CFG/SSA/LLVM Notes

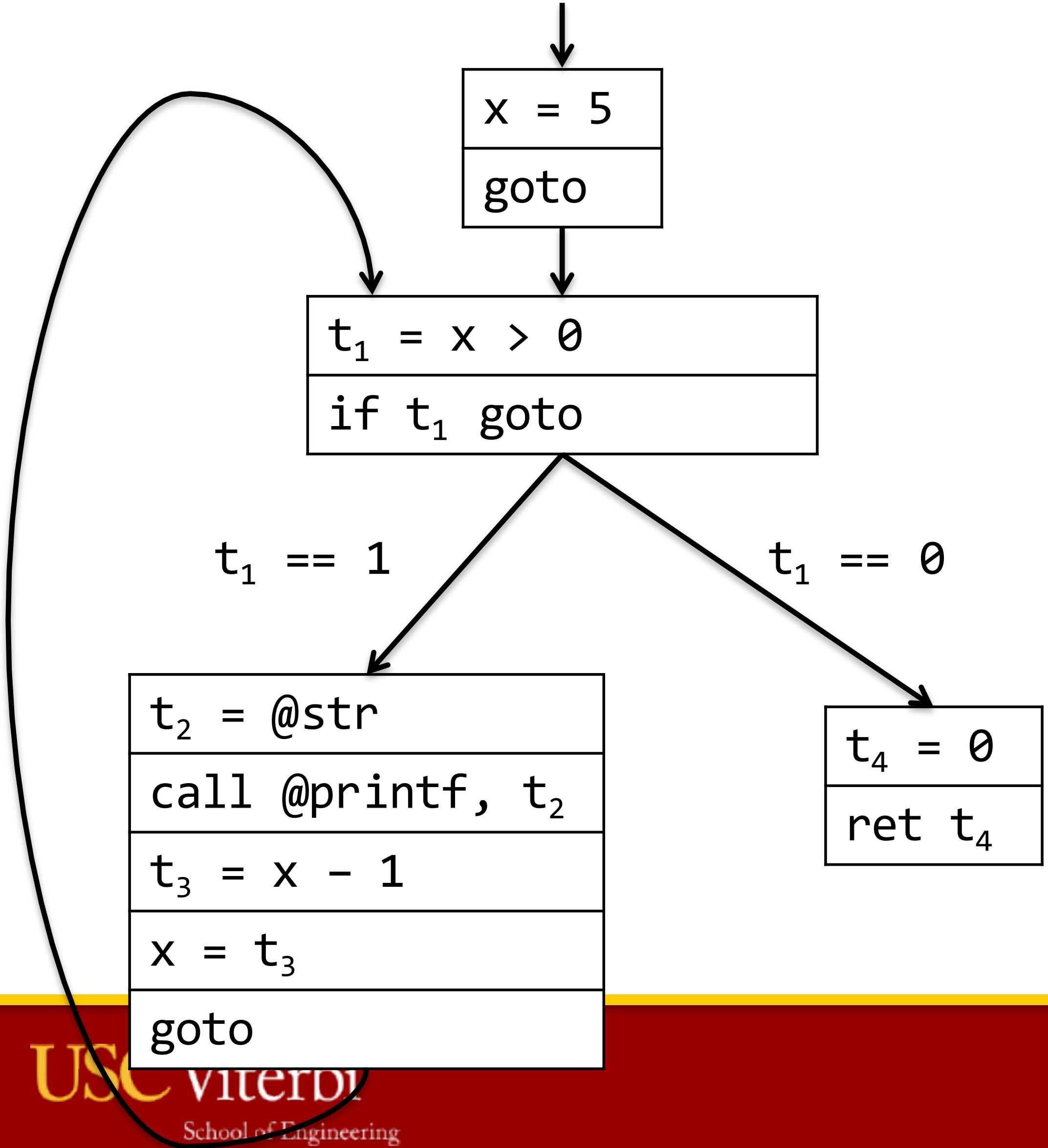
Lecturer: Sanjay Madhav

Control Flow Graph

- A ***control flow graph*** (CFG) is a hybrid IR
- In a CFG, sequences of linear code (without any jumps, gotos, branches, or the like) are called ***basic blocks***
- Basic blocks are usually represented by 3 address code (or similar)
- Jumps in control flow are represented by edges in the graph

Control Flow Graph, Cont’d

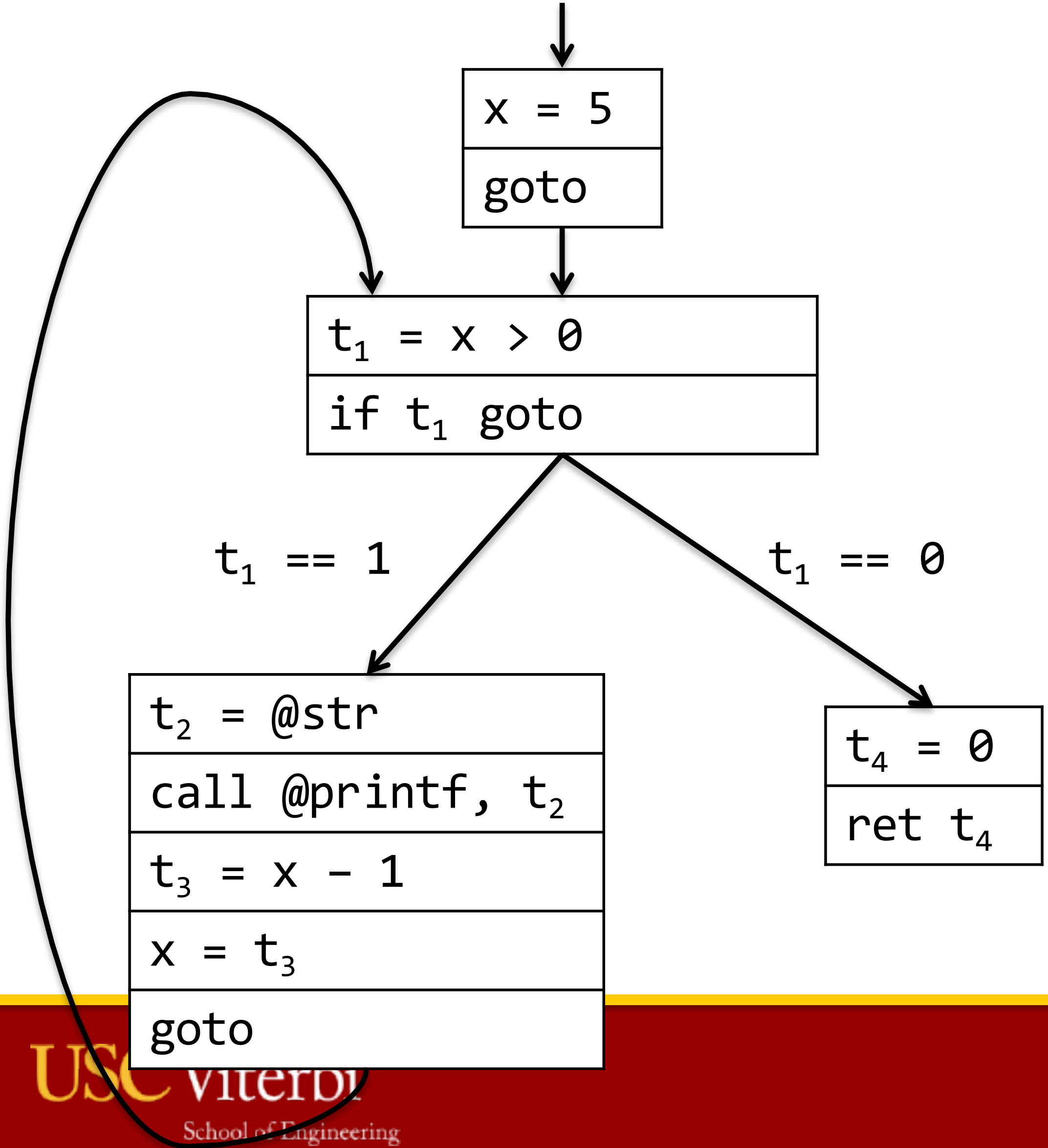
- Example of a CFG




```c
int main() {
    int x = 5;
    while (x > 0) {
        printf("Hello!");
        --x;
    }
    return 0;
}
```

Comparing CFG to 3 Address Code

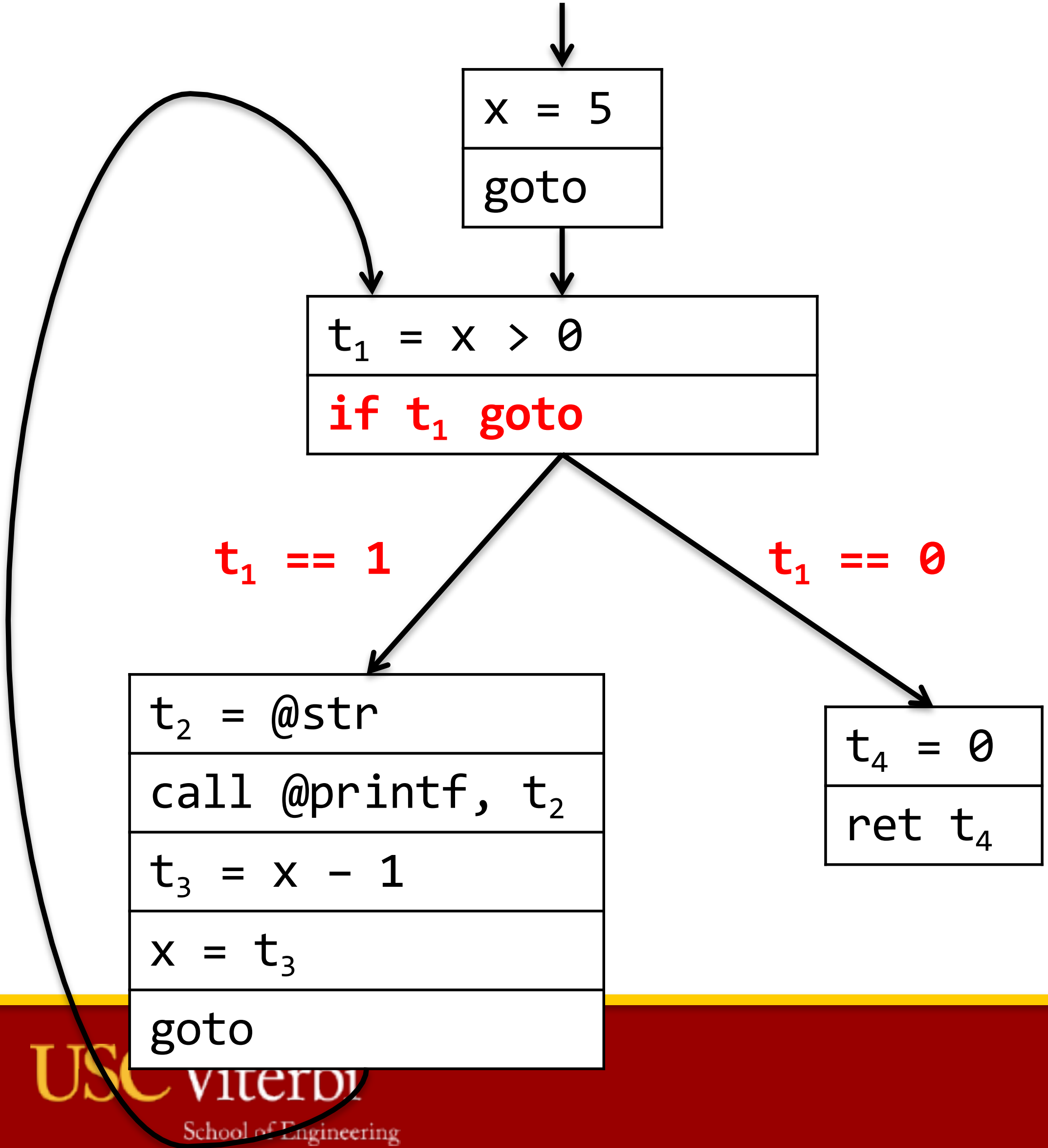
- Because of the “goto 2”, the “x = 5” must be in a separate block



1:	x = 5
2:	t_1 = x > 0
3:	if !t_1 goto 10
4:	t_2 = @str
5:	t_3 = @printf
6:	call t_3, t_2
7:	t_4 = x - 1
8:	x = t_4
9:	goto 2
10:	t_5 = 0
11:	ret t_5

Comparing CFG to 3 Address Code

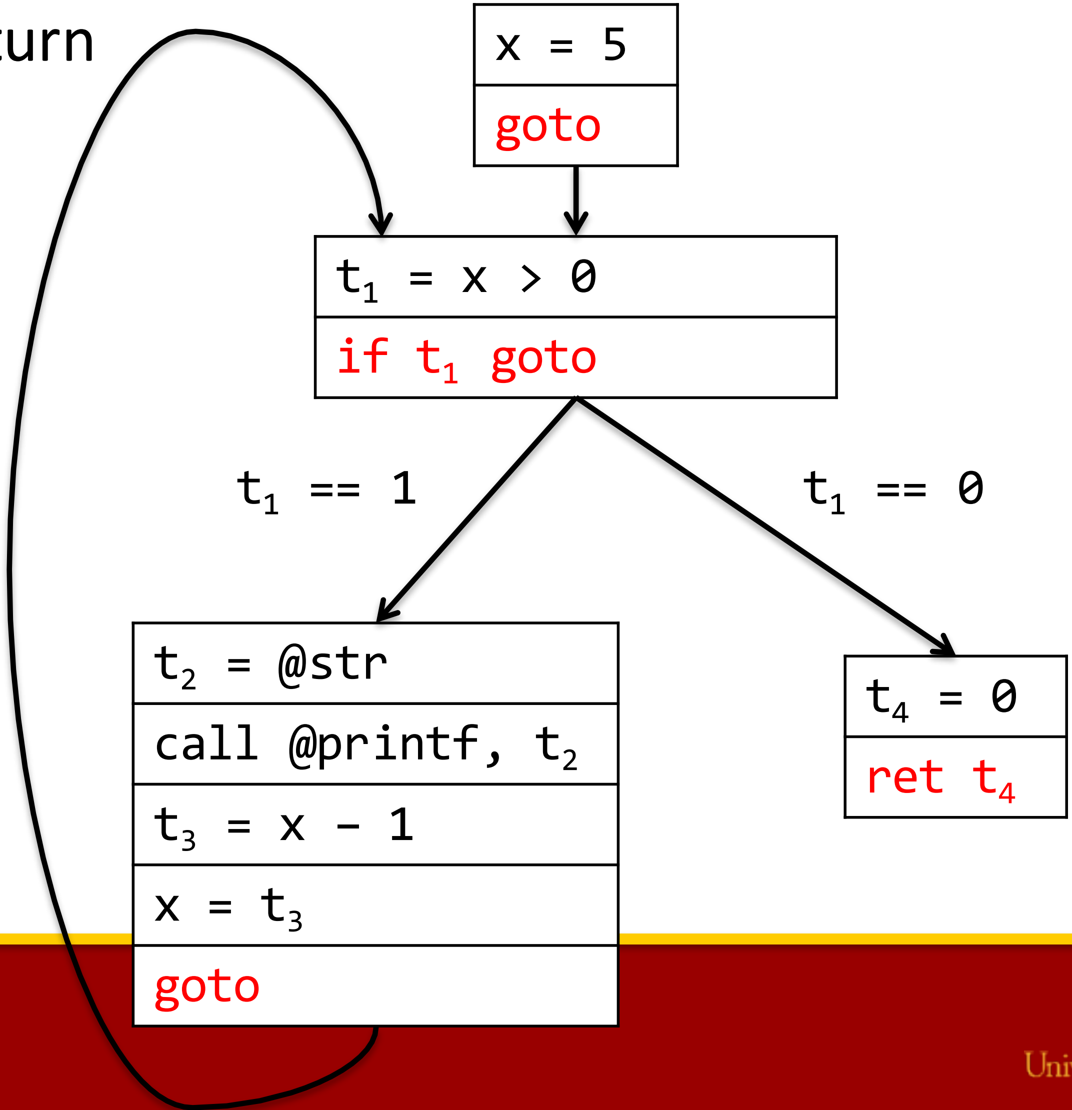
- The way I wrote the if/goto is slightly different, also



1:	x = 5
2:	t_1 = x > 0
3:	**if !t_1 goto 10**
4:	t_2 = @str
5:	t_3 = @printf
6:	call t_3, t_2
7:	t_4 = x – 1
8:	x = t_4
9:	goto 2
10:	t_5 = 0
11:	ret t_5

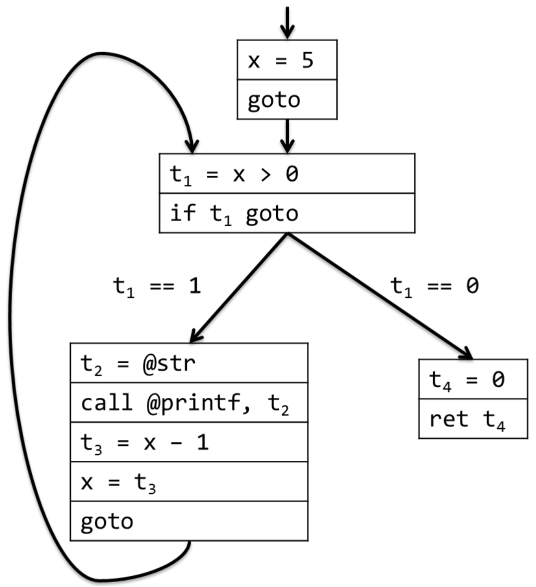
Control Flow Graph, Cont'd

- Each basic block *must* end with a ***terminator instruction***
- The terminator is either a branch (conditional or unconditional) or a function return



CFG Advantages/Disadvantages

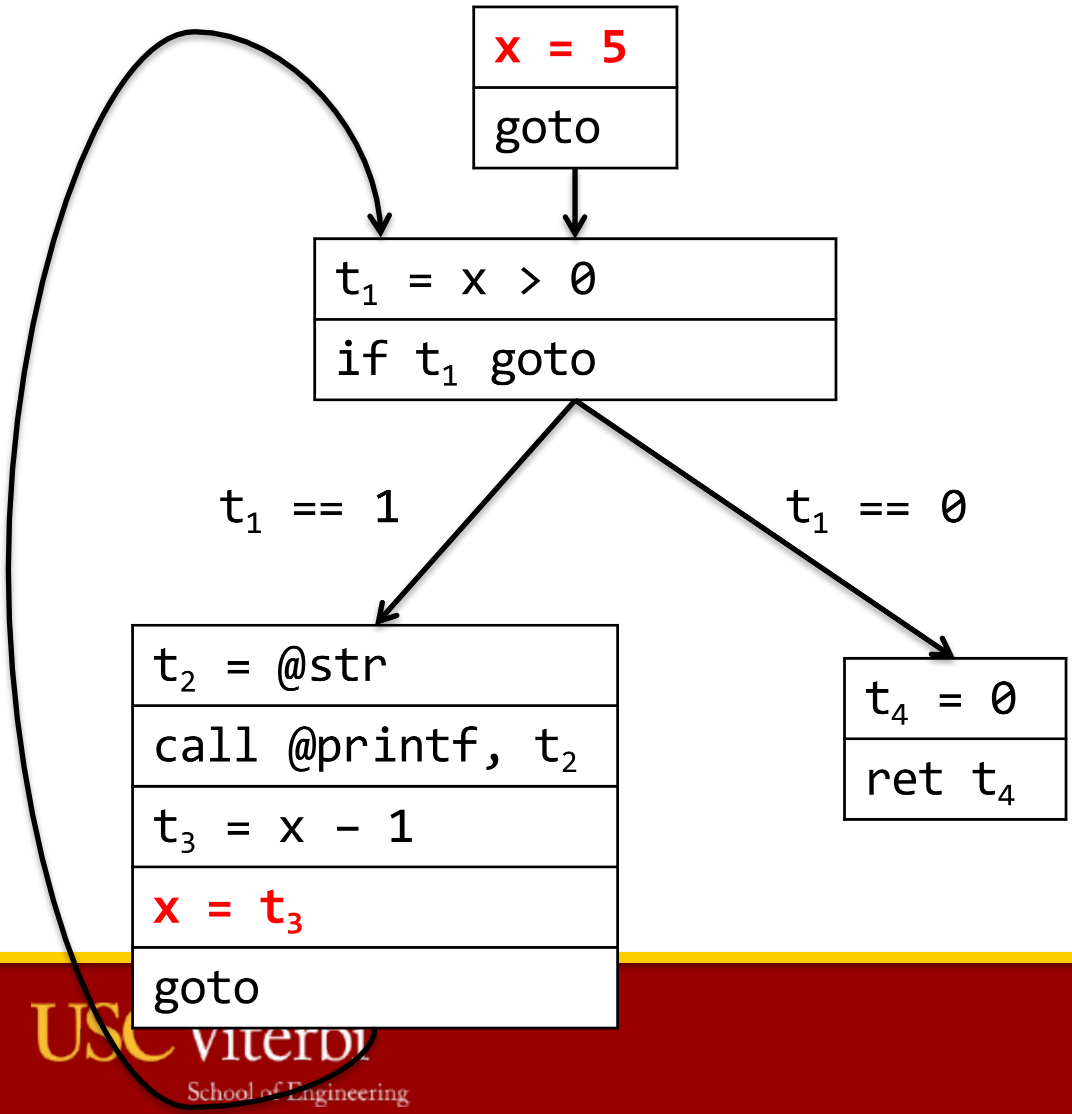
- Advantages:
  - Clearly represents control flow, which allows for loops to be optimized
  - Each basic block is guaranteed to have sequential code and so it can be aggressively optimized
- Disadvantages:
  - Cannot be generated at parse time
  - Requires the most amount of code to create, out of all the IR covered




```
int main() {
   int x = 5;
   while (x > 0) {
      printf("Hello!");
      --x;
   }
   return 0;
}
```

Static Single Assignment Form

- In static ***single assignment form*** (SSA) form, each variable in the IR can only have one assignment statement

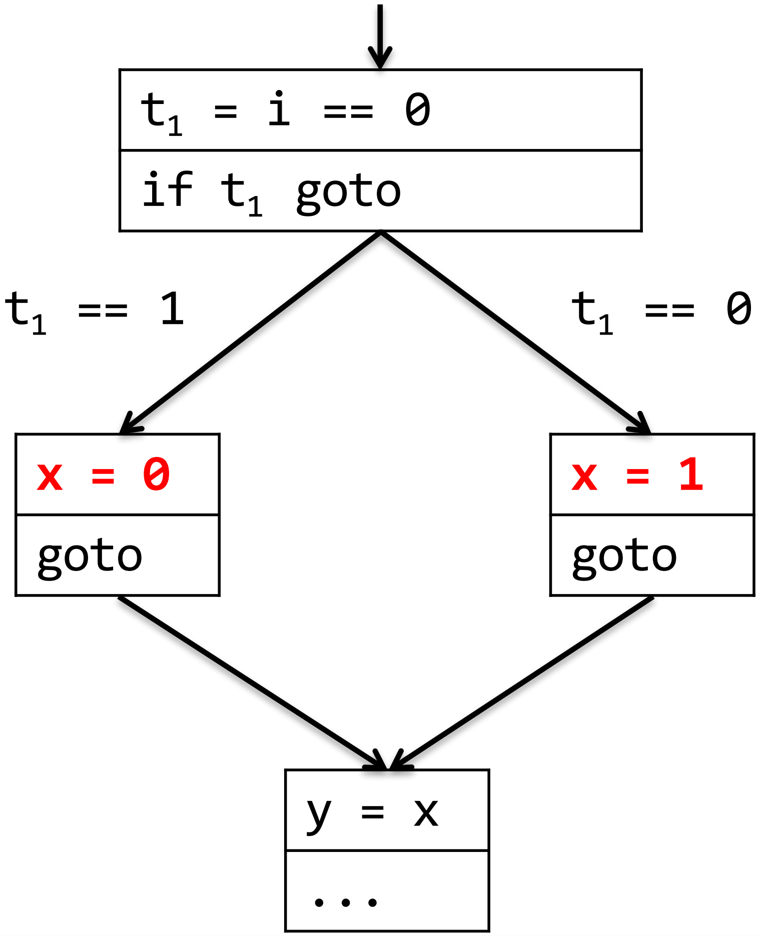



```
int main() {
    int x = 5;
    while (x > 0) {
        printf("Hello!");
        --x;
    }
    return 0;
}
```

NOT SSA FORM ☹

A Simpler Example

- Still not SSA form ☹

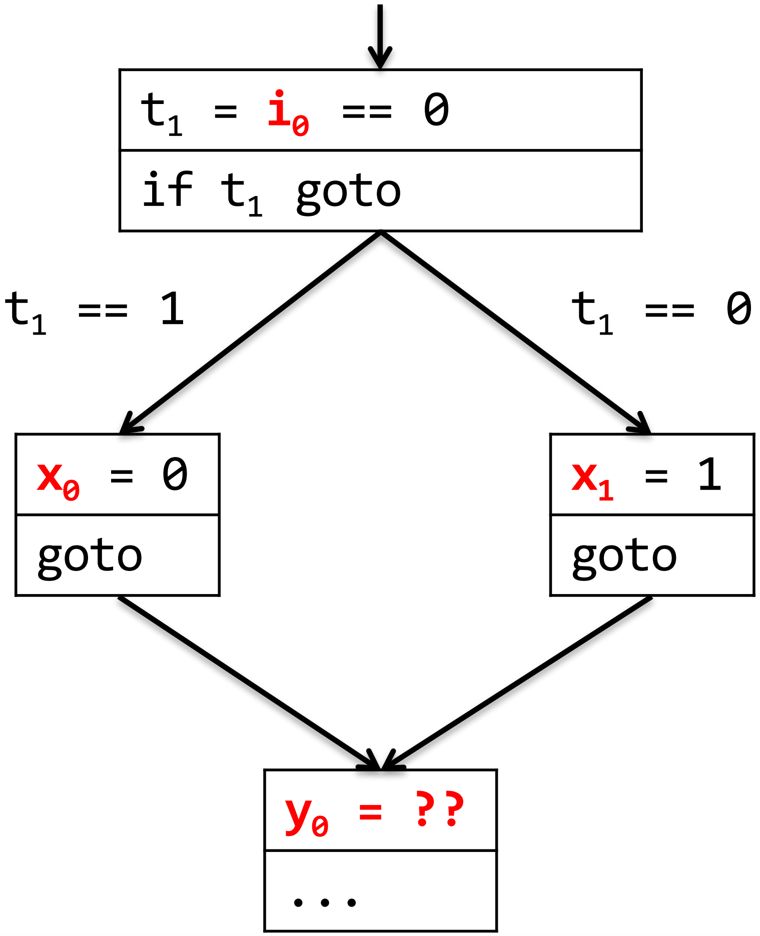



```
if (i == 0) {
   x = 0;
} else {
   x = 1;
}

y = x;
```

Converting to SSA

- We can add a subscript to each variable assignment to make it unique

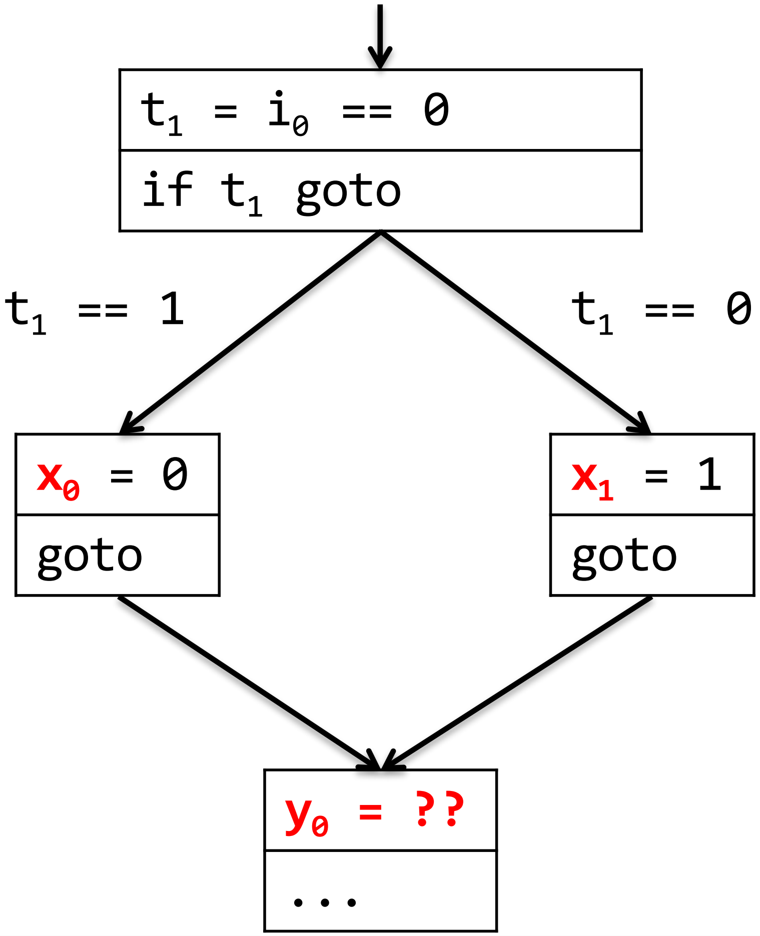



```
if (i == 0) {
   x = 0;
} else {
   x = 1;
}

y = x;
```

Converting to SSA

- ***Problem:*** Should y_0 be set to x_0 or x_1?

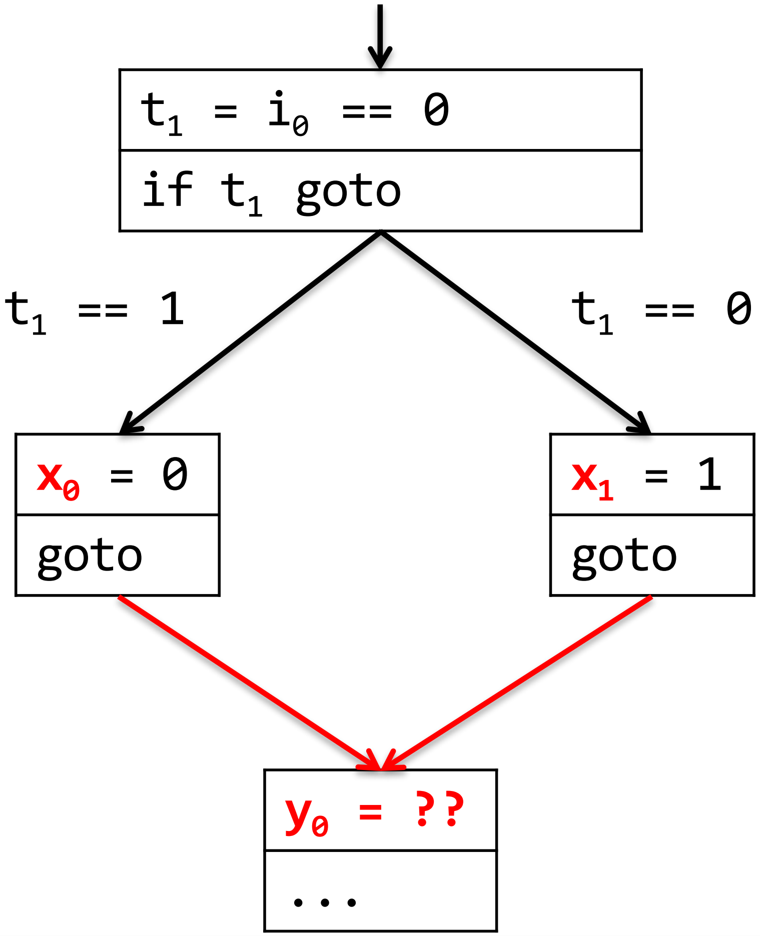



```
if (i == 0) {
    x = 0;
} else {
    x = 1;
}

y = x;
```

Converting to SSA

- ***Problem:*** Should y_0 be set to x_0 or x_1?
- ***It depends on which basic block we came from!***

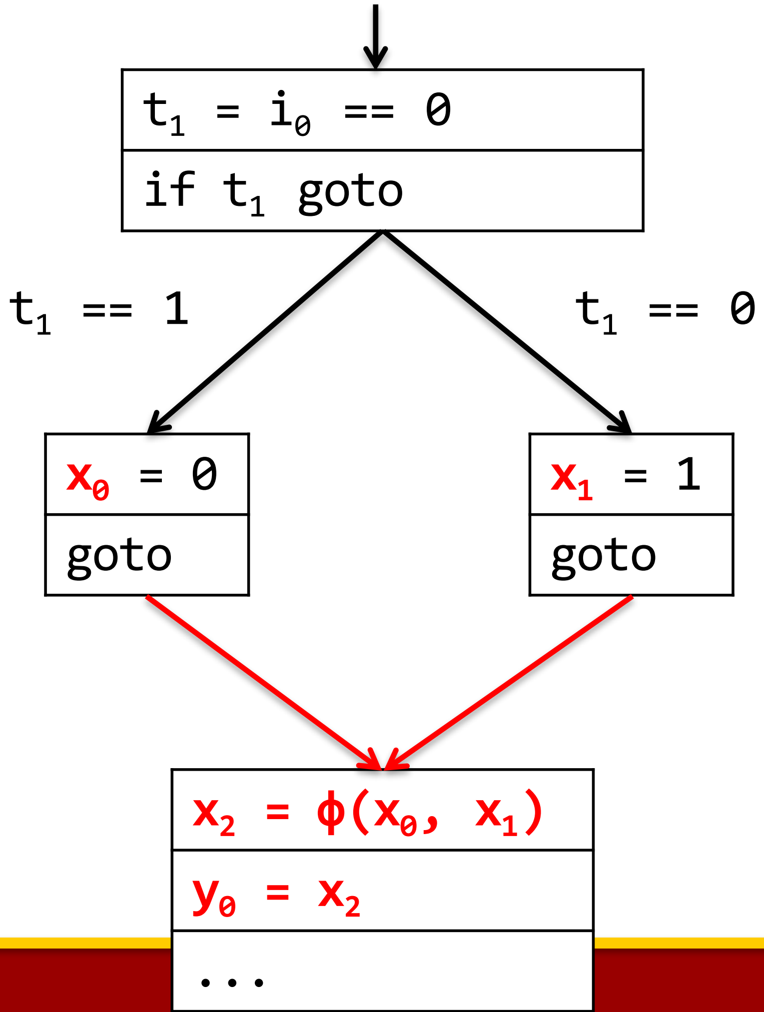



```
if (i == 0) {
    x = 0;
} else {
    x = 1;
}

y = x;
```

Converting to SSA – Phi Nodes

- A ***phi node*** (or ***ϕ-node***) is a special instruction that will select from multiple options based on the incoming edge in the CFG

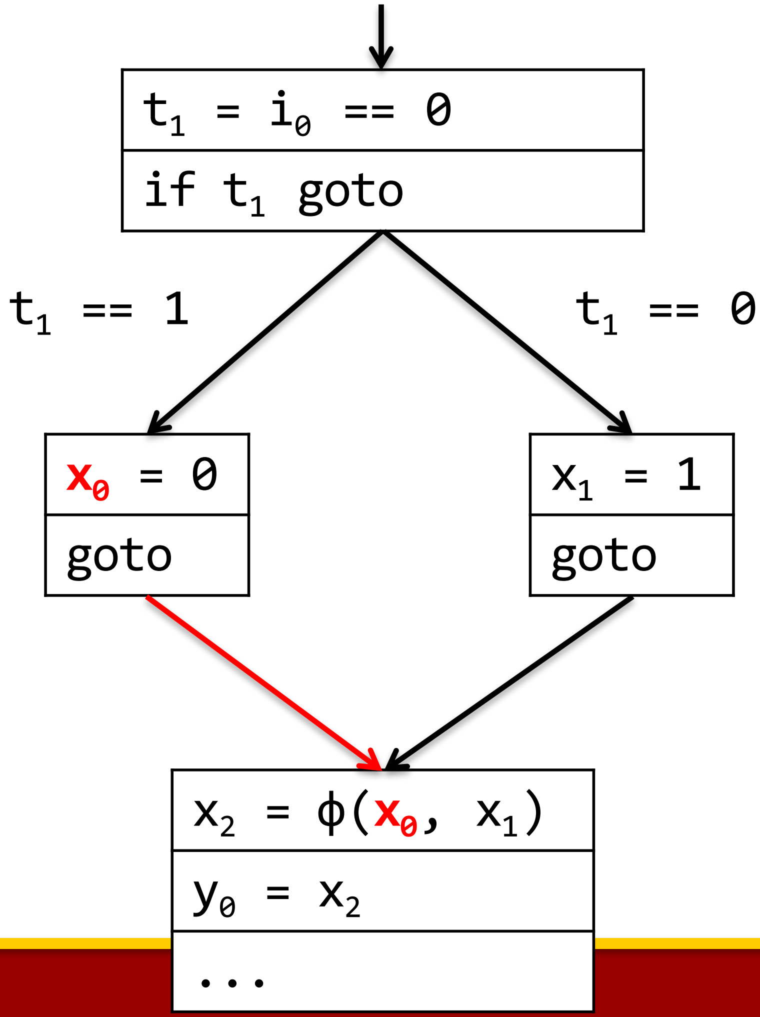



```
if (i == 0) {
   x = 0;
} else {
   x = 1;
}

y = x;
```

Converting to SSA – Phi Nodes

- If control flow comes from the ***left*** predecessor, then $x_2 = x_0$

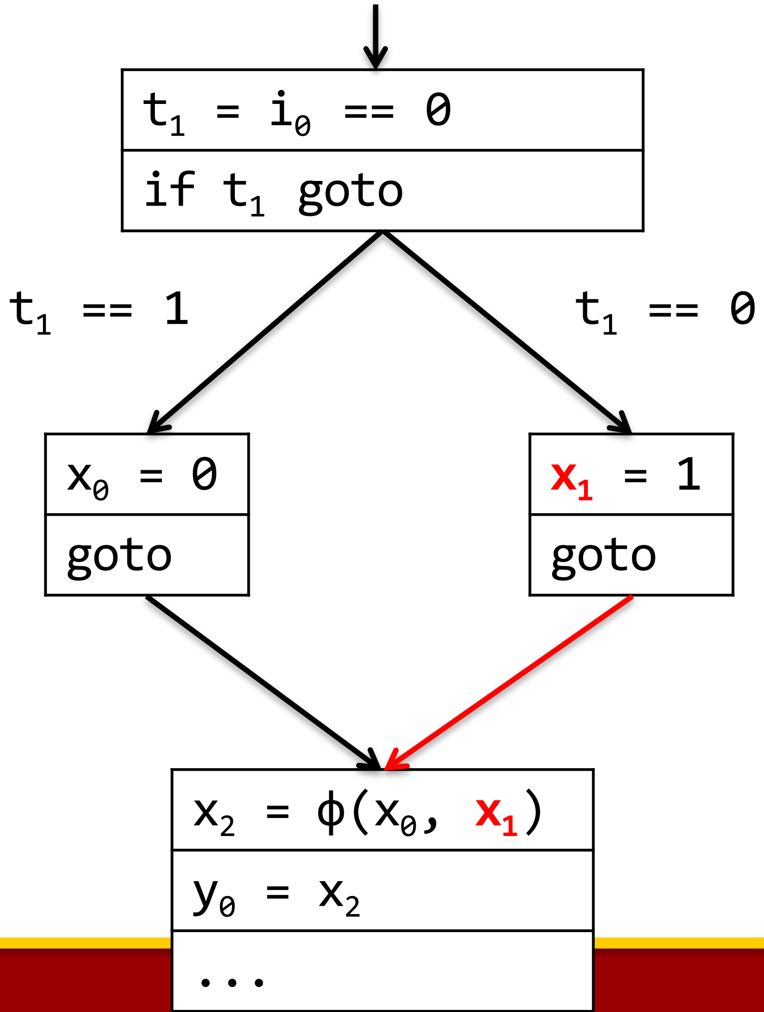



```
if (i == 0) {
   x = 0;
} else {
   x = 1;
}

y = x;
```

Converting to SSA – Phi Nodes

- If control flow comes from the ***right*** predecessor, then $x_2 = x_1$




```
if (i == 0) {
   x = 0;
} else {
   x = 1;
}

y = x;
```

Three Modes of the LLVM IR

- The LLVM IR can be used in three different ways:

1. As a text file on disk, that looks a lot like assembly
2. As a binary file on disk, which can be compiled to native code or interpreted
3. As a set of data structures loaded in memory, used by the compiler

Three Modes of the LLVM IR

- First, let's focus on #1

1. **As a text file on disk, that looks a lot like assembly**

2. As a binary file on disk, which can be compiled to native code or interpreted

3. As a set of data structures loaded in memory, used by the compiler

An Example

- Last time we had the following simple USC program:

```
int main() {
   int x = 5;
   while (x > 0) {
      printf("Hello!");
      --x;
   }
   return 0;
}
```

- Let’s look at it in LLVM IR…

In LLVM IR

```
; ModuleID = 'main'
@.str = private unnamed_addr constant [7 x i8] c"Hello!\00"
declare i32 @printf(i8*, ...)
define i32 @main() {
entry:
  br label %while.cond

while.cond:    ; preds = %while.body, %entry
  %x = phi i32 [ %dec, %while.body ], [ 5, %entry ]
  %cmp = icmp sgt i32 %x, 0
  %0 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %0, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:    ; preds = %while.cond
  %1 = i8* getelementptr inbounds ([7 x i8]* @.str, i32 0, i32 0)
  %2 = call i32 (i8*, ...)* @printf(%1)
  %dec = sub i32 %x, 1
  br label %while.cond

while.end:     ; preds = %while.cond
  ret i32 0
}
```

Some Basic Properties…

```
; ModuleID = 'main'
@.str = private unnamed_addr constant [7 x i8] c"Hello!\00"
declare i32 @printf(i8*, ...)
define i32 @main() {
entry:
  br label %while.cond

while.cond:    ; preds = %while.body, %entry
  %x = phi i32 [ %dec, %while.body ], [ 5, %entry ]
  %cmp = icmp sgt i32 %x, 0
  %0 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %0, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:    ; preds = %while.cond
  %1 = i8* getelementptr inbounds ([7 x i8]* @.str, i32 0, i32 0)
  %2 = call i32 (i8*, ...)* @printf(%1)
  %dec = sub i32 %x, 1
  br label %while.cond

while.end:     ; preds = %while.cond
  ret i32 0
}
```

Comments begin w/ semi-colon

Some Basic Properties…

```
; ModuleID = 'main'
@.str = private unnamed_addr constant [7 x i8] c"Hello!\00"
declare i32 @printf(i8*, ...)
define i32 @main() {
entry:
  br label %while.cond

while.cond:    ; preds = %while.body, %entry
  %x = phi i32 [ %dec, %while.body ], [ 5, %entry ]
  %cmp = icmp sgt i32 %x, 0
  %0 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %0, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:    ; preds = %while.cond
  %1 = i8* getelementptr inbounds ([7 x i8]* @.str, i32 0, i32 0)
  %2 = call i32 (i8*, ...)* @printf(%1)
  %dec = sub i32 %x, 1
  br label %while.cond

while.end:     ; preds = %while.cond
  ret i32 0
}
```

Function implementations are enclosed in braces, just like in C/C++

Some Basic Properties…

```
; ModuleID = 'main'
@.str = private unnamed_addr constant [7 x i8] c"Hello!\00"
declare i32 @printf(i8*, ...)
define i32 @main() {
entry:
  br label %while.cond

while.cond:    ; preds = %while.body, %entry
  %x = phi i32 [ %dec, %while.body ], [ 5, %entry ]
  %cmp = icmp sgt i32 %x, 0
  %0 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %0, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:    ; preds = %while.cond
  %1 = i8* getelementptr inbounds ([7 x i8]* @.str, i32 0, i32 0)
  %2 = call i32 (i8*, ...)* @printf(%1)
  %dec = sub i32 %x, 1
  br label %while.cond

while.end:     ; preds = %while.cond
  ret i32 0
}
```

Each basic block begins with the name (label) of the block, followed by comments referencing any predecessor blocks.

Some Basic Properties…

```
; ModuleID = 'main'
@.str = private unnamed_addr constant [7 x i8] c"Hello!\00"
declare i32 @printf(i8*, ...)
define i32 @main() {
entry:
  br label %while.cond

while.cond:    ; preds = %while.body, %entry
  %x = phi i32 [ %dec, %while.body ], [ 5, %entry ]
  %cmp = icmp sgt i32 %x, 0
  %0 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %0, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:    ; preds = %while.cond
  %1 = i8* getelementptr inbounds ([7 x i8]* @.str, i32 0, i32 0)
  %2 = call i32 (i8*, ...)* @printf(%1)
  %dec = sub i32 %x, 1
  br label %while.cond

while.end:     ; preds = %while.cond
  ret i32 0
}
```

Variables or ***virtual registers*** are always prefaced with a % sign.

Basic block names are prefaced with % only when used as an operand

Some Basic Properties…

```
; ModuleID = 'main'
@.str = private unnamed_addr constant [7 x i8] c"Hello!\00"
declare i32 @printf(i8*, ...)
define i32 @main() {
entry:
  br label %while.cond

while.cond:    ; preds = %while.body, %entry
  %x = phi i32 [ %dec, %while.body ], [ 5, %entry ]
  %cmp = icmp sgt i32 %x, 0
  %0 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %0, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:    ; preds = %while.cond
  %1 = i8* getelementptr inbounds ([7 x i8]* @.str, i32 0, i32 0)
  %2 = call i32 (i8*, ...)* @printf(%1)
  %dec = sub i32 %x, 1
  br label %while.cond

while.end:     ; preds = %while.cond
  ret i32 0
}
```

The language is ***strongly-typed***

i32 = 32-bit int
i1 = bool
i8 = char
i8* = pointer to char
7 x i8 = array of 7 characters

Some Basic Properties…

```
; ModuleID = 'main'
@.str = private unnamed_addr constant [7 x i8] c"Hello!\00"
declare i32 @printf(i8*, ...)
define i32 @main() {
entry:
  br label %while.cond

while.cond:    ; preds = %while.body, %entry
  %x = phi i32 [ %dec, %while.body ], [ 5, %entry ]
  %cmp = icmp sgt i32 %x, 0
  %0 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %0, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:    ; preds = %while.cond
  %1 = i8* getelementptr inbounds ([7 x i8]* @.str, i32 0, i32 0)
  %2 = call i32 (i8*, ...)* @printf(%1)
  %dec = sub i32 %x, 1
  br label %while.cond

while.end:     ; preds = %while.cond
  ret i32 0
}
```

It as phi nodes, which means SSA form.

But…

SSA Form and LLVM IR

- There are two ways data can be stored in LLVM IR:
  - In virtual registers, which must be in SSA form
  - On the stack/memory, which is ***not*** in SSA form (eg. you can write to the same memory address multiple times)

- For simplicity, in PA3 we'll use the stack for all declared variables, and virtual registers only for temporary computations

- We'll worry about SSA form in PA4

The Prior Example w/ the Stack…

```
; ModuleID = 'main'
@.str = private unnamed_addr constant [7 x i8] c"Hello!\00"
declare i32 @printf(i8*, ...)
define i32 @main() {
entry:
  %x = alloca i32
  store i32 5, i32* %x
  br label %while.cond

while.cond:      ; preds = %while.body, %entry
  %x1 = load i32* %x
  %cmp = icmp sgt i32 %x1, 0
  %0 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %0, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:      ; preds = %while.cond
  %1 = i8* getelementptr inbounds ([7 x i8]* @.str, i32 0, i32 0)
  %2 = call i32 (i8*, ...)* @printf(%1)
  %x2 = load i32* %x
  %dec = sub i32 %x2, 1
  store i32 %dec, i32* %x
  br label %while.cond

while.end:       ; preds = %while.cond
  ret i32 0
}
```

LLVM Instructions of Note

- We'll only cover the ones you'll need to use in this class

- Most instructions have a lot of optional parameters, but I've pared it down to the bare minimum

- Full Language Reference: http://llvm.org/docs/LangRef.html

alloca Instruction

- Allocates memory on the stack and returns a pointer to the memory

- Syntax:

```
<result> = alloca <type> [,<ty> <NumElements>]
```

- Examples:

```
%x = alloca i32 ; Allocates one 32-bit value

%y = alloca i32, i32 5 ; Allocates an array of 5 i32s
```

store Instruction

- Stores a value into a memory address

- Syntax:

```
store <type> <value>, <ty>* <pointer>
```

- Example:

```
%x = alloca i32 ; Allocates one 32-bit value
store i32 20, i32* %x ; Store 20 in the address
```

load Instruction

- Read data from memory

- Syntax:

```
<result> = load <ty>* <pointer>
```

- Example:

```
%x = alloca i32 ; Allocates one 32-bit value
store i32 20, i32* %x ; Store 20 in the address
%val = load i32* %x ; Loads *x (20) into val
```

Binary Operators

- All of the binary operators follow essentially the same syntax:

```
<result> = <instr> <ty> <op1>, <op2>
```

- Operators of note:
  - `add` – Integer addition
  - `sub` – Integer subtraction
  - `mul` – Integer multiplication
  - `sdiv` – Signed integer division
  - `srem` – Signed integer remainder/modulus

- Examples:

```
%a = add i32 %x, %y ; %a = %x + %y
%b = sub i32 %a, 5 ; %b = %a - 5
```

sext Instruction

- Signed extend from a smaller integral type to a larger integral type

- Syntax:

```
<result> = sext <ty> <value> to <ty2>
```

- Example:

```
%x = sext i8 %a to i32 ; Extend from 8 to 32 bits
```

trunc Instruction

- Truncate from a larger integral type to a smaller integral type

- Syntax:

```
<result> = trunc <ty> <value> to <ty2>
```

- Example:

```
%x = trunc i32 %a to i8 ; Truncate from 32 to 8 bits
```

icmp Instruction

- Compares two integral values (or pointers), and returns a bool result of the comparison

- Syntax:

```
<result> = icmp <cond> <ty> <op1>, <op2>
```

…where <cond> is the condition such as eq, ne, …

- Examples:

```
%b = icmp eq i32 %x, %y ; %b = %x == %y

%c = icmp sgt i32 %x, %y ; %c = %x > %y
```

ret Instruction

- Returns from a function. This is a terminator instruction. The type returned must match the function signature
- Syntax:

```
ret <type> <value>
```

or

```
ret void
```

- Examples:

```
ret i32 %x ; Returns the 32-bit integer %x

ret void ; Returns void
```

br Instruction

- Branch – can be either conditional or unconditional. This is a terminator instruction.

- Syntax:

```
br label <dest>
```

or

```
br i1 <cond>, label <iftrue>, label <iffalse>
```

- Examples:

```
br label %bb0 ; Unconditional jump to bb0

; (%x ? goto %bb0 : goto %bb1)
br i1 %x, label %bb0, label %bb1
```

phi Instruction

- Used for SSA form phi nodes

- Syntax:

```
<result> = phi <ty> [ <val0>, <label0>], ...
```

- Example:

```
; %x = 20 if coming from %bb0, or %a if from %bb1
%x = phi i32 [20, %bb0], [%a, %bb1]
```

getelementptr Instruction

- Used to get the address of an element in arrays and structs (among other things)

- So confusing that there even is an entire doc on it: http://llvm.org/docs/GetElementPtr.html

- Don't worry about the syntax!

LLVM Intrinsics

- LLVM also supports some slightly higher level intrinsic functions, such as some useful Standard C library functions:
  - llvm.memcpy
  - llvm.memset
  - llvm.sqrt
  - llvm.pow
  - llvm.ceil/floor

The Prior Example w/ the Stack…

```
; ModuleID = 'main'
@.str = private unnamed_addr constant [7 x i8] c"Hello!\00"
declare i32 @printf(i8*, ...)
define i32 @main() {
entry:
  %x = alloca i32
  store i32 5, i32* %x
  br label %while.cond

while.cond:      ; preds = %while.body, %entry
  %x1 = load i32* %x
  %cmp = icmp sgt i32 %x1, 0
  %0 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %0, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:      ; preds = %while.cond
  %1 = i8* getelementptr inbounds ([7 x i8]* @.str, i32 0, i32 0)
  %2 = call i32 (i8*, ...)* @printf(%1)
  %x2 = load i32* %x
  %dec = sub i32 %x2, 1
  store i32 %dec, i32* %x
  br label %while.cond

while.end:       ; preds = %while.cond
  ret i32 0
}
```

```
int main() {
    int x = 5;
    while (x > 0) {
        printf("Hello!");
        --x;
    }
    return 0;
}
```

Three Modes of the LLVM IR

- Ok, what about some stuff on #3?

1. As a text file on disk, that looks a lot like assembly

2. As a binary file on disk, which can be compiled to native code or interpreted

3. **As a set of data structures loaded in memory, used by the compiler**

llvm::Module

- The Module class corresponds to all code in one object file

- Contains things such as:
  - List of all functions in the Module
  - List of all global variables
  - LLVM's internal SymbolTable (that you should pretty much never touch)

llvm::Value

- *Most* of the types you'll be using inherit from `llvm::Value`
- One of the features `llvm::Value` provides is a custom RTTI implementation, via:

```
// isa returns true if value is-a pointer to Type
isa<Type>(value)

// Returns pointer to Type if value is-a pointer to Type
// Otherwise, returns nullptr
dyn_cast<Type>(value);

// Like dyn_cast, except it asserts if the cast fails
cast<Type>(value);
```

llvm::Function

- Encapsulates a function

- Inherits (eventually) from llvm::Value

- Allows you to do things such as:
  - Get the entry block of the function
  - Iterate over all of the basic blocks in a function
  - Access/iterate over the arguments to the function

llvm::BasicBlock

- Corresponds to a basic block

- Inherits (eventually) from llvm::Value

- Allows you to do things such as:
  - Iterate over all of the linear instructions in the basic block
  - Get the terminator instruction

llvm::Instruction

- Every instruction inherits from this

- Since llvm::Instruction inherits from llvm::Value, every instruction can also be treated as a value

- This makes it really simple to pass the result of an instruction as an operand of another instruction

IRBuilder

- IRBuilder is absolutely your best friend when generating LLVM IR
- It has a factory method to create every instruction, which prevents you from having to write out the IR in text form

- Example (from some of the provided code):

```
{
    IRBuilder<> build(ctx.mBlock);
    // We can assume it WILL be an i32 here
    // since it'd have been zero-extended otherwise
    lhsVal = build.CreateICmpNE(lhsVal, ctx.mZero, "tobool");
    build.CreateCondBr(lhsVal, rhsBlock, endBlock);
}
```