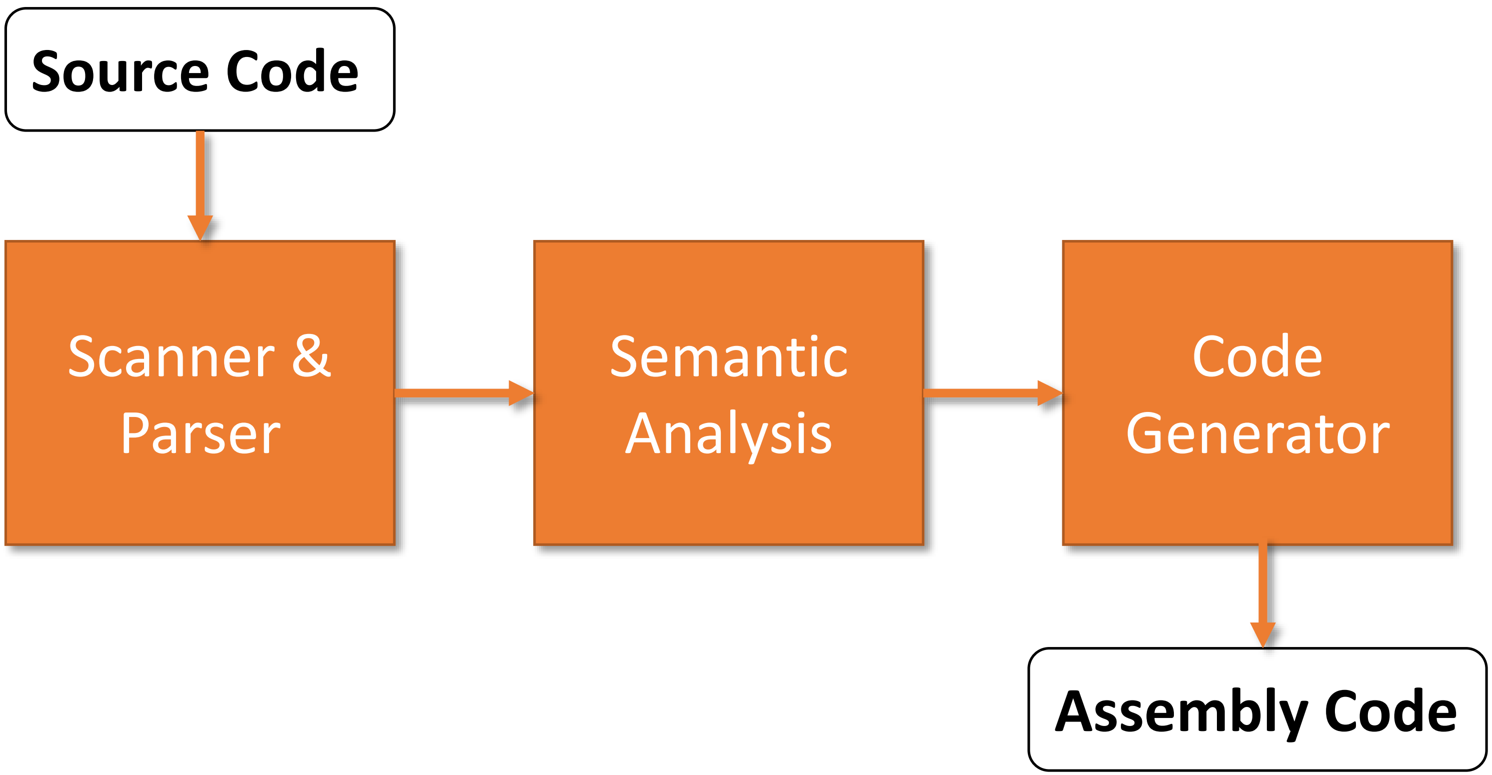
# 6.035 Infosession 4

# Up to This Point: Compiler

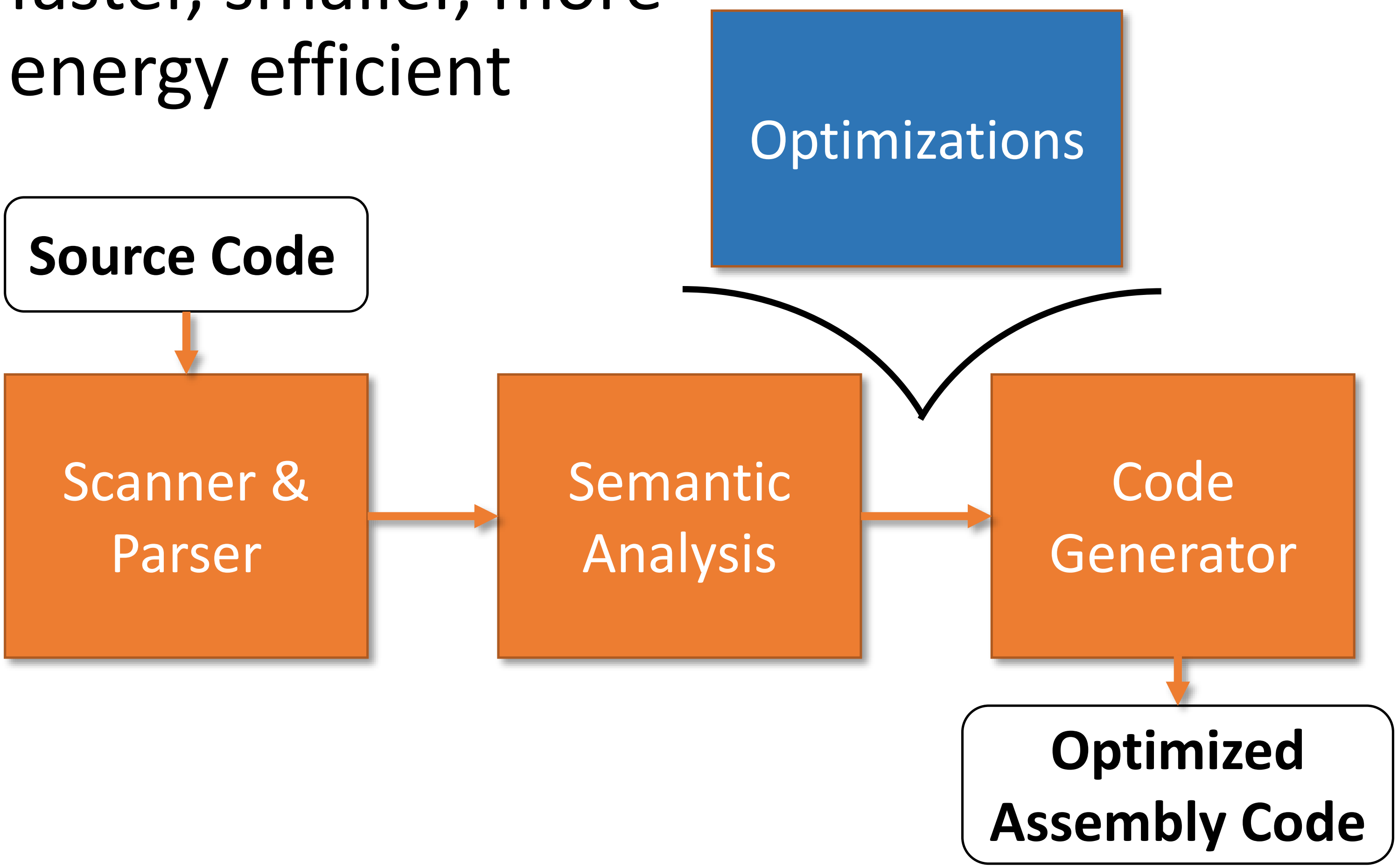
- We built a compiler!




- What's next?

# From Now On: Optimizing Compiler

- Optimize program: make programs faster, smaller, more energy efficient



# From Now On: Optimizing Compiler

- Transformations:
  - Move, remove, and add instructions
  - Or basic blocks, functions, variables

- Ensure: semantics remains the same
  - Task of program analysis
  - Apply transformation only when it's safe
  - Both regular and irregular conditions

# Optimization

- Previous Pass: Generates Control Flow Graph
- **Iterate:**
  - Control Flow Analysis
  - Data Flow Analysis
  - Transform Control Flow Graph
- Previous Pass: Generates Assembly Code

# Control Flow Analysis

- Construct basic blocks from Instruction-level CFG
- Find blocks that always execute before/after other blocks
- Keep track of structure of programs:
  - Conditionals
  - Loops
  - Function calls

# Data Flow Analysis

- Gathers information about values calculated at locations of interest within a function
- Within basic block: e.g., value numbering
  - Symbolic execution of the basic block
- Global: beyond basic block – how control flow affects the sets of data
  - Transfer function: OutSet = transfer(generated_set)
  - Confluence Operator: InSet = confluence(previous_set)

# Transformations: Peephole

- Within a single basic block:
  - Sequential code only
- Finds a better sequence of operations
- Examples:
  - (Local) Common subexpression elimination, constant folding
  - Algebraic simplifications
  - Dead code elimination

# Transformations: Intraprocedural

- Beyond a single basic block
  - Can use temporaries created in different basic blocks
  - Can move instructions beyond basic block boundaries
- Examples:
  - Global CSEE, constant folding
  - Dead store elimination
  - Loop optimizations
  - Invariant code motion

# Dataflow Analysis: Worklist Algorithm

```
Initialize InSet, OutSet;

Analyze the Entry Node:
    Compute InSet[EntryNode], OutSet[EntryNode]
Initialize Worklist (to Entry node or its successors)

while (Worklist != Empty) {
    Choose a node n in Worklist;
    Worklist = Worklist - { n };

    OldOutSet_n = OutSet[n]
    Compute InSet[n] and OutSet[n]
        • Use Use predecessor information
        • Gen/Kill Sets

    if (OldOutSet_n != OutSet[n])
        Update worklist
}
```

# Available Expressions

- An expression x+y is available at a point p if
  - every path from the initial node to p must evaluate x+y before reaching p,
  - and there are no assignments to x or y after the evaluation but before p.
- Available Expression information can be used to do global (across basic blocks) CSE
- If expression is available at use, no need to reevaluate it

# Available Expressions

- Expressions:
  - z = x op y
  - z = x
  - x cmp y
- Each basic block has
  - InSet- set of expressions available at start of block
  - OutSet - set of expressions available at end of block
  - GEN - set of expressions computed in the block
  - KILL - set of expressions killed in the block
- Compiler scans each basic block to derive GEN and KILL sets

# Available Expressions

Dataflow Equations:

- Forward Analysis: Starts from Entry of the function
- IN[entry] = AllEmpty
- IN[b] = OUT[b1] $\cap$ ... $\cap$ OUT[bn]
  - where b1, ..., bn are predecessors of b in CFG
- OUT[b] = (IN[b] - KILL[b]) U GEN[b]
- Result: system of equations

# Worklist Algorithm: Available Expressions

Initialize InSet, OutSet;

Analyze the Entry Node:
    Compute InSet[EntryNode], OutSet[EntryNode]
Initialize Worklist (to Entry node or its successors)

while (Worklist != Empty) {
    Choose a node **n** in Worklist;
    Worklist = Worklist - { **n** };

    OldOutSet_n = OutSet[**n**]
    Compute InSet[**n**] and OutSet[**n**]
- Use Use predecessor information
- Gen/Kill Sets

    if (OldOutSet_n != OutSet[n])
        Update Worklist
}

For node n
    OutSet[n] = AllExpressions;

InSet[EntryNode] = emptyset;
OutSet[EntryNode] = GEN[Entry];
Worklist= AllNodes - { Entry };

InSet[n] = AllExpressions;
for all nodes p in predecessors(n)
    InSet[n] = InSet[n] $\cap$ OutSet[p];

OutSet[n] = GEN[n] U (InSet[n] - KILL[n]);

for all nodes s in successors(n)
        Worklist = Worklist <-  s ;

# Worklist Algorithm: Available Expressions

Initialize InSet, OutSet;

For node n
    OutSet[n] = AllExpressions;

Analyze the Entry Node:
    Compute InSet[EntryNode], OutSet[EntryNode]
Initialize Worklist (to Entry node or its successors)

InSet[EntryNode] = emptyset;
OutSet[EntryNode] = GEN[Entry];
Worklist= AllNodes - { Entry };

```
while (Worklist != Empty) {
    Choose a node n in Worklist;
    Worklist = Worklist - { n };

    OldOutSet_n = OutSet[n]
    Compute InSet[n] and OutSet[n]
        • Use Use predecessor information
        • Gen/Kill Sets

    if (OldOutSet_n != OutSet[n])
        Update Worklist
}
```

```
InSet[n] = AllExpressions;
for all nodes p in predecessors(n)
    InSet[n] = InSet[n] ∩ OutSet[p];

OutSet[n] = GEN[n] U (InSet[n] - KILL[n]);
```

```
for all nodes s in successors(n)
        Worklist = Worklist <-  s ;
```

# Use of Analysis in Global CSEE

- Available Expression information can be used to do global CSE
- If expression is available at use, no need to reevaluate it

- Create a temporary variable t
- At computation site – assign t with expression:

  a = exp;

  t = a
- At use site – if expression is available replace it with t

# Examples

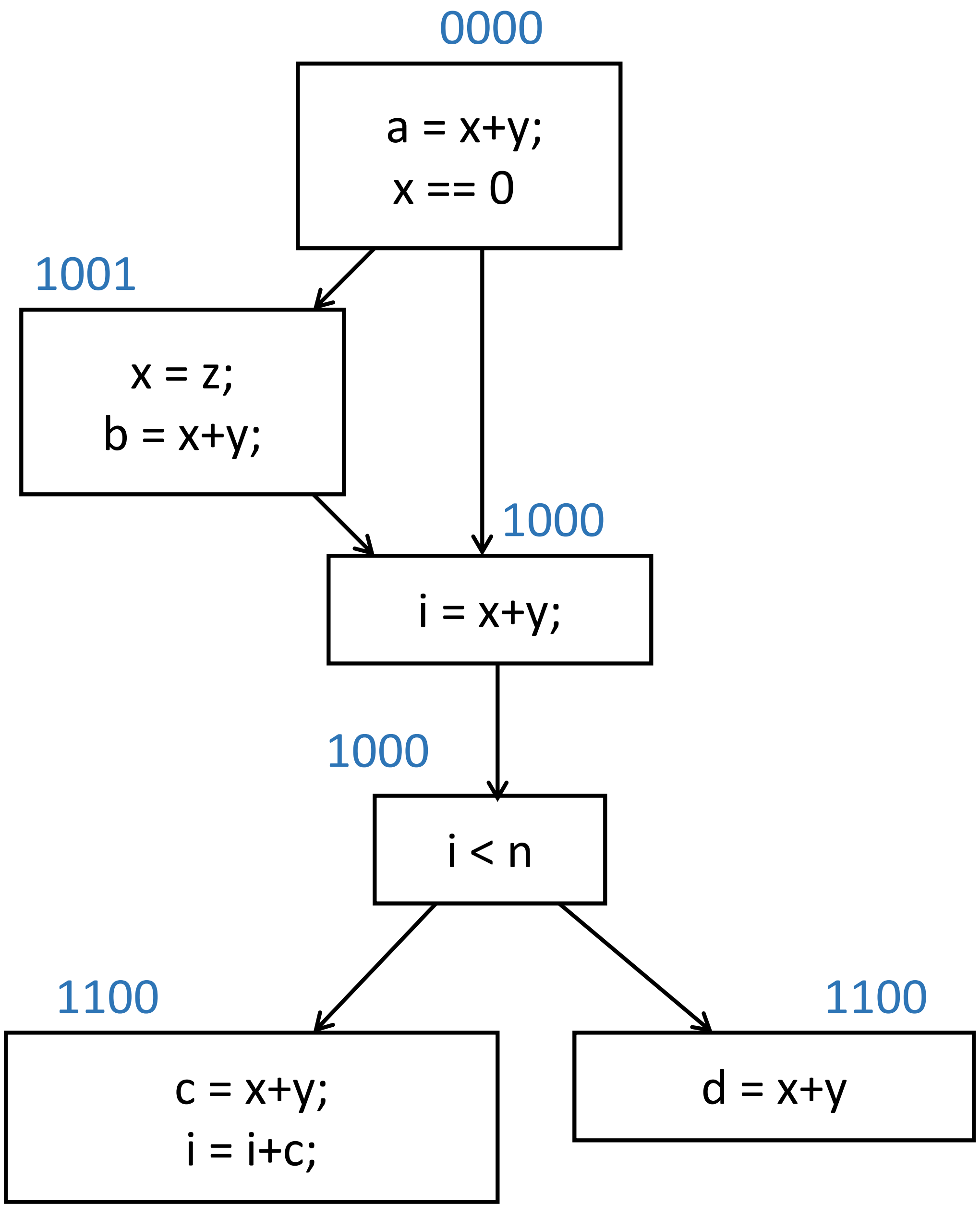
**Expressions**

**1: x+y**

**2: i<n**

**3: i+c**

**4: x==0**



# Global CSE Transform

**Expressions**

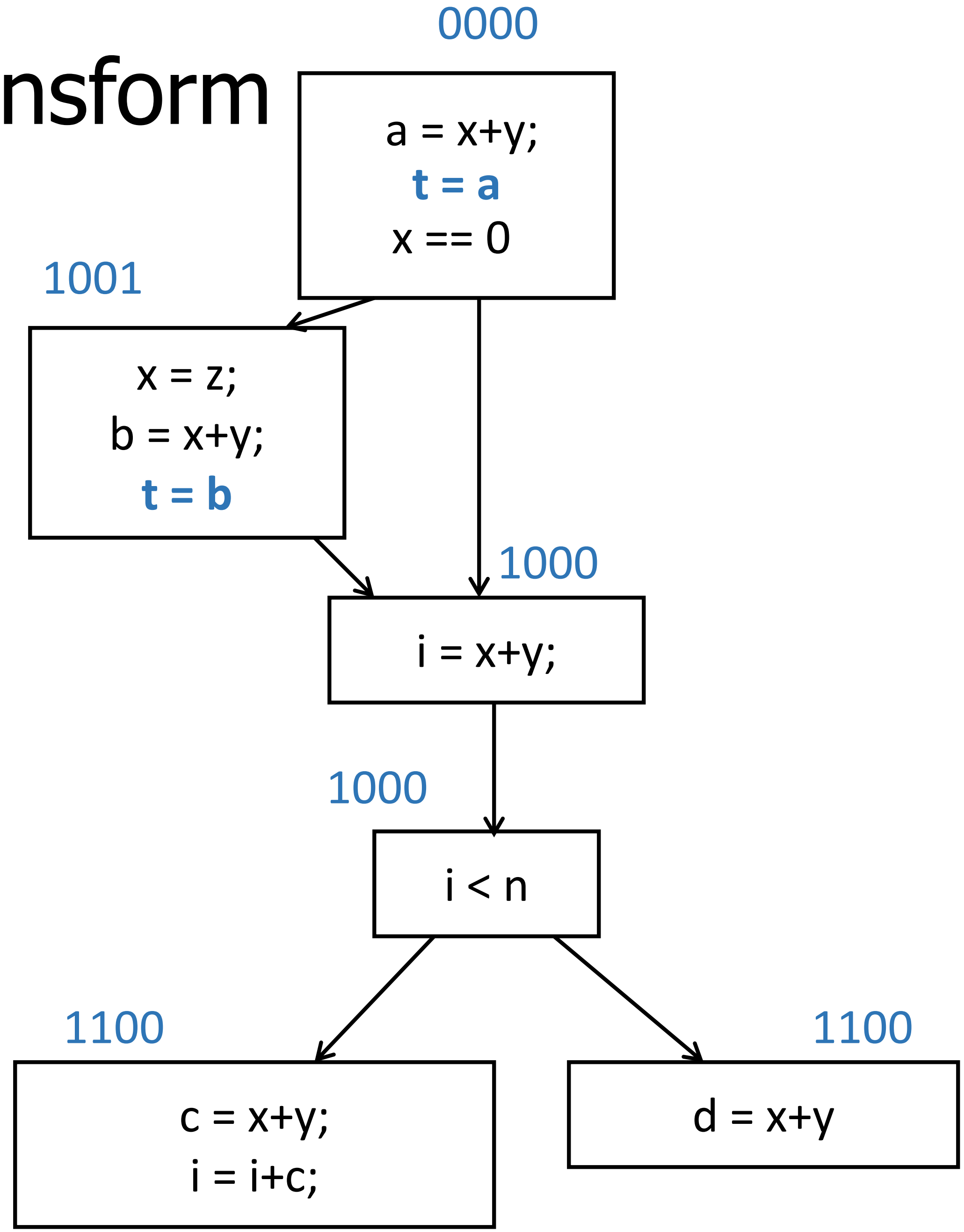
**1: x+y**

**2: i<n**

**3: i+c**

**4: x==0**

must use same temp
for CSE in all blocks



# Global CSE Transform

**Expressions**

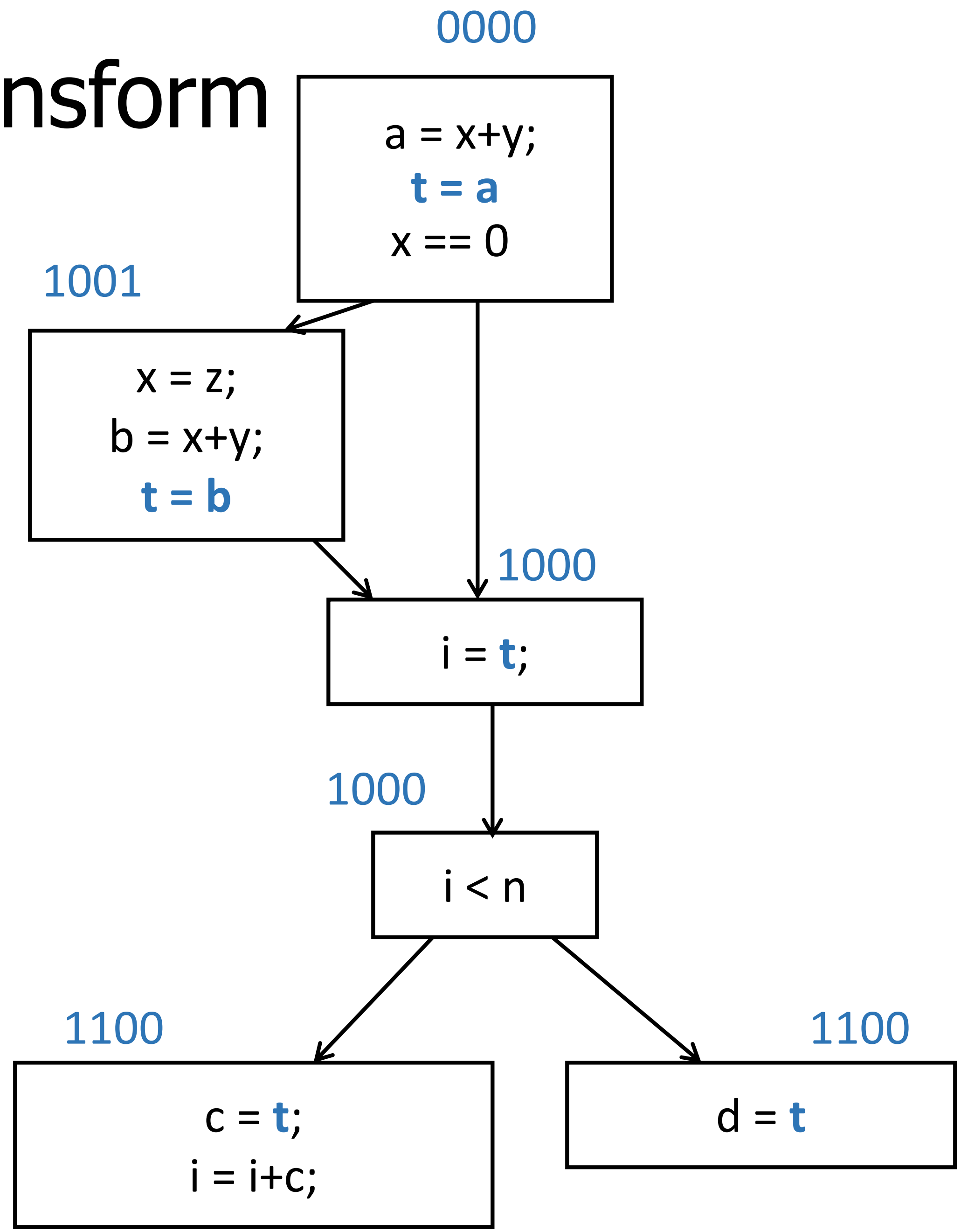
**1: x+y**

**2: i<n**

**3: i+c**

**4: x==0**

must use same temp
for CSE in all blocks



# Warm-up

```
void main ( ) {
   int a, b, c, d;
   a = 2; b = 3;
   c = 0;  d = 0;

   c = a + b;
   d = a + b;
}
```

# Globals

```
int a, b, c, d;

void main ( ) {
   a = 2 ;  b = 3;
   c = 0;  d = 0;

   c = a + b;
   d = a + b;
}
```

# Arrays

```
void main( ) {
    int a[10];
    int i, x;

    i = … ;
    a[i] = 1;
    a[i] = a[i] + 1;
}
```

# Algebraic Transformations

```
void main ( ) {
   int a, b, c, d;
   a = 2; b = 3;
   c = 0;  d = 0;

   c = a + b;
   d = a + 1 + b ;
}
```

# Grading Info

- Total: 45 points
- Testing: 50%
  - You implemented all required optimizations
  - Produce correct output for test cases
- Derby performance: 30%
- Documentation: 20%