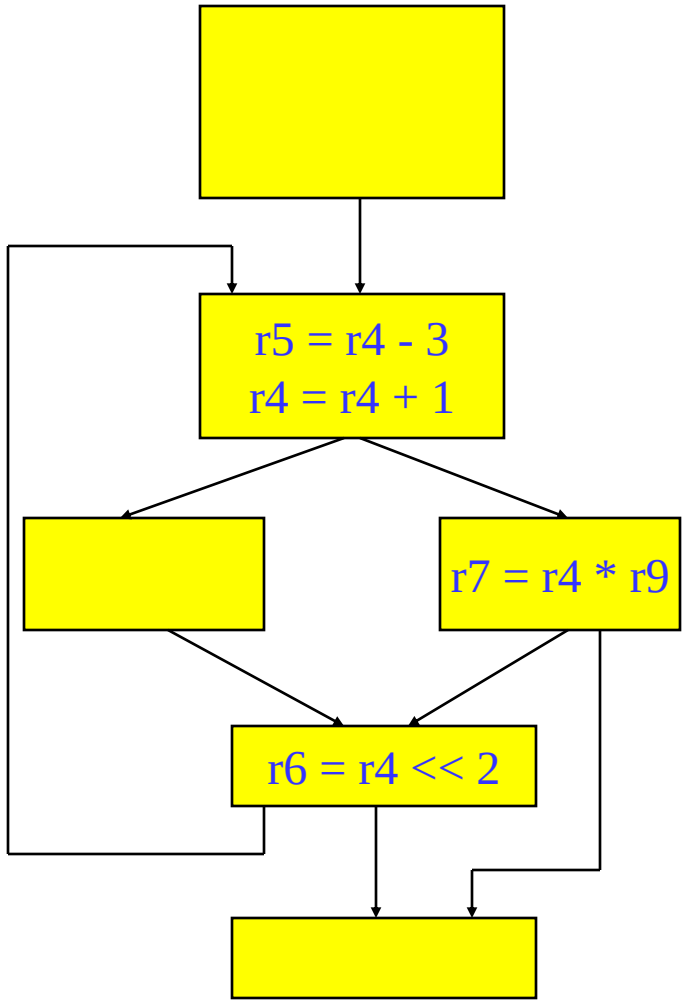
Induction Variable Strength Reduction

- 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

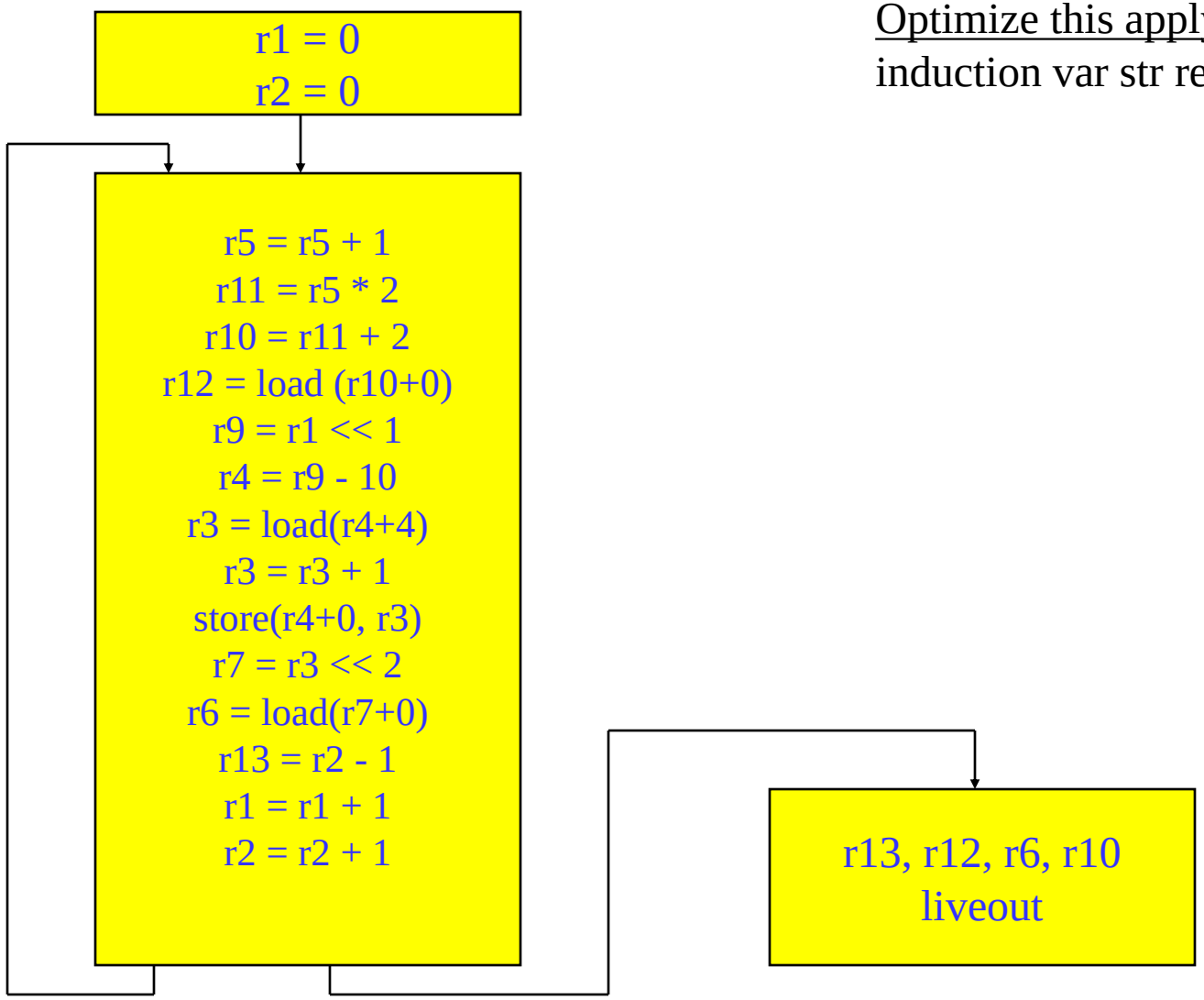


Induction Variable Strength Reduction (2)

- 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



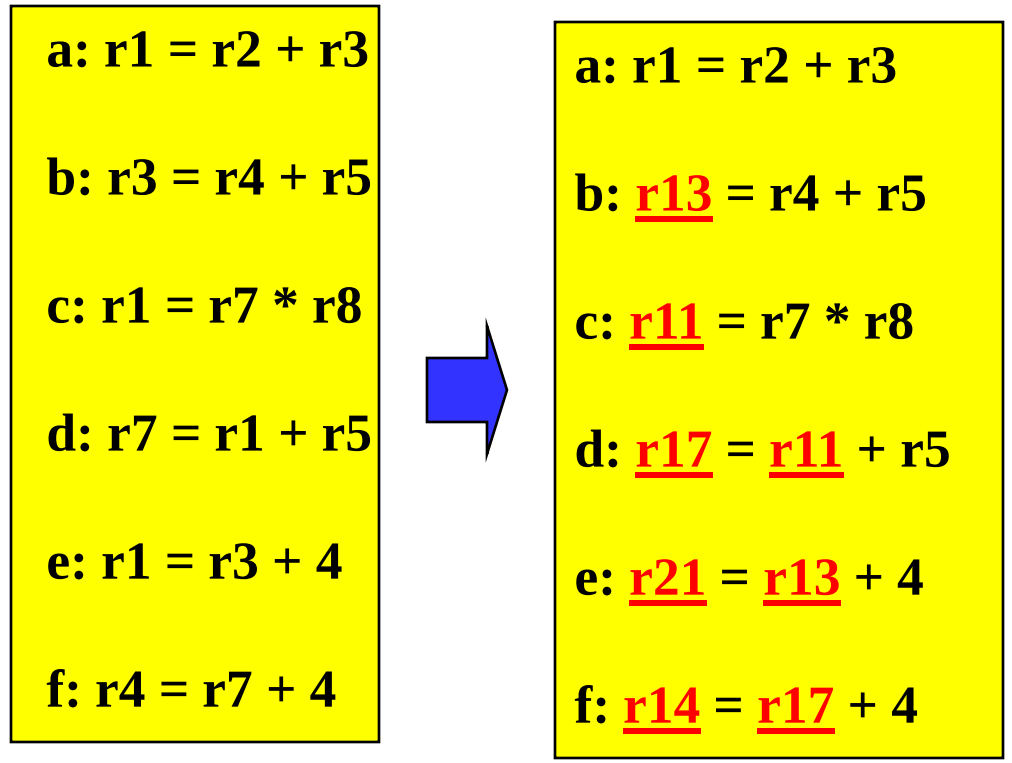
Optimize this applying
induction var str reduction

ILP Optimization

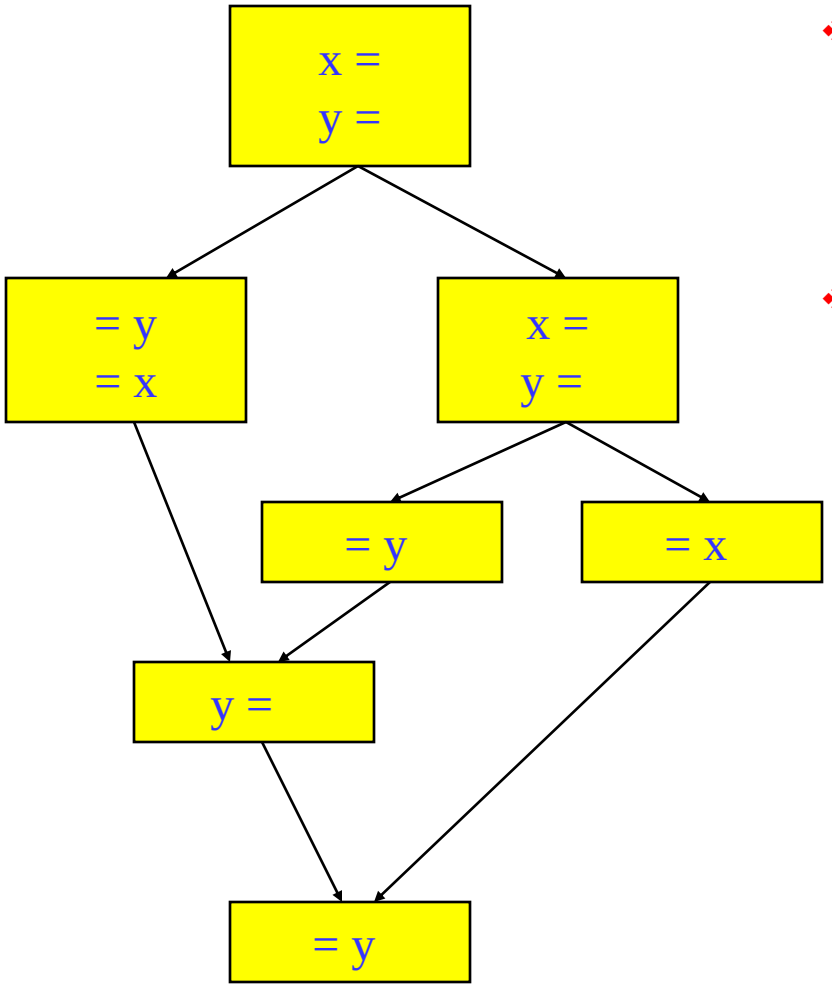
- Traditional optimizations
  - Redundancy elimination
  - Reducing operation count
- ILP (instruction-level parallelism) optimizations
  - Increase the amount of parallelism and the ability to overlap operations
  - Operation count is secondary, often trade parallelism for extra instructions (avoid code explosion)
- ILP increased by breaking dependences
  - True or flow = read after write dependence
  - False or (anti/output) = write after read, write after write

Register Renaming

- Similar goal to SSA construction, but simpler
- Remove dependences caused by variable re-use
  - Re-use of source variables
  - Re-use of temporaries
  - Anti, output dependences
- Create a new variable to hold each unique life time
- Very simple transformation with straight-line code
  - Make each def a unique register
  - Substitute new name into subsequent uses



Global Register Renaming



- Straight-line code strategy does not work
  - A single use may have multiple reaching defs
- Web = Collection of defs/uses which have possible value flow between them
  - Identify webs
    - Take a def, add all uses
    - Take all uses, add all reaching defs
    - Take all defs, add all uses
    - repeat until stable soln
  - Each web renamed if name is the same as another web

Back Substitution

- Generation of expressions by compiler frontends 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 optis
- 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

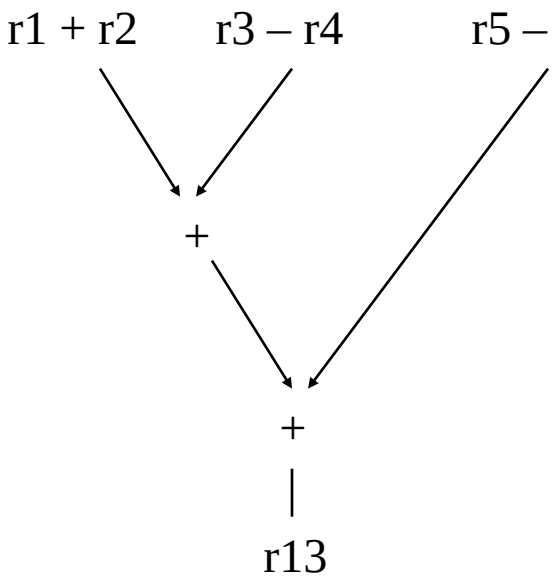
- Re-compute expression as a balanced binary tree
  - » Obey precedence rules
  - » Essentially re-parenthesize
  - » Combine literals if possible
- Effects
  - » Height reduced (n terms)
    - n-1 (assuming unit latency)
    - ceil(log2(n))
  - » Number of operations remains constant
  - » Cost
    - 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
```

Class Problem

Assume: + = 1, * = 3

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

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

Back susbstitute

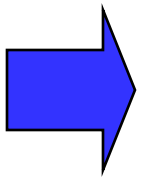
Re-express in tree-height reduced form

Account for latency and arrival times

Optimizing Unrolled Loops

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
       r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

unroll 3 times



```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ----------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter2  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ----------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter3  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

Unroll = replicate loop body n-1 times.

Hope to enable overlap of operation execution from different iterations

Not possible!

Register Renaming on Unrolled Loop

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter2  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter3  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r11 = load(r2)
       r13 = load(r4)
       r15 = r11 * r13
iter2  r6 = r6 + r15
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r21 = load(r2)
       r23 = load(r4)
       r25 = r21 * r23
iter3  r6 = r6 + r25
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

Register Renaming is Not Enough!

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r11 = load(r2)
       r13 = load(r4)
       r15 = r11 * r13
iter2  r6 = r6 + r15
       r2 = r2 + 4
       r4 = r4 + 4
       ------------------
       r21 = load(r2)
       r23 = load(r4)
       r25 = r21 * r23
iter3  r6 = r6 + r25
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

- Still not much overlap possible
- Problems
  - r2, r4, r6 sequentialize the iterations
  - Need to rename these
- 2 specialized renaming optis
  - Accumulator variable expansion (r6)
  - Induction variable expansion (r2, r4)

Accumulator Variable Expansion

```
        r16 = r26 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
        ------------------
        r11 = load(r2)
        r13 = load(r4)
        r15 = r11 * r13
iter2   r16 = r16 + r15
        r2 = r2 + 4
        r4 = r4 + 4
        ------------------
        r21 = load(r2)
        r23 = load(r4)
        r25 = r21 * r23
iter3   r26 = r26 + r25
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop
        r6 = r6 + r16 + r26
```

- Accumulator variable
  - x = x + y or x = x – y
  - where y is loop variant!!
- Create n-1 temporary accumulators
- Each iteration targets a different accumulator
- Sum up the accumulator variables at the end
- May not be safe for floating-point values

Induction Variable Expansion

```
          r12 = r2 + 4, r22 = r2 + 8
          r14 = r4 + 4, r24 = r4 + 8
          r16 = r26 = 0
loop:     r1 = load(r2)
          r3 = load(r4)
          r5 = r1 * r3
iter1     r6 = r6 + r5
          r2 = r2 + 12
          r4 = r4 + 12
          ------------------
          r11 = load(r12)
          r13 = load(r14)
          r15 = r11 * r13
iter2     r16 = r16 + r15
          r12 = r12 + 12
          r14 = r14 + 12
          ------------------
          r21 = load(r22)
          r23 = load(r24)
          r25 = r21 * r23
iter3     r26 = r26 + r25
          r22 = r22 + 12
          r24 = r24 + 12
          if (r4 < 400) goto loop
          r6 = r6 + r16 + r26
```

- Induction variable
  - x = x + y or x = x – y
  - where y is loop invariant!!
- Create n-1 additional induction variables
- Each iteration uses and modifies a different induction variable
- Initialize induction variables to init, init+step, init+2*step, etc.
- Step increased to n*original step
- Now iterations are completely independent !!

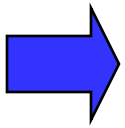
Better Induction Variable Expansion

```
        r16 = r26 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5

        ------------------
        r11 = load(r2+4)
        r13 = load(r4+4)
        r15 = r11 * r13
iter2   r16 = r16 + r15

        ------------------
        r21 = load(r2+8)
        r23 = load(r4+8)
        r25 = r21 * r23
iter3   r26 = r26 + r25
        r2 = r2 + 12
        r4 = r4 + 12
        if (r4 < 400) goto loop
        r6 = r6 + r16 + r26
```

- With base+displacement addressing, often don't need additional induction variables
  - Just change offsets in each iterations to reflect step
  - Change final increments to n * original step

Homework Problem

```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```



```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```

Optimize the unrolled loop

Renaming
Tree height reduction
Ind/Acc expansion