15-745: Optimizing Compilers for Modern Architectures

Lecture 1: Introduction

What would you get out of this course?

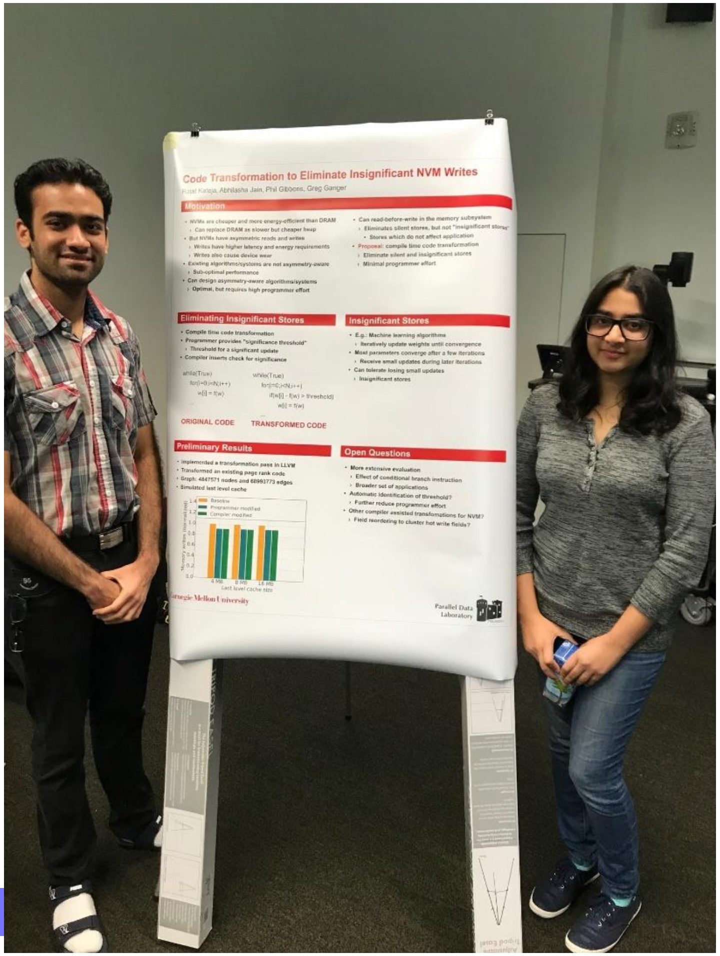
Structure of a Compiler

Optimization Example

Course Logistics

- If you are on the waitlist, come see me after class
  - This course is not intended to be your first compiler course

- Let Abilasha know if can’t get on Piazza or Canvas for this course



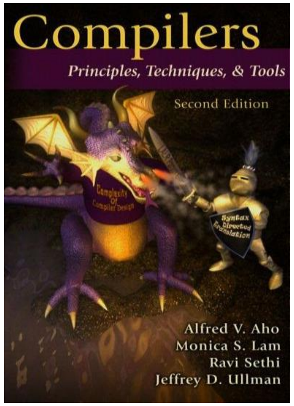
Poster Session, Spring 2017

Course Logistics

- If you are on the waitlist, come see me after class
  - This course is not intended to be your first compiler course

- Let Abilasha know if can't get on Piazza or Canvas for this course

- Need to get the book



- Let's run through the course webpage at http://www.cs.cmu.edu/~15745/

What Do Compilers Do?

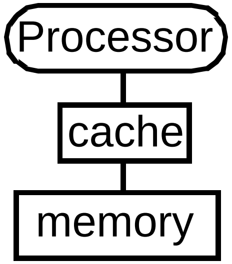
1. Translate one language into another
   - e.g., convert C++ into x86 object code
   - difficult for “natural” languages, but feasible for computer languages

2. Improve (i.e. “optimize”) the code
   - e.g., make the code run 3 times faster
     - or more energy efficient, more robust, etc.
   - driving force behind modern processor design

How Can the Compiler Improve Performance?

Execution time = **Operation count** * **Machine cycles per operation**

- **Minimize the number of operations**
  - arithmetic operations, memory accesses
- **Replace expensive operations with simpler ones**
  - e.g., replace 4-cycle multiplication with 1-cycle shift
- **Minimize cache misses**
  - both data and instruction accesses
- **Perform work in parallel**
  - instruction scheduling within a thread
  - parallel execution across multiple threads

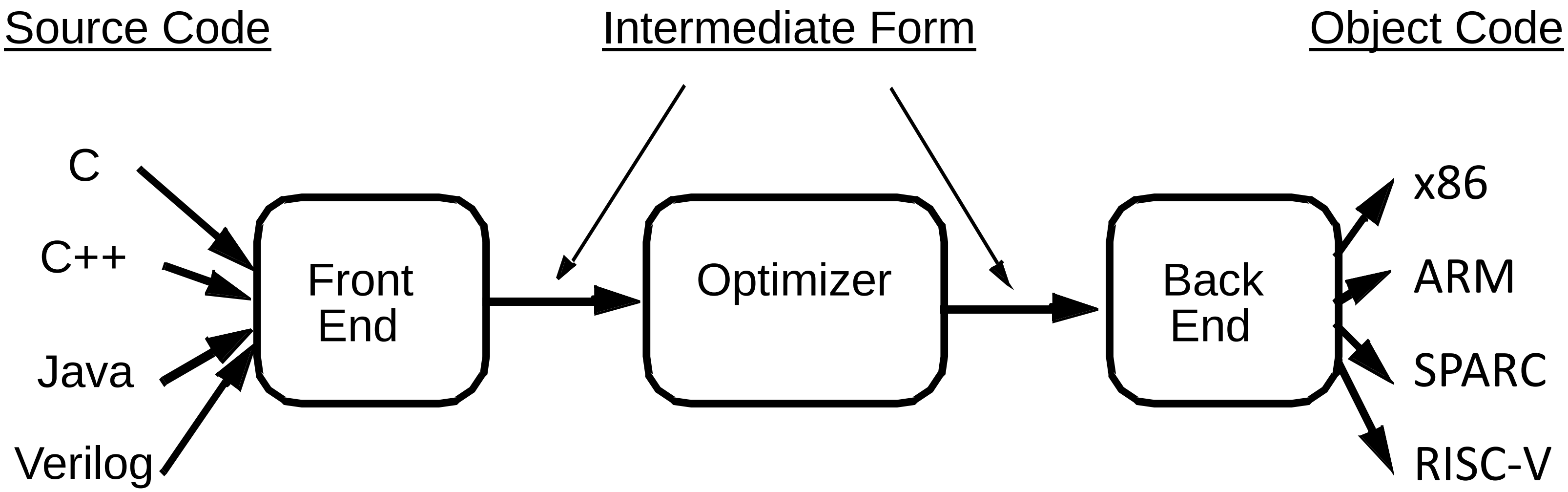


More accurately, machine cycles per operation must account for instruction overlap

What Would You Get Out of This Course?

- Basic knowledge of existing compiler optimizations
- Hands-on experience in constructing optimizations within a fully functional research compiler
- Basic principles and theory for the development of new optimizations

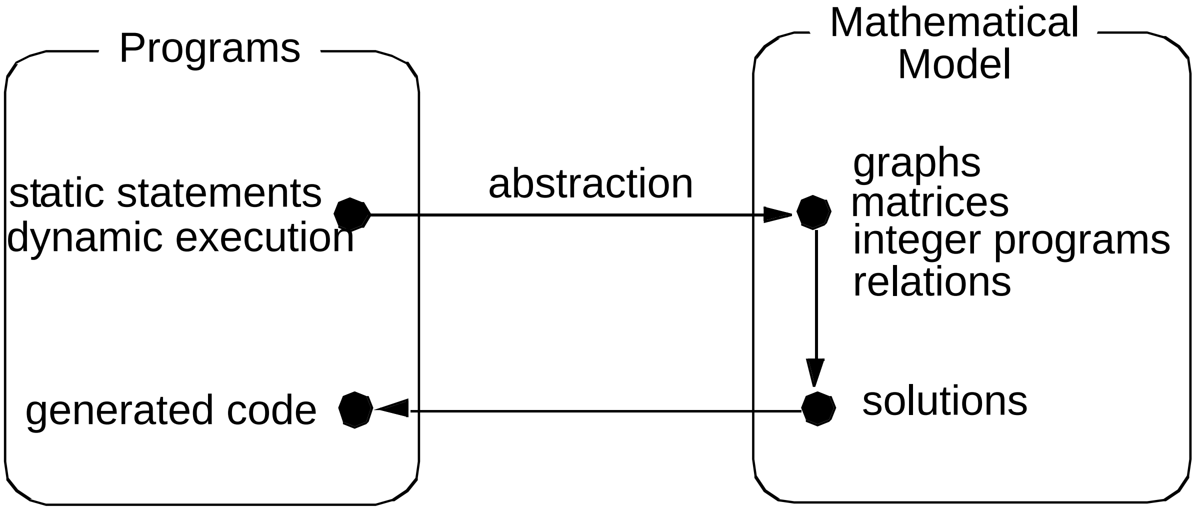
II. Structure of a Compiler



- **Optimizations are performed on an “intermediate form”**
  - similar to a generic RISC instruction set
- **Enables easy portability to multiple source languages, target machines**

Ingredients in a Compiler Optimization

- **Formulate optimization problem**
  - Identify opportunities of optimization
    - applicable across many programs
    - affect key parts of the program (loops/recursions)
    - amenable to "efficient enough" algorithm
- **Representation**
  - Must abstract essential details relevant to optimization



Ingredients in a Compiler Optimization

- **Formulate optimization problem**
  - Identify opportunities of optimization
    - applicable across many programs
    - affect key parts of the program (loops/recursions)
    - amenable to “efficient enough” algorithm
- **Representation**
  - Must abstract essential details relevant to optimization
- **Analysis**
  - Detect when it is desirable and safe to apply transformation
- **Code Transformation**

- **Experimental Evaluation (and repeat process)**

Representation: Instructions

- **Three-address code**

  `A := B op C`

  - LHS: name of variable e.g. **`x, A[t]`** (address of **`A`** + contents of **`t`**)
  - RHS: value

- **Typical instructions**

```
A := B op C
A := unaryop B
A := B
GOTO s
IF A relop B GOTO s
CALL    f
RETURN
```

III. Optimization Example: Bubblesort

- **Bubblesort program that sorts an array `A` that is allocated in static storage:**
  - an element of `A` requires four bytes of a byte-addressed machine
  - elements of `A` are numbered 1 through `n` (`n` is a variable)
  - `A[j]` is in location `&A+4*(j-1)`

```
for (i = n-1; i >= 1; i--) {
    for (j = 1; j <= i; j++)
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
}
```

Translated (Pseudo) Code

```
     i  := n-1
L5:  if i<1 goto L1
     j  := 1
L4:  if j>i goto L2
     t1  := j-1
     t2  := 4*t1
     t3  := A[t2]    ;A[j]
     t4  := j+1
     t5  := t4-1
     t6  := 4*t5
     t7  := A[t6]    ;A[j+1]
     if t3<=t7 goto L3
```

```
     t8  := j-1
     t9  := 4*t8
     temp  := A[t9]   ;temp:=A[j]
     t10  := j+1
     t11:= t10-1
     t12 := 4*t11
     t13  := A[t12]   ;A[j+1]
     t14  := j-1
     t15  := 4*t14
     A[t15]  := t13 ;A[j]:=A[j+1]
     t16  := j+1
     t17  := t16-1
     t18  := 4*t17
     A[t18]:=temp   ;A[j+1]:=temp
L3:  j  := j+1
     goto L4
L2:  i  := i-1
     goto L5
L1:
```

```
for (i = n-1; i >= 1; i--) {
  for (j = 1; j <= i; j++)
    if (A[j] > A[j+1]) {
      temp = A[j];
      A[j] = A[j+1];
      A[j+1] = temp;
    }
}
```

Instructions

29 in outer loop

25 in inner loop

Representation: a Basic Block

- **Basic block = a sequence of 3-address statements**
  - only the first statement can be reached from outside the block (no branches into middle of block)
  - all the statements are executed consecutively if the first one is (no branches out or halts except perhaps at end of block)

- **We require basic blocks to be *maximal***
  - they cannot be made larger without violating the conditions

- **Optimizations within a basic block are *local* optimizations**

Find the Basic Blocks

```
     i  := n-1                            B1
L5:  if i<1 goto L1                       B2
     j  := 1                              B3
L4:  if j>i goto L2                       B4
     t1  := j-1                           B5
     t2  := 4*t1
     t3  := A[t2]      ;A[j]
     t4  := j+1
     t5  := t4-1
     t6  := 4*t5
     t7  := A[t6]      ;A[j+1]
     if t3<=t7 goto L3
```

```
     t8  := j-1                           B6
     t9  := 4*t8
     temp  := A[t9]    ;temp:=A[j]
     t10  := j+1
     t11:= t10-1
     t12 := 4*t11
     t13  := A[t12]    ;A[j+1]
     t14  := j-1
     t15  := 4*t14
     A[t15]  := t13 ;A[j]:=A[j+1]
     t16  := j+1
     t17  := t16-1
     t18  := 4*t17
     A[t18]:=temp   ;A[j+1]:=temp
L3:  j  := j+1                            B7
     goto L4
L2:  i  := i-1                            B8
     goto L5
L1:
```

Basic Block:

Only enter at first

Only exit at last

Flow Graphs

- **Nodes: basic blocks**

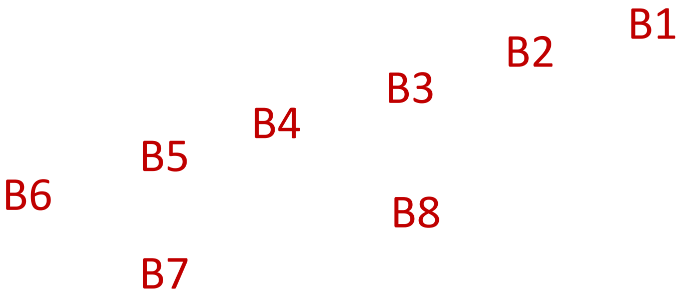
- **Edges: B_i -> B_j, iff B_j can follow B_i immediately in *some* execution**
  - Either first instruction of B_j is target of a goto at end of B_i
  - Or, B_j physically follows $B_{i,}$ which does not end in an unconditional goto.

- **The block led by first statement of the program is the *start*, or *entry* node.**

Example Flow Graph

```
     i  := n-1                      B1
L5:  if i<1 goto L1                 B2
     j  := 1                        B3
L4:  if j>i goto L2                 B4
     t1  := j-1                     B5
     t2  := 4*t1
     t3  := A[t2]      ;A[j]
     t4  := j+1
     t5  := t4-1
     t6  := 4*t5
     t7  := A[t6]      ;A[j+1]
     if t3<=t7 goto L3
```

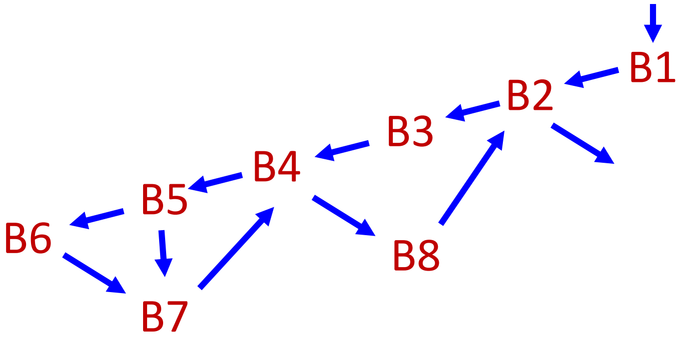
```
     t8  := j-1                               B6
     t9  := 4*t8
     temp  := A[t9]   ;temp:=A[j]
     t10  := j+1
     t11:= t10-1
     t12 := 4*t11
     t13  := A[t12]   ;A[j+1]
     t14  := j-1
     t15  := 4*t14
     A[t15]  := t13  ;A[j]:=A[j+1]
     t16  := j+1
     t17  := t16-1
     t18  := 4*t17
     A[t18]:=temp   ;A[j+1]:=temp
L3:  j  := j+1                                B7
     goto L4
L2:  i  := i-1                                B8
     goto L5
L1:
```



Example Flow Graph

```
     i  := n-1                      B1
L5:  if i<1 goto L1                 B2
     j  := 1                        B3
L4:  if j>i goto L2                 B4
     t1  := j-1                     B5
     t2  := 4*t1
     t3  := A[t2]     ;A[j]
     t4  := j+1
     t5  := t4-1
     t6  := 4*t5
     t7  := A[t6]     ;A[j+1]
     if t3<=t7 goto L3
```

```
     t8  := j-1                         B6
     t9  := 4*t8
     temp  := A[t9]   ;temp:=A[j]
     t10  := j+1
     t11:= t10-1
     t12 := 4*t11
     t13  := A[t12]   ;A[j+1]
     t14  := j-1
     t15  := 4*t14
     A[t15]  := t13 ;A[j]:=A[j+1]
     t16  := j+1
     t17  := t16-1
     t18  := 4*t17
     A[t18]:=temp   ;A[j+1]:=temp
L3:  j  := j+1                          B7
     goto L4
L2:  i  := i-1                          B8
     goto L5
L1:
```



Sources of Optimizations

- **Algorithm optimization**

- **Algebraic optimization**

  `A := B+0      =>      A := B`

- **Local optimizations**
  - within a basic block -- across instructions
- **Global optimizations**
  - within a flow graph -- across basic blocks
- **Interprocedural analysis**
  - within a program -- across procedures (flow graphs)

Local Optimizations

- **Analysis & transformation performed within a basic block**
- **No control flow information is considered**
- **Examples of local optimizations:**
  - local common subexpression elimination
    analysis: same expression evaluated more than once in a block
    transformation: replace with single calculation
  - local constant folding or elimination
    analysis: expression can be evaluated at compile time
    transformation: replace by constant, compile-time value
  - dead code elimination

Local Optimization (Redundancy in Address Calculation)

```
     i  := n-1
L5:  if i<1 goto L1
     j  := 1
L4:  if j>i goto L2
     t1  := j-1                 B5
     t2  := 4*t1
     t3  := A[t2]     ;A[j]
     t4  := j+1
     t5  := t4-1
     t6  := 4*t5
     t7  := A[t6]     ;A[j+1]
     if t3<=t7 goto L3
```

```
     t8  :=j-1
     t9  := 4*t8
     temp  := A[t9]   ;temp:=A[j]
     t10  := j+1
     t11:= t10-1
     t12 := 4*t11
     t13  := A[t12]   ;A[j+1]
     t14  := j-1
     t15  := 4*t14
     A[t15]  := t13 ;A[j]:=A[j+1]
     t16  := j+1
     t17  := t16-1
     t18  := 4*t17
     A[t18]:=temp   ;A[j+1]:=temp
L3:  j  := j+1
     goto L4
L2:  i  := i-1
     goto L5
L1:
```

Local Optimization Example

```
     i  := n-1
L5:  if i<1 goto L1
     j  := 1
L4:  if j>i goto L2
     t1  := j-1                  B5
     t2  := 4*t1
     t3  := A[t2]      ;A[j]
     t6  := 4*j
     t7  := A[t6]      ;A[j+1]
     if t3<=t7 goto L3
```

```
     t8  :=j-1
     t9  := 4*t8
     temp  := A[t9]   ;temp:=A[j]
     t10  := j+1
     t11:= t10-1
     t12 := 4*t11
     t13  := A[t12]   ;A[j+1]
     t14  := j-1
     t15  := 4*t14
     A[t15]  := t13  ;A[j]:=A[j+1]
     t16  := j+1
     t17  := t16-1
     t18  := 4*t17
     A[t18]:=temp    ;A[j+1]:=temp
L3:  j  := j+1
     goto L4
L2:  i  := i-1
     goto L5
L1:
```

Local Optimization Example

```
     i  := n-1
L5:  if i<1 goto L1
     j  := 1
L4:  if j>i goto L2
     t1  := j-1
     t2  := 4*t1
     t3  := A[t2]    ;A[j]
     t6  := 4*j
     t7  := A[t6]    ;A[j+1]
     if t3<=t7 goto L3
```

B6

```
     t8  :=j-1
     t9  := 4*t8
     temp  := A[t9]   ;temp:=A[j]
     t10  := j+1
     t11:= t10-1
     t12 := 4*t11
     t13  := A[t12]   ;A[j+1]
     t14  := j-1
     t15  := 4*t14
     A[t15]  := t13  ;A[j]:=A[j+1]
     t16  := j+1
     t17  := t16-1
     t18  := 4*t17
     A[t18]:=temp    ;A[j+1]:=temp
L3:  j  := j+1
     goto L4
L2:  i  := i-1
     goto L5
L1:
```

After Local Optimization

```
    i := n-1
L5: if i<1 goto L1
    j := 1
L4: if j>i goto L2
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2]       ;A[j]
    t6 := 4*j
    t7 := A[t6]      ;A[j+1]
    if t3<=t7 goto L3
```

B6

```
    t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;temp:=A[j]
    t12 := 4*j
    t13 := A[t12]   ;A[j+1]
    A[t9]:= t13     ;A[j]:=A[j+1]
    A[t12]:=temp    ;A[j+1]:=temp
L3: j := j+1
    goto L4
L2: i := i-1
    goto L5
L1:
```

Instructions

20 in outer loop

16 in inner loop

(Intraprocedural) Global Optimizations

- **Global versions of local optimizations**
  - global common subexpression elimination
  - global constant propagation
  - dead code elimination
- **Loop optimizations**
  - reduce code to be executed in each iteration
  - code motion
  - induction variable elimination
- **Other control structures**
  - Code hoisting: eliminates copies of identical code on parallel paths in a flow graph to reduce code size.

Global (Across Basic Blocks) Optimization Example

```
    i := n-1
L5: if i<1 goto L1
    j := 1
L4: if j>i goto L2
```

B5

```
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2]      ;A[j]
    t6 := 4*j
    t7 := A[t6]      ;A[j+1]
    if t3<=t7 goto L3
```

B6

```
    t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;temp:=A[j]
    t12 := 4*j
    t13 := A[t12]   ;A[j+1]
    A[t9]:= t13     ;A[j]:=A[j+1]
    A[t12]:=temp    ;A[j+1]:=temp
```

```
L3: j := j+1
    goto L4
L2: i := i-1
    goto L5
L1:
```

After Global Subexpression Elimination

```
     i  := n-1
L5:  if i<1 goto L1
     j  := 1
L4:  if j>i goto L2
     t1 := j-1                      B5
     t2 := 4*t1
     t3 := A[t2]       ;old_A[j]
     t6 := 4*j
     t7 := A[t6]       ;A[j+1]
     if t3<=t7 goto L3
```

```
     A[t2] := t7    ;A[j]:=A[j+1]       B6
     A[t6] := t3    ;A[j+1]:=old_A[j]
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Instructions

15 in outer loop

11 in inner loop

Induction Variable Elimination

- **Intuitively**
  - Loop indices are induction variables (counting iterations)
  - Linear functions of the loop indices are also induction variables (for accessing arrays)

- **Analysis: detection of induction variable**

- **Optimizations**
  - strength reduction:
    - replace multiplication by additions
  - elimination of loop index:
    - replace termination by tests on other induction variables

Induction Variable Elimination Example

```
    i  := n-1
L5: if i<1 goto L1
    j  := 1
L4: if j>i goto L2
    t1  := j-1
    t2  := 4*t1
    t3  := A[t2]
    t6  := 4*j
    t7  := A[t6]
    if t3<=t7 goto L3
    A[t2]  := t7
    A[t6]  := t3
L3: j  := j+1
    goto L4
L2: i  := i-1
    goto L5
L1:
```

After Induction Variable Elimination

```
    i  := n-1
L5: if i<1 goto L1
    j  := 1
L4: if j>i goto L2
    t1  := j-1
    t2  := 4*t1
    t3  := A[t2]
    t6  := 4*j
    t7  := A[t6]
    if t3<=t7 goto L3
    A[t2]  := t7
    A[t6]  := t3
L3: j  := j+1
    goto L4
L2: i  := i-1
    goto L5
L1:
```

```
    i  := n-1
L5: if i<1 goto L1
    t2  := 0
    t6  := 4
L4: t19  := 4*i
    if t6>t19 goto L2
    t3  := A[t2]
    t7  := A[t6]
    if t3<=t7 goto L3
    A[t2]  := t7
    A[t6]  := t3
L3: t2  := t2+4
    t6  := t6+4
    goto L4
L2: i  := i-1
    goto L5
L1:
```

Instructions
15 in outer loop
10 in inner loop

Loop Invariant Code Motion

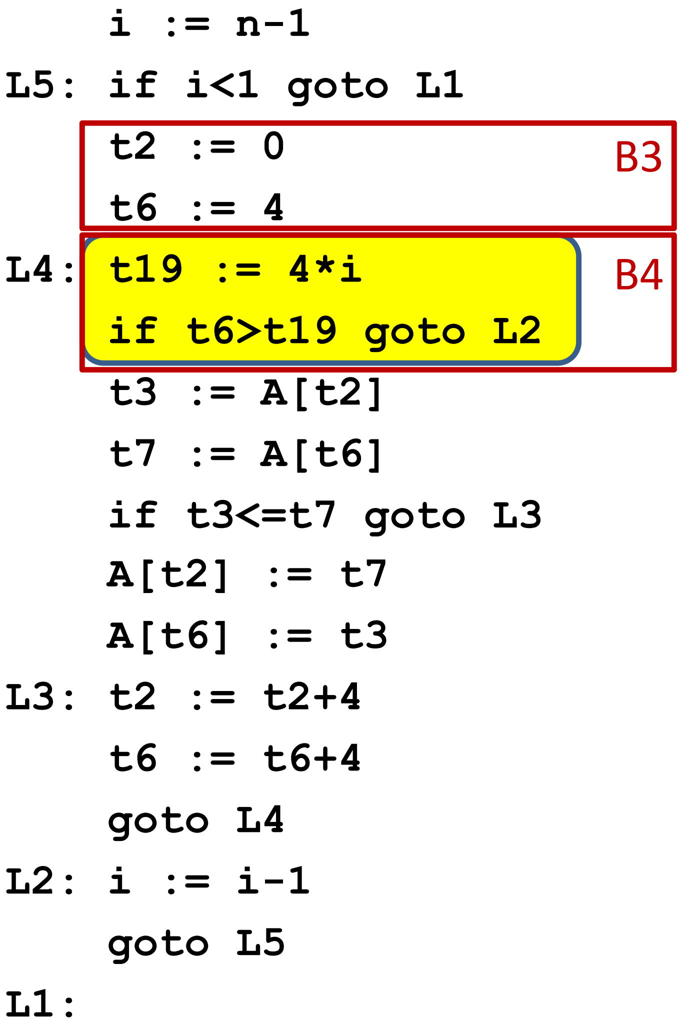
- **Analysis**
  - a computation is done within a loop and
  - result of the computation is the same as long as we keep going around the loop

- **Transformation**
  - move the computation outside the loop

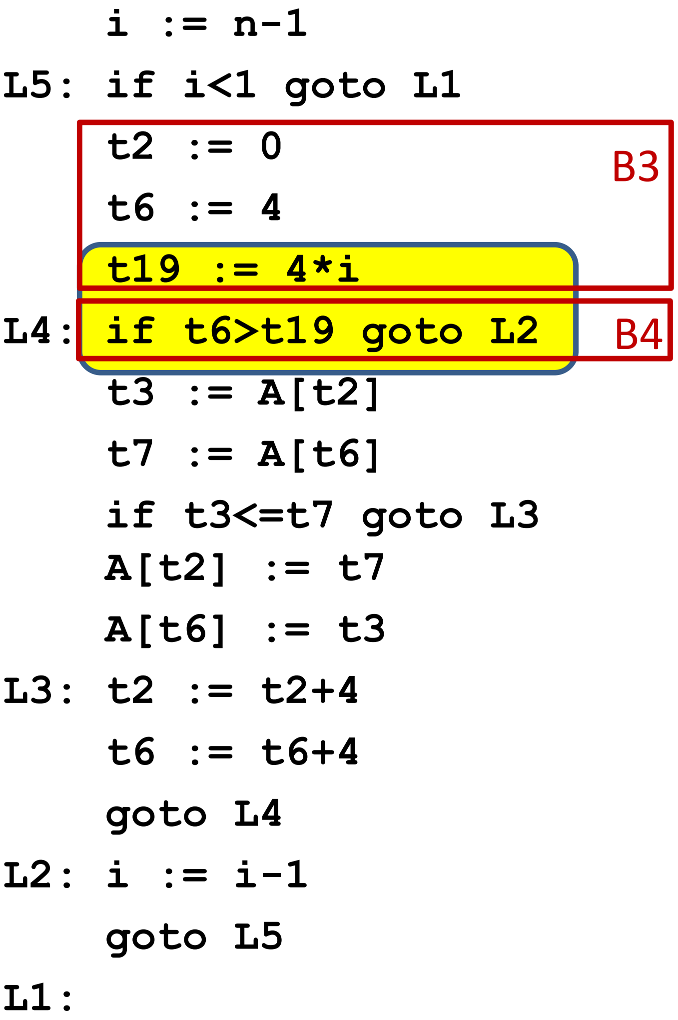
Loop Invariant Code Motion Example

```
     i  := n-1
L5:  if i<1 goto L1
     t2  := 0                 B3
     t6  := 4
L4:  t19  := 4*i              B4
     if t6>t19 goto L2
     t3  := A[t2]
     t7  := A[t6]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  t2  := t2+4
     t6  := t6+4
     goto L4
L2:  i  := i-1
     goto L5
L1:
```

After Loop Invariant Code Motion



```
     i  := n-1
L5:  if i<1 goto L1
     t2  := 0
     t6  := 4
L4:  t19  := 4*i
     if t6>t19 goto L2
     t3  := A[t2]
     t7  := A[t6]
     if t3<=t7 goto L3
     A[t2]  := t7
     A[t6]  := t3
L3:  t2  := t2+4
     t6  := t6+4
     goto L4
L2:  i  := i-1
     goto L5
L1:
```



```
     i  := n-1
L5:  if i<1 goto L1
     t2  := 0
     t6  := 4
     t19  := 4*i
L4:  if t6>t19 goto L2
     t3  := A[t2]
     t7  := A[t6]
     if t3<=t7 goto L3
     A[t2]  := t7
     A[t6]  := t3
L3:  t2  := t2+4
     t6  := t6+4
     goto L4
L2:  i  := i-1
     goto L5
L1:
```

Final Code

```
    i := n-1
L5: if i<1 goto L1
    t2 := 0
    t6 := 4
    t19 := i<<2
L4: if t6>t19 goto L2
    t3 := A[t2]
    t7 := A[t6]
    if t3<=t7 goto L3
    A[t2] := t7
    A[t6] := t3
L3: t2 := t2+4
    t6 := t6+4
    goto L4
L2: i := i-1
    goto L5
L1:
```

Instruction Count
Before Optimizations
29 in outer loop
25 in inner loop

Instruction Count
After Optimizations
15 in outer loop
9 in inner loop

Machine Dependent Optimizations

- Register allocation
- Instruction scheduling
- Memory hierarchy optimizations
- etc.

Wednesday’s Class

- Abhilasha will present “LLVM Compiler: Getting Started”
  - part 1 of 2 on LLVM

- Assignment 1 will be handed out

Reminder: Wait listed students see me now