

HW2 – Frequent Path Loop Invariant Code Motion

Sunghyun Park

Sep 23, 2019

Loop Invariant Code Motion (LICM)

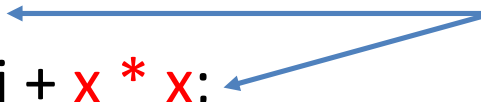
```
for (int i = 0; i < n; i++) {
```

```
    x = y + z;
```

```
    a[i] = 6 * i + x * x;
```

```
}
```

Their values don't
change within the loop



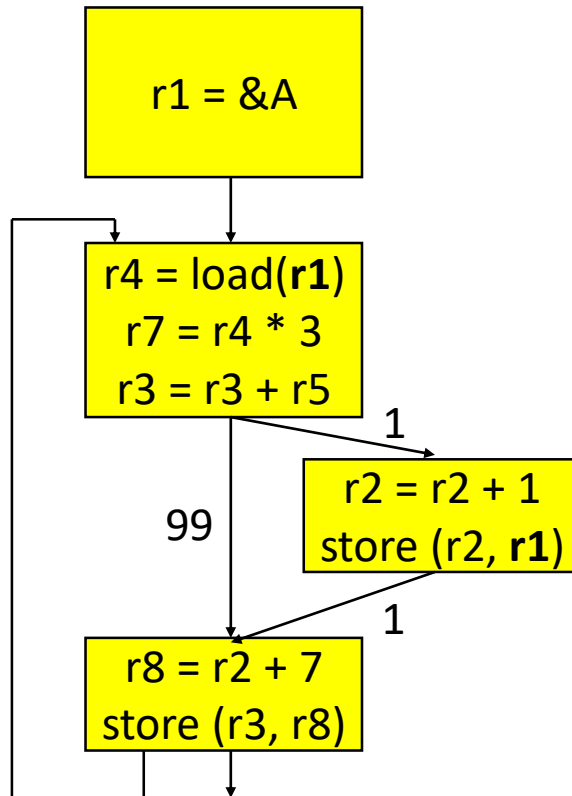
Loop Invariant Code Motion (LICM)

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

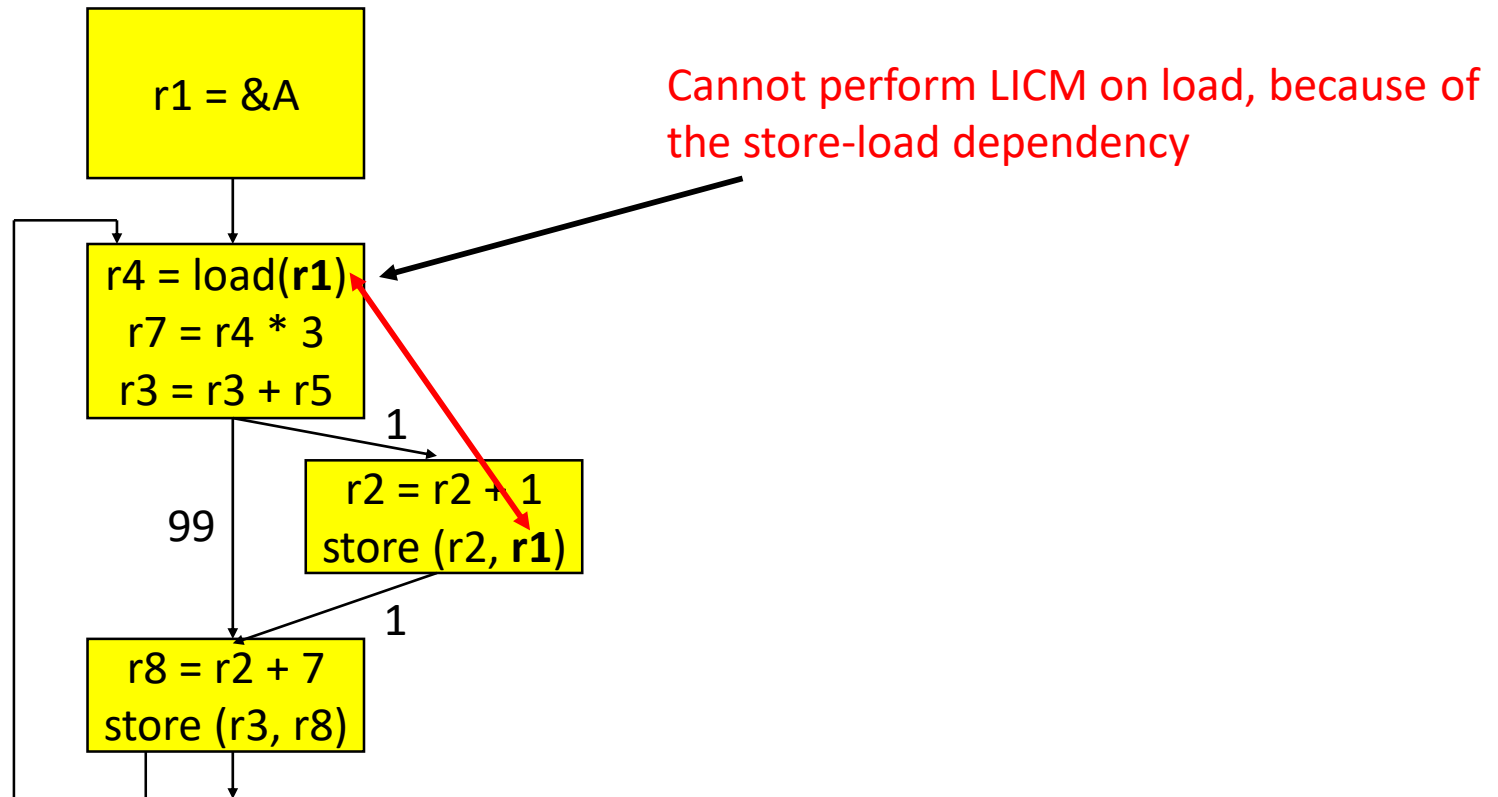
```
x = y + z;  
t1 = x * x;  
for (int i = 0; i < n; i++) {  
    a[i] = 6 * i + t1;  
}
```

- 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`

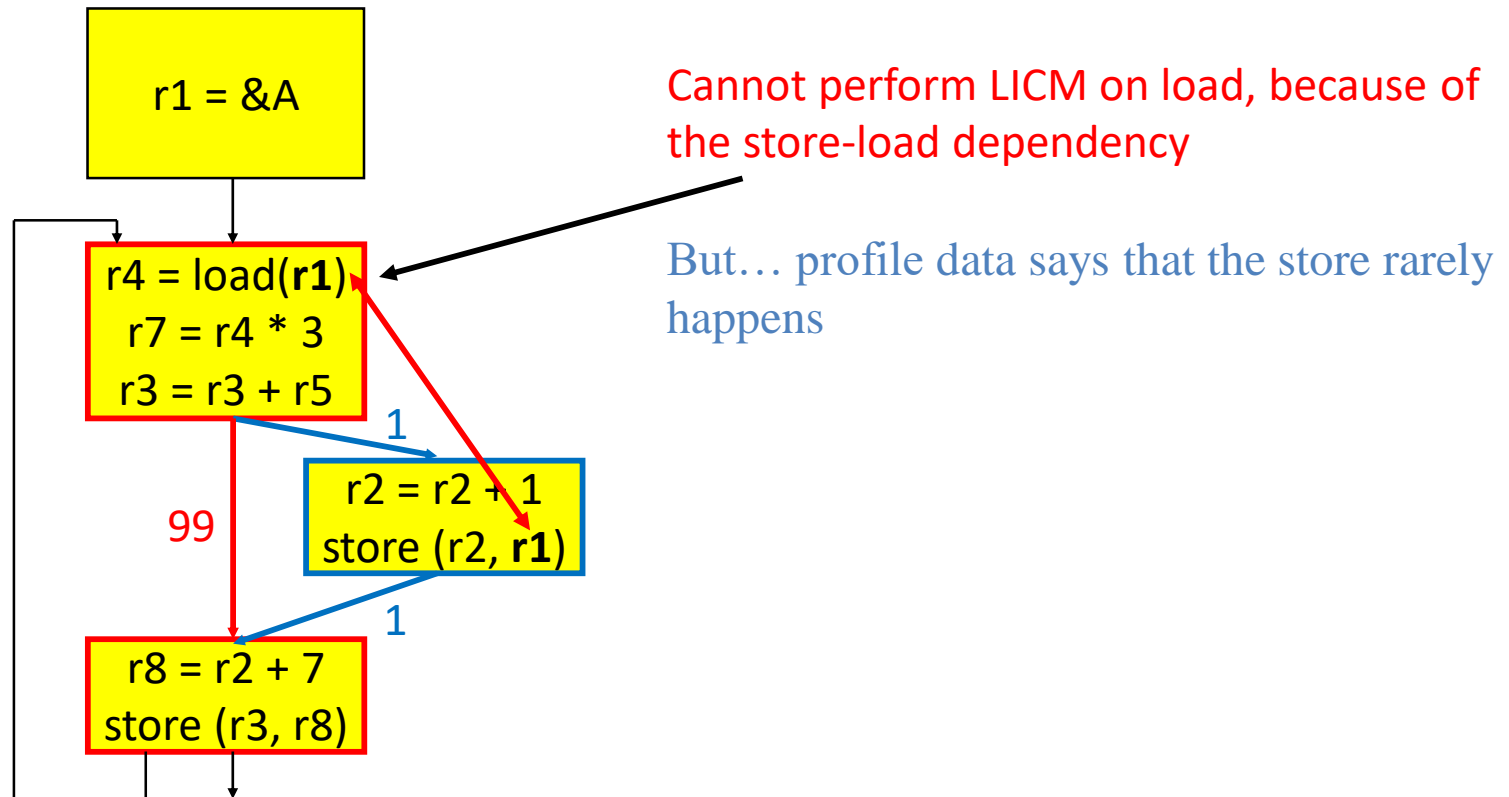
Your Assignment: Frequent Path LICM



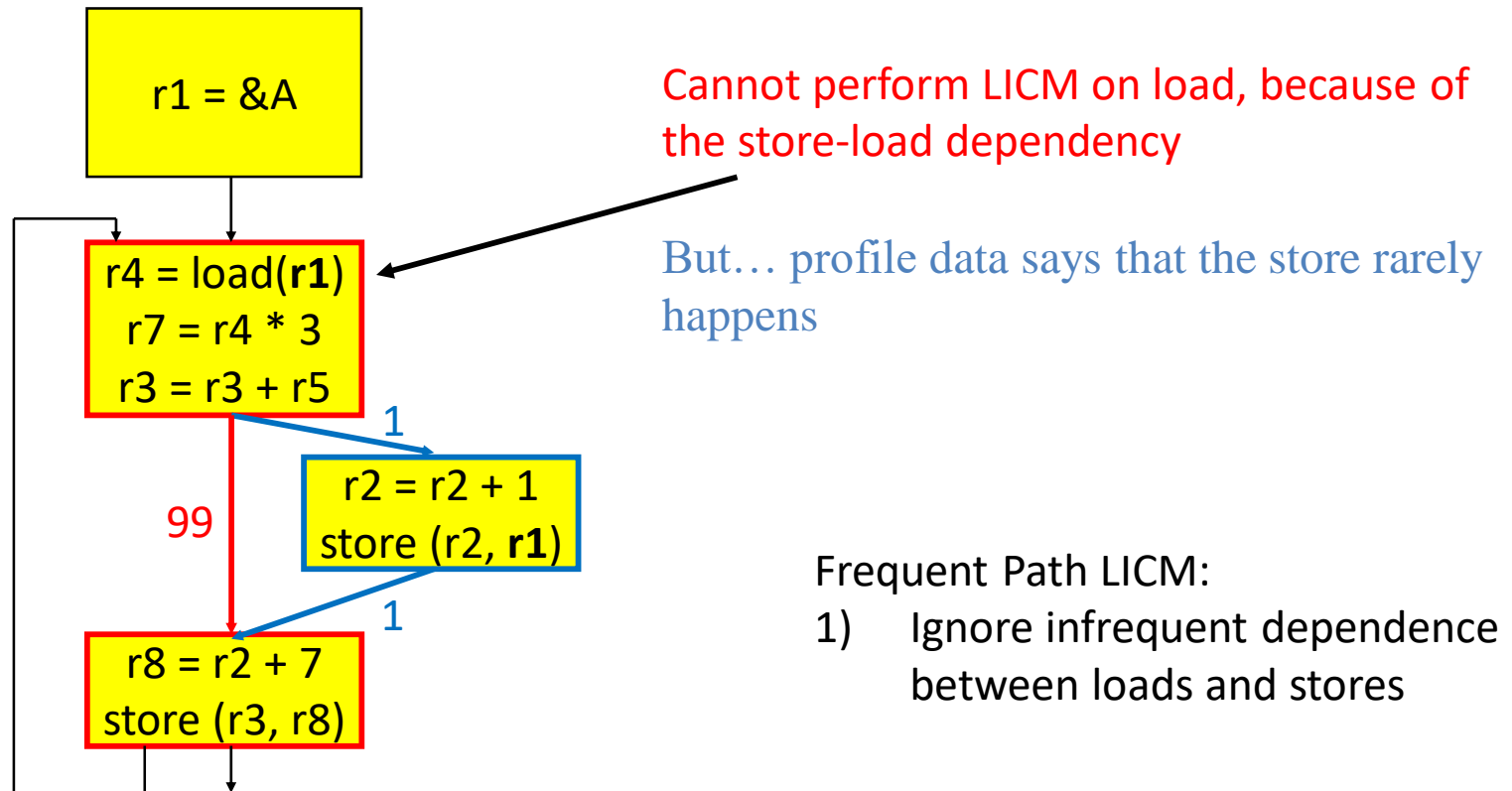
Your Assignment: Frequent Path LICM



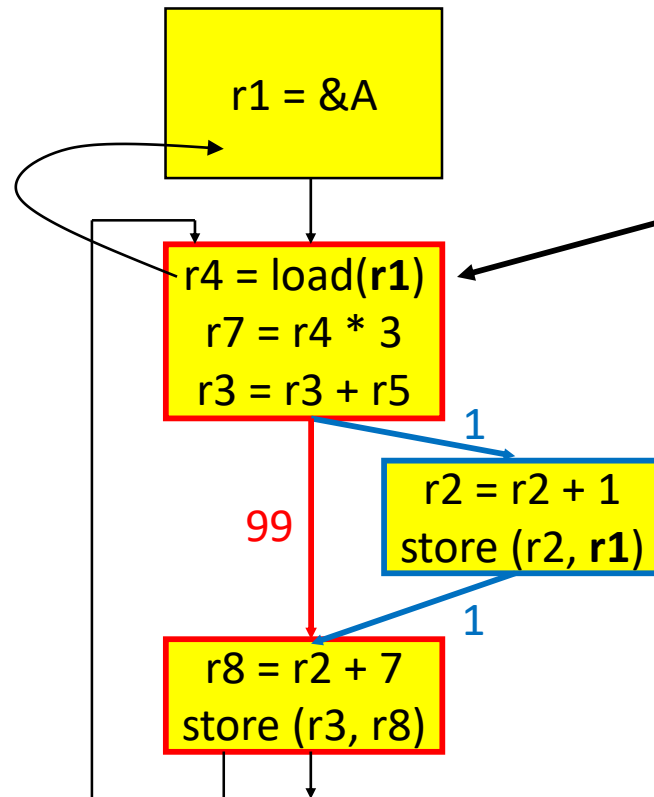
Your Assignment: Frequent Path LICM



Your Assignment: Frequent Path LICM



Your Assignment: Frequent Path LICM



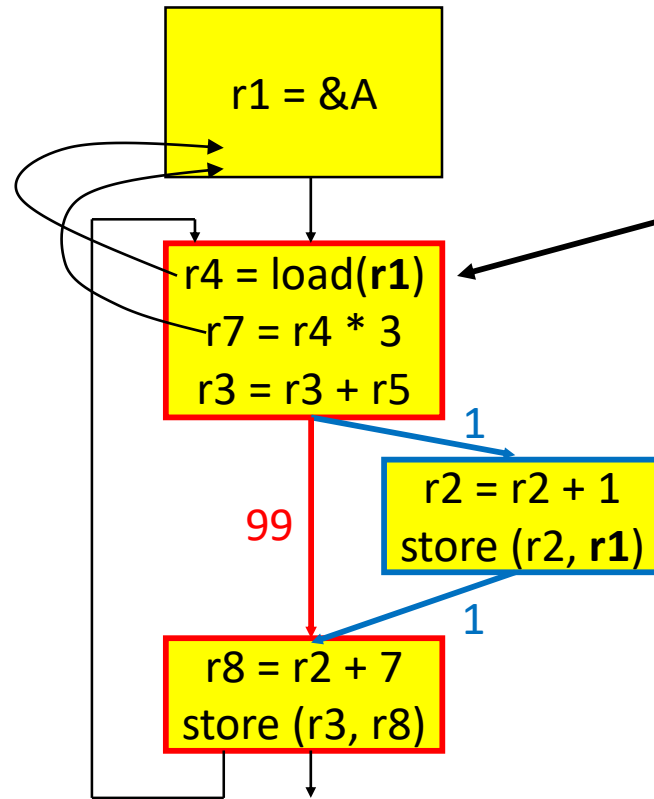
Cannot perform LICM on load, because of the store-load dependency

But... profile data says that the store rarely happens

Frequent Path LICM:

- 1) Ignore infrequent dependence between loads and stores
- 2) Perform LICM on load

Your Assignment: Frequent Path LICM



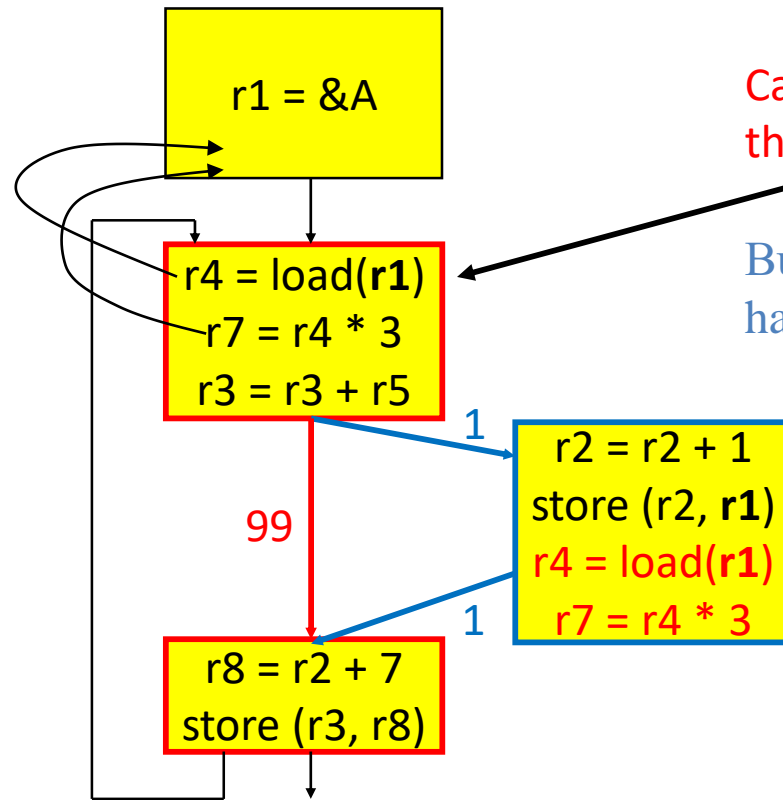
Cannot perform LICM on load, because of the store-load dependency

But... profile data says that the store rarely happens

Frequent Path LICM:

- 1) Ignore infrequent dependence between loads and stores
- 2) Perform LICM on load
- 3) Perform LICM on any consumers of the load that become invariant

Your Assignment: Frequent Path LICM



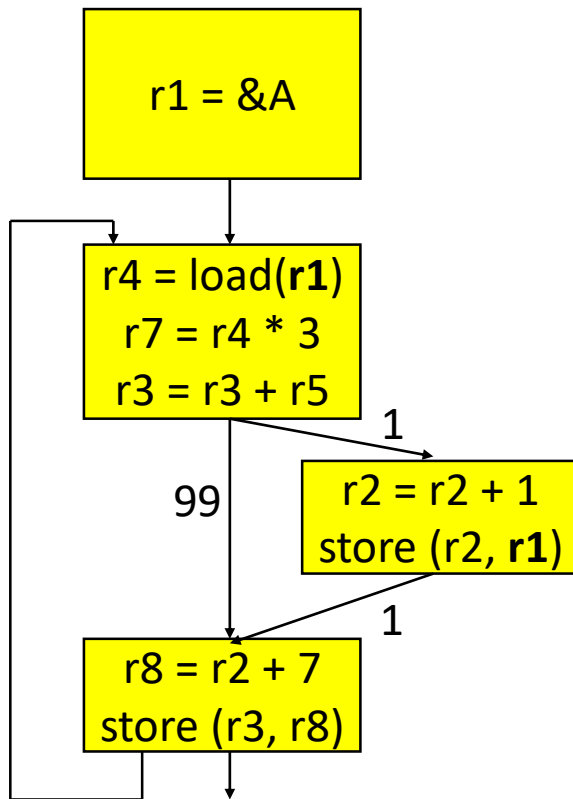
Cannot perform LICM on load, because of the store-load dependency

But... profile data says that the store rarely happens

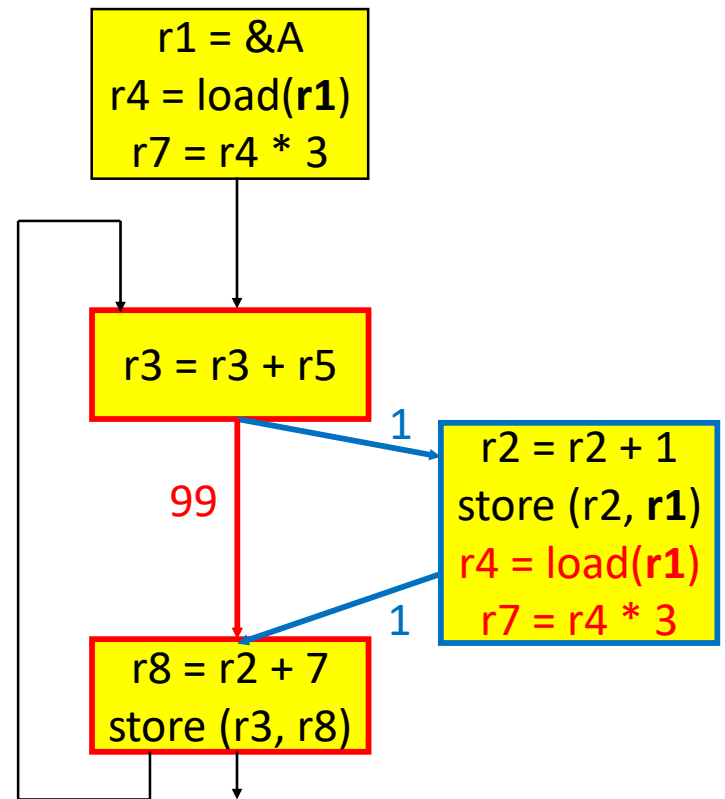
Frequent Path LICM:

- 1) Ignore infrequent dependence between loads and stores
- 2) Perform LICM on load
- 3) Perform LICM on any consumers of the load that become invariant
- 4) Insert fix-up code to restore correct execution

Your Assignment: Frequent Path LICM



Before FPLICM



After FPLICM

HW2: Frequent Path (FP) LICM

- Identify the FP within the loop
- Find store instructions among all infrequent BBs and their dependent load instructions in frequent BBs
 - destination operand of infrequent store = source operand of frequent load*
 - For simplicity, don't worry about pointer for this assignment.
- Create a heuristic that determines to perform FP LICM or not.
 - Smart heuristic should apply optimization when it's profitable.
- Hoist the profitable FP invariants.
 - Load instruction
 - Consumers of the load that become invariant* (For bonus points)
- Replicate all hoisted instructions in the infrequent path

HW2: Useful Resources

- `run.sh`
 - List of commands used in HW2
 - Check correctness of your pass!
- Project Template
 - `HW2PASS.cpp`: Mostly from current LLVM LICM Implementation.
 - *`runOnLoop(...)` `hoistRegion(...)` `hoist(...)`*
- Benchmarks
 - 6 correctness tests + README (Required)
 - Only need to hoist the dependent load instructions
 - Must generate the correct output after applying your FPLICM pass
 - 4 performance tests + README (Optional)
 - Hoist as many instructions as possible
 - Correctness first, then the performance

LLVM Code of Interest

- The following slides present code from the LLVM codebase that may help you with HW2.
- Disclaimers:
 - Use of following API is your choice. There are many ways to do this assignment.
 - You are free to use any other code that exists in LLVM 8.0.1 or that you develop.
 - **Read the documentation/source before asking for help!**
<http://llvm.org/docs/ProgrammersManual.html#helpful-hints-for-common-operations>

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

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

- `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`

Code: Creating and Inserting Instructions

- Various ways to create & insert instructions

```
// 1) create load, insert at end of
//    specified basic block
LoadInst *LD =
    new LoadInst(Val,
                  "loadflag",
                  BB1);
```
 - Hint: Instructions have a **clone()** member function
 - See specific instruction constructors/member functions in:
 - `<llvm-src-root>/include/llvm/IR/Instruction.s.h`
 - See general instruction functions available to all instructions in:
 - `<llvm-src-root>/include/llvm/IR/Instruction.h`
- ```
// 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 `AllocInst` to allocate memory space on the stack.

```
// 1) Create a variable in the
// function Entry block
AllocInst *Val = new AllocInst(
 I->getType(),
 0,
 Entry->getTerminator()
);
```

```
// 2) store to the variable
StoreInst *ST = new StoreInst(
 Result,
 Val,
 Entry->getTerminator()
);
```

# Important: Maintaining SSA Form

- Static Single Assignment form requires unique destination registers for each instruction
  - Replicated instructions in your infrequent BB will write to different regs compared to the instructions in the preheader!
  - Store results of hoisted instrs to stack variables (see prev. slide)
  - Make sure AllocInst's are in function's entry BB!

# General Notes Regarding HW2

- Start early!
- Will be released on 9/25 (Wed)
- Make sure your optimization doesn't break a program!
- Start with script/template.
- Try the bonus part
- Running/Debugging
  - Revisit information from LLVM overview slides
- Performance Competition: Generate correct **AND** fast bitcode

# Temporary GSI

- Ze Zhang (zezhang@umich.edu)
- 9/26 (Thu), 9/27(Fri), 10/1 (Tue)
- Office hours/location will be same!

