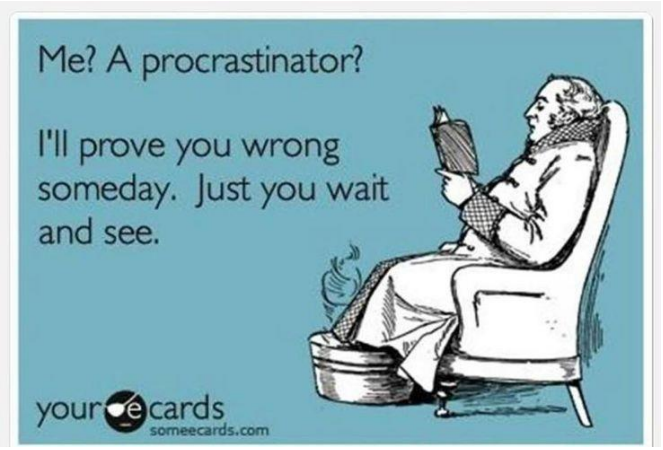
HW2 – Speculative Loop Invariant Code Motion



Do not try with homework2

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

```
for (int i = 0; i < n; i++) {
  x = y + z;
  a[i] = 6 * i + x * x;
}
```

- LICM code exists in LLVM!
  - /lib/Transforms/Scalar/LICM.cpp

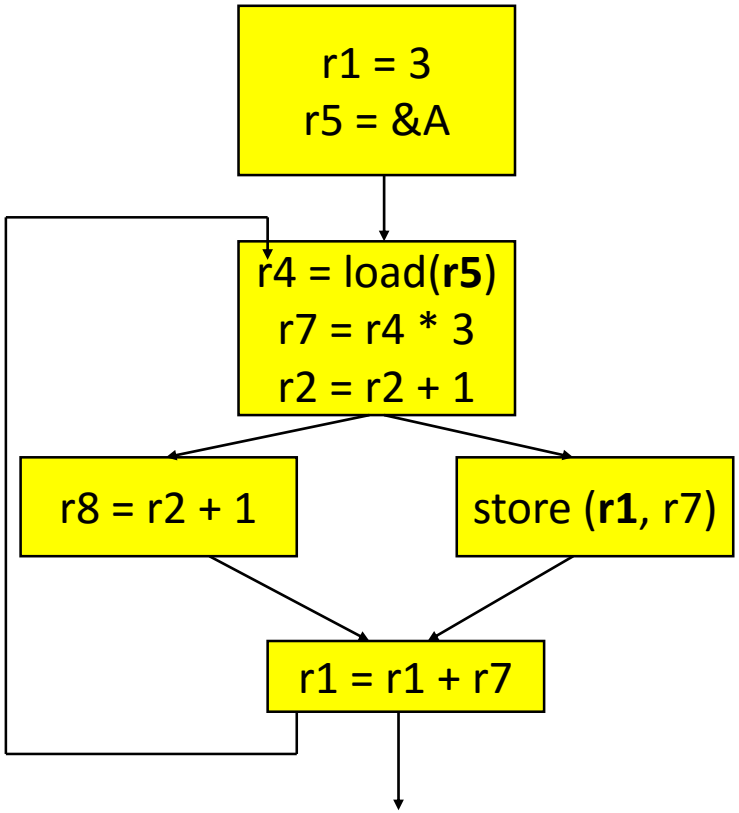
Loop Invariant Code Motion (LICM)

- 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 code exists in LLVM!
  - /lib/Transforms/Scalar/LICM.cpp

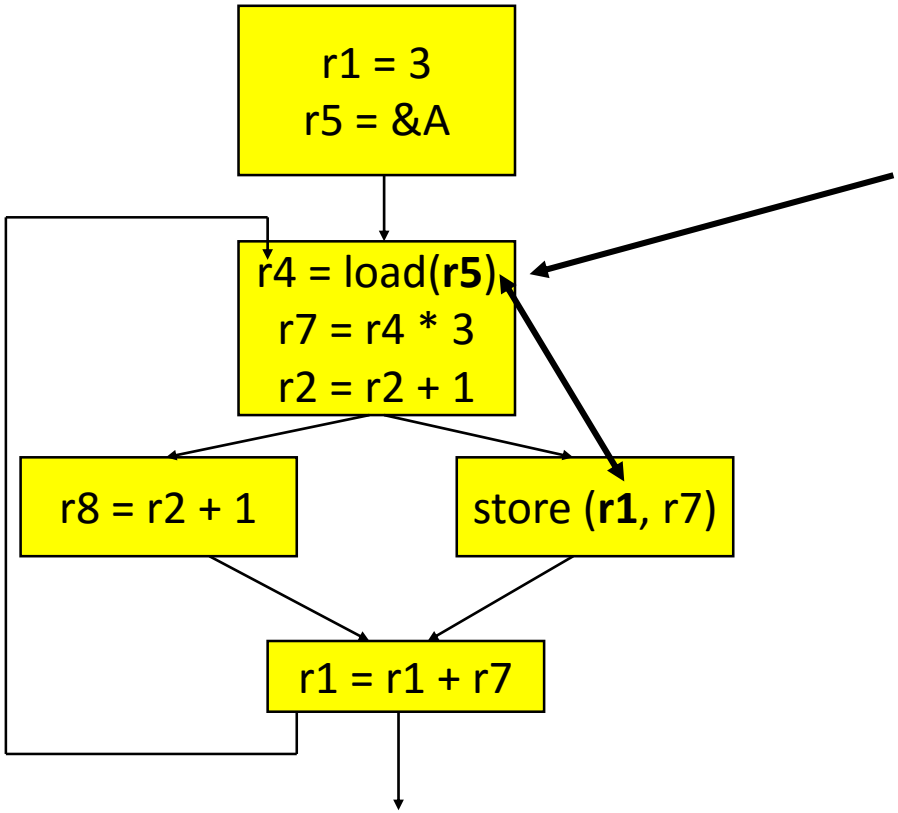
```
for (int i = 0; i < n; i++) {
  x = y + z;
  a[i] = 6 * i + x * x;
}
```

```
x = y + z;
t1 = x * x;
for (int i = 0; i < n; i++) {
  a[i] = 6 * i + t1;
}
```

Your Assignment: Speculative LICM



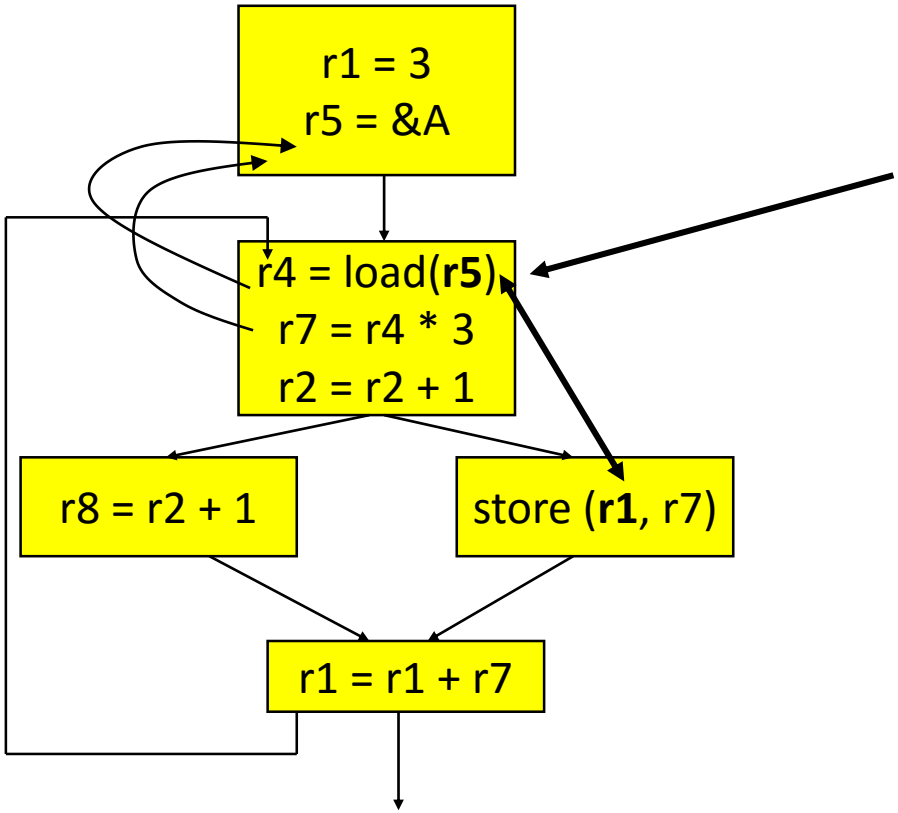
Your Assignment: Speculative LICM



Cannot perform LICM on load, because there may be an alias with the store

But… memory profile says that these rarely alias

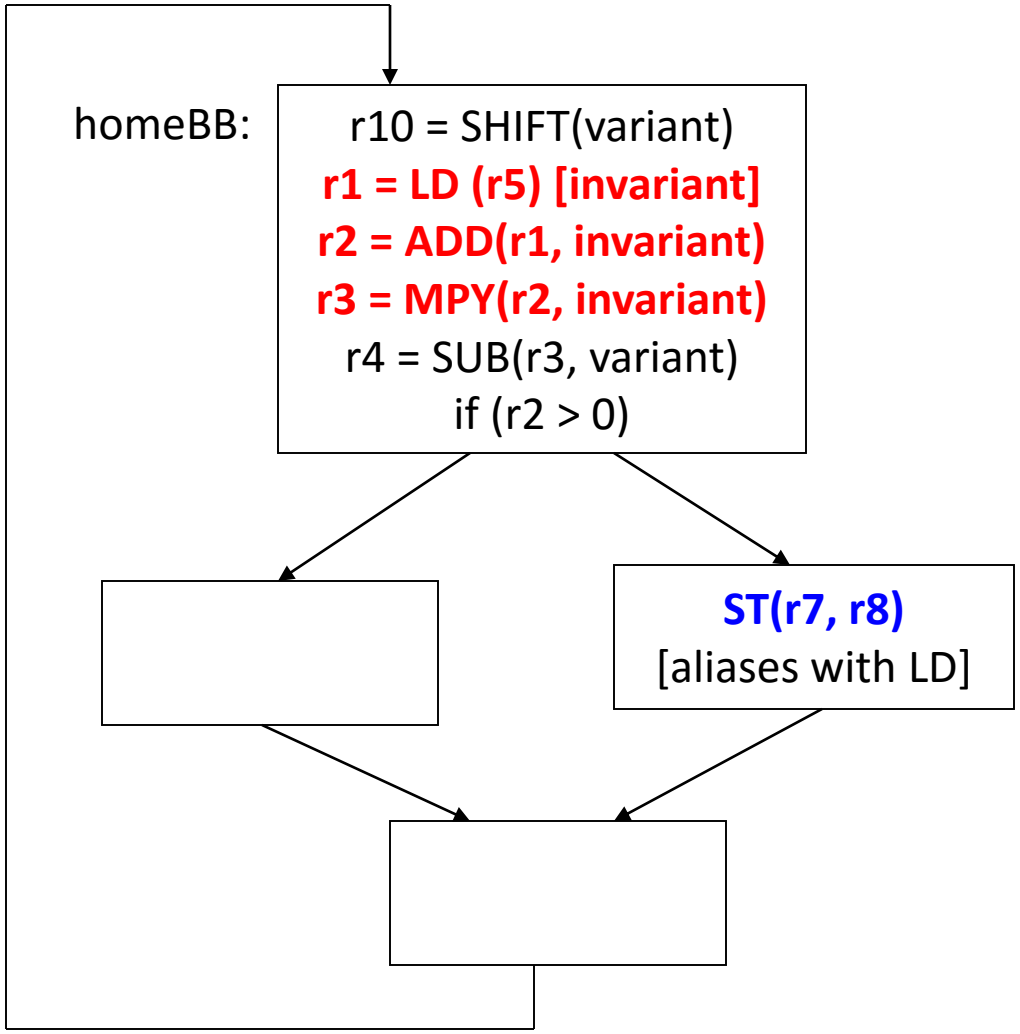
Your Assignment: Speculative LICM



Cannot perform LICM on load, because there may be an alias with the store

But… memory profile says that these rarely alias

Conceptual LICM Transformation for HW2



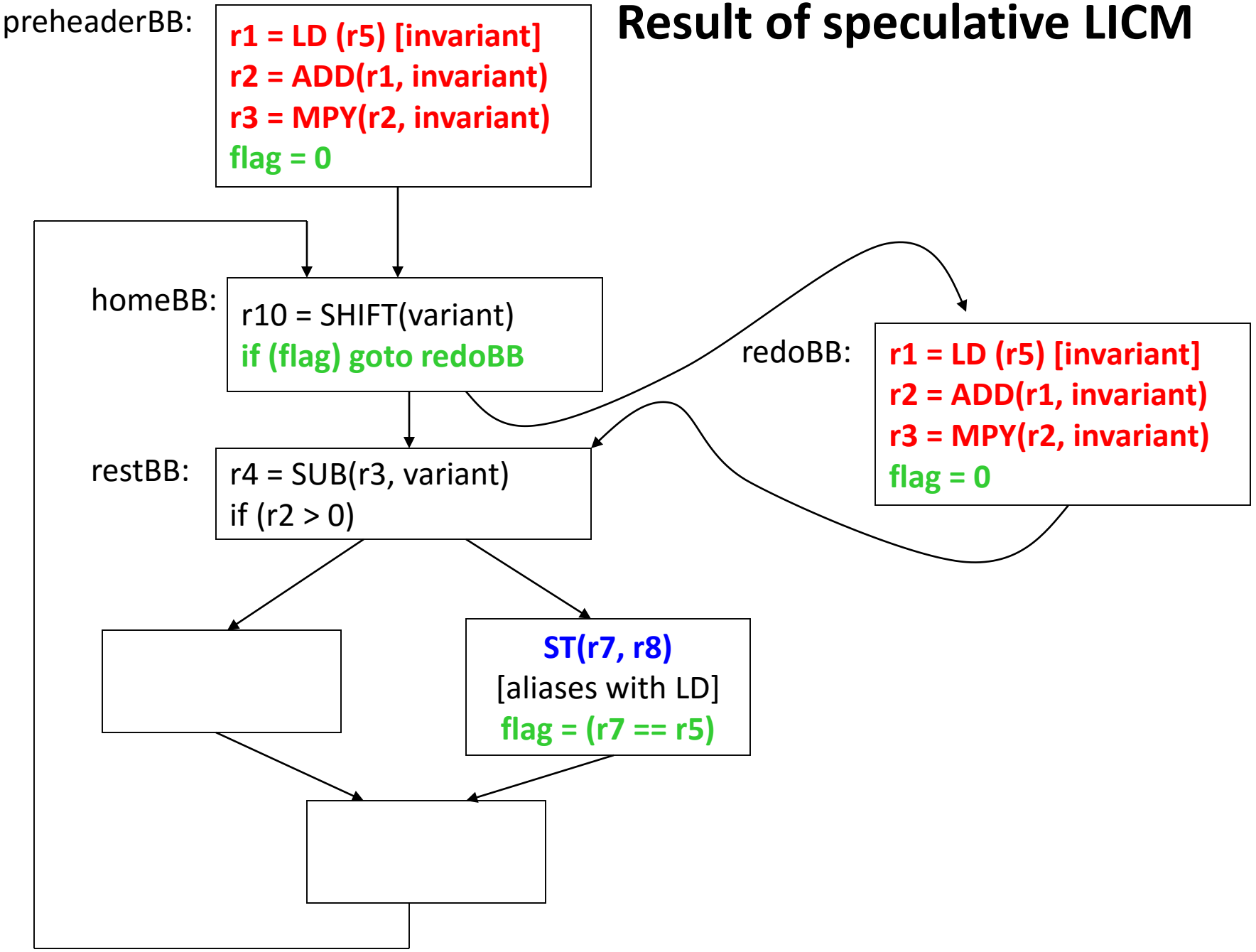
1) Find invariant load that cannot be hoisted due to aliasing store.

2) Check the memory dependence profile and see that the alias count between LD and ST is small

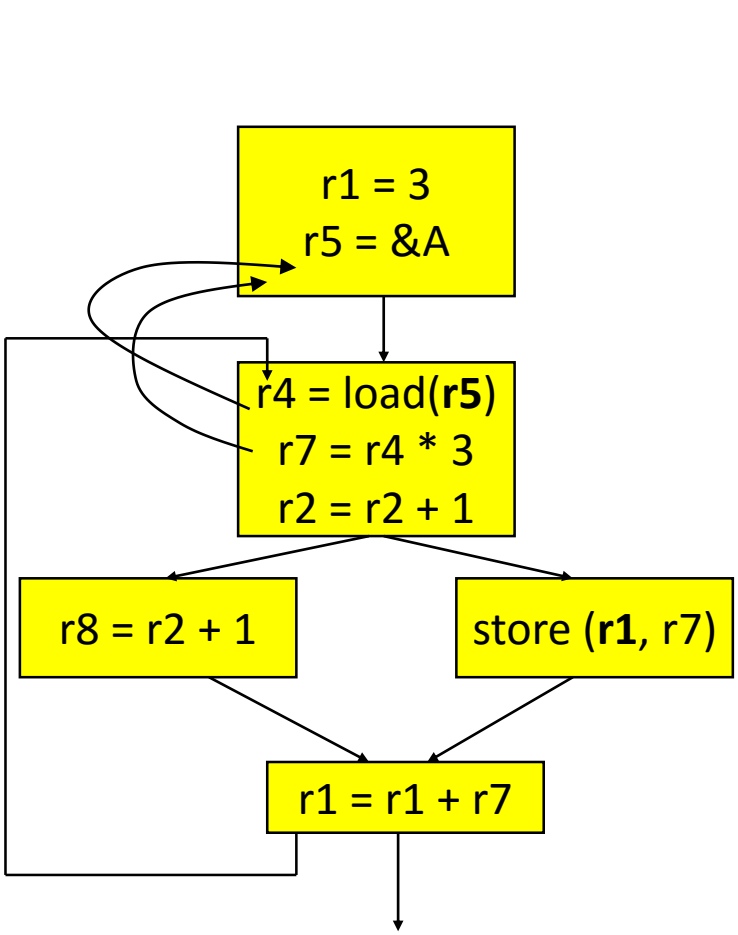
3) Hoist out the LD

4) LICM will then take over and hoist out any instructions that become invariant after the LD is removed. In this case, the ADD and MPY will also be moved out

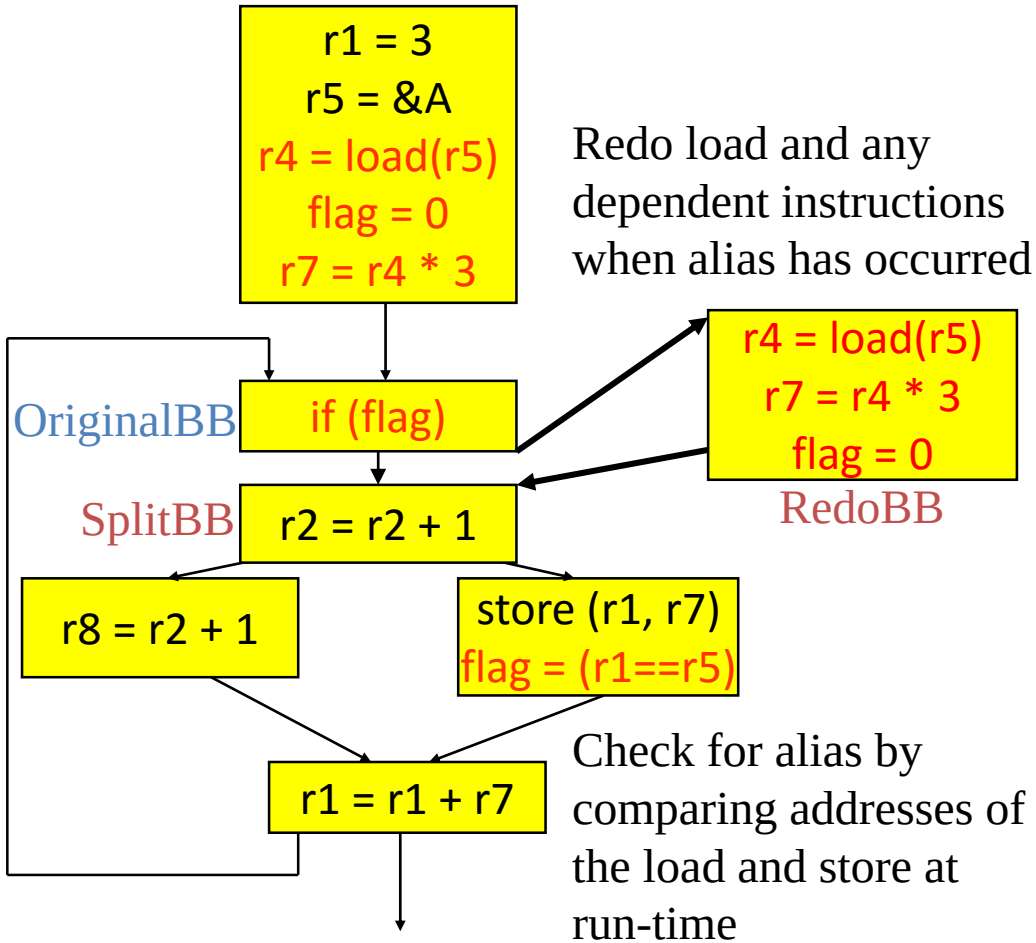
Result of speculative LICM



Speculative LICM



Before SLICM



Redo load and any dependent instructions when alias has occurred

Check for alias by comparing addresses of the load and store at run-time

After SLICM

General Notes Regarding HW2

- Start early!
- Template SLICM.cpp pass is provided
- Submit **two** versions of each benchmark
  - One that speculatively hoists **ALL** eligible loads
  - One that speculatively hoists invariant loads that your heuristic finds beneficial to hoist
- Running/Debugging
  - Revisit information from LLVM overview slides
- Performance Competition: Generate fast **AND** correct bitcode
- Use piazza!

Important: Maintaining SSA Form

- LLVM follows Single Static Assignment form – all instructions write in a new unique destination register
- Issue – The registers in redoBB will write into new destination registers and **you will have to make sure to pass these values to the user instructions**
- Recommended solution: Allow the mem2reg pass to handle this!
  - Store results of spec. hoisted instrs to stack variables
  - Make sure AllocaInst's are in function's entry BB
  - After running your pass, run the "**-mem2reg**" pass to convert these stack vars to registers and maintain SSA form
  - More info: http://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register

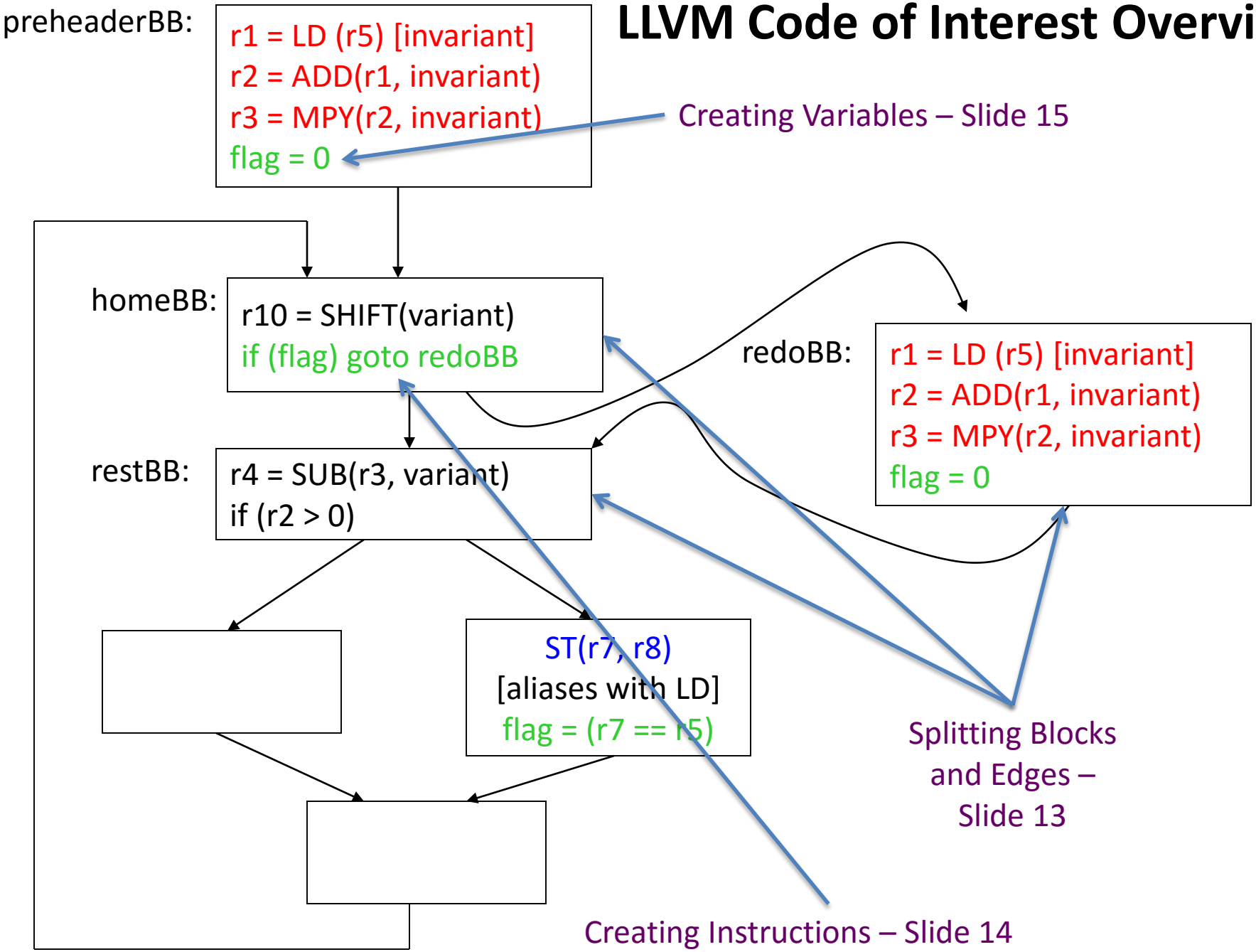
References for LLVM coding

- Later slides are useful for planning what exactly to code
- Most of the information present in the later slides is already in the homework

Future references

- Split home BB of hoisted LD
  - Everything above LD stays put in homeBB
  - Everything after LD gets put in new BB called restBB
  - New branch added to end of homeBB, that tests if flag==1 and branches to redoBB if true, and restBB if false
- redoBB
  - Place copy of hoisted LD, and copy of any other hoisted invariant instructions that directly or indirectly use the result of the LD.
    - It is important to populate the list of hoisted instructions correctly. Look at the set of instrs that are hoisted to identify those that became invariant because of the LD by examining use lists.
    - Invariant uses of the LD will automatically get hoisted by LICM, you do not need to do anything to make this happen
    - Even if these uses that become invariant occur in other BBs, you should redo them all in redoBB (think about this one!)
  - Clear flag at the end of redoBB.
- Preheader
  - Set all flags to 0 at end of preheader.
  - Note: each LD that you hoist should have its own flag variable.

LLVM Code of Interest Overview

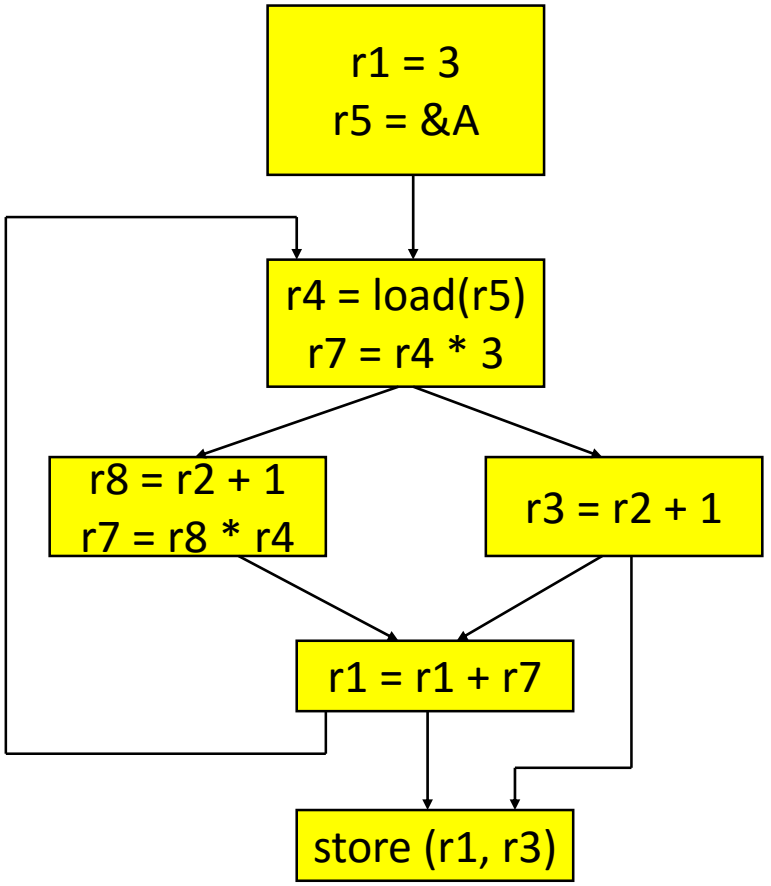


LLVM Code of Interest

- The following slides present code from the LLVM codebase that may help you with HW2.
- Disclaimers:
  - Use of this code is by no means required. There are many ways to do this assignment.
  - You are free to use any other code that exists in LLVM 3.3 or that you develop.
  - **Read the documentation/source before asking for help!**
    - **READ THIS!!!**http://llvm.org/docs/ProgrammersManual.html#helpful-hints-for-common-operations

LLVM's LICM Transformation

- ***LICM::runOnLoop(…)*** process current loop
  - Calls LICM::HoistRegion(…) to handle hoisting for the loop
- ***LICM::HoistRegion(…)*** processes loop's CFG
  - Makes sure instruction has invariant operands and is safe to hoist, then calls LICM::hoist(…) to hoist an instruction
  - LICM::HoistRegion(…) called recursively until whole loop's CFG is processed
- ***LICM::hoist(…)***
  - Hoists an instruction before the terminating instruction in the loop's preheader basic block



Code: Manipulating Basic Blocks

- SplitBlock(…) splits a BB at a specified instr, returns ptr to new BB that starts with the instr, connects the BBs with an unconditional branch
- SplitEdge(…) will insert a BB between two specified BBs
- Code found in:
  - `<llvm-src-root>/include/llvm/Transforms/Utils/BasicBlockUtils.h`
  - `<llvm-src-root>/lib/Transforms/Utils/BasicBlockUtils.cpp`

```
// I is an Instruction*
BasicBlock *BB1 = I->getParent();
BasicBlock *BB3 =
    SplitBlock(BB1, I, this);
BasicBlock *BB2 =
    SplitEdge(BB1, BB3, this);
```

Code: Creating and Inserting Instructions

- Various ways to create & insert instructions
- Hint: Instructions have a **clone()** member function
- See specific instruction constructors/member functions in:
  - `<llvm-src-root>/include/llvm/IR/Instructions.h`
- See general instruction functions available to all instructions in:
  - `<llvm-src-root>/include/llvm/IR/Instruction.h`

```
// 1) create load, insert at end of
//    specified basic block
LoadInst *LD =
    new LoadInst(flag,
                 "loadflag",
                 BB1);

// 2) create branch using Create
//    method, insert before BB1's
//    terminating instruction
Branch::Create(BB1, BB2, flag,
               BB1->getTerminator());

// 3) create a store inst that stores
//    result of LD to some variable
//    (related to next slide)
StoreInst *ST =
    new StoreInst(LD, var);
//    inserting store into code
ST->insertAfter(LD);
```

Code: Creating Variables

- Use AllocaInst to allocate space on the function's stack frame for a variable

```
// 1) Create flag variable in the
//    function Entry block
AllocaInst *flag = new
AllocaInst(
        Type::getInt1Ty(
            Entry->getContext(…)
        ),
        "flag",
        Entry->getTerminator()
        );

// 2) Using the flag
StoreInst *ST = new StoreInst(
        ConstantInt::getFalse(…),
        flag,
        Entry->getTerminator()
        );
```