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!"ера);
        --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: t1 = x > 0
3: if !t1 goto 10
4: t2 = @str
5: t3 = @printf
6: call t3, t2
7: t4 = x - 1
8: x = t4
9: goto 2
10: t5 = 0
11: ret t5
```
Comparing CFG to 3 Address Code

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

```
1: x = 5
2: t1 = x > 0
3: if !t1 goto 10
4: t2 = @str
5: t3 = @printf
6: call t3, t2
7: t4 = x - 1
8: x = t4
9: goto 2
10: t5 = 0
11: ret t5
```

```
x = 5
goto

| t1 = x > 0 |
| if t1 goto |

| t1 == 1 |
| t1 == 0 |

| t2 = @str |
call @printf, t2 |
| t3 = x - 1 |
x = t3 |
goto |

| t4 = 0 |
| ret t4 |
```
• Each basic block must end with a terminator instruction
• The terminator is either a branch (conditional or unconditional) or a function return

```
x = 5
if t_1 goto
```

```
t_1 = x > 0
if t_1 goto
```

```
t_1 == 1
```

```
t_2 = @str
call @printf, t_2
```

```
t_3 = x - 1
x = t_3
```

```
t_4 = 0
ret t_4
```

```
t_1 == 0
```

```
x = 5
goto
== 1
```

```
x = 0
goto
== 0
```
**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
Static Single Assignment Form

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

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

**NOT SSA FORM 😞**
A Simpler Example

• Still not SSA form 😞

```java
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;
```

\[
t_1 = i_0 == 0 \\
if \ t_1 \ \text{goto}
\]

\[
x_\theta = 0 \\
goto
\]

\[
x_1 = 1 \\
goto
\]

\[
x_2 = \phi(x_\theta, x_1) \\
y_\theta = x_2 \\
\ldots
\]
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;
}
```
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:

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

• Let’s look at it 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 %x, 1

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
}
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 %x, 1

  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 %x, 1
  }

  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
}
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 %x

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](http://llvm.org/docs/LangRef.html)
alloca Instruction

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

• Syntax:
  \<result> = \textit{alloca} \ <type> [ , \<ty> \ <NumElements>] \n
• Examples:
  %x = \textit{alloca} \ i32 ; \textit{Allocates one 32-bit value}

  %y = \textit{alloca} \ i32, i32 5 ; \textit{Allocates an array of 5 i32s}
store Instruction

• Stores a value into a memory address

• Syntax:
  \texttt{store} \texttt{<type>} \texttt{<value>}, \texttt{<ty>}* \texttt{<pointer>}

• Example:
  %x = \texttt{alloca} \texttt{i32} ; Allocates one 32-bit value
  \texttt{store} \texttt{i32} 20, \texttt{i32}* %x ; Store 20 in the address
### load Instruction

- Read data from memory

- Syntax:
  \[
  \text{<result> = load <ty>* <pointer>}
  \]

- Example:
  \[
  \%
  \text{x = alloca i32 ; Allocates one 32-bit value}
  \%
  \text{store i32 20, i32* \%x ; Store 20 in the address}
  \%
  \text{val = load i32* \%x ; Loads *x (20) into val}
  \]
Binary Operators

• All of the binary operators follow essentially the same syntax:
  \( \text{result} = \text{instr} \ <\text{ty}> \ <\text{op1}>, \ <\text{op2}> \)

• 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

USC Viterbi
School of Engineering

University of Southern California
sext Instruction

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

• Syntax:
  \[ \text{result} = \text{sext } \text{ty} \ \text{value} \ \text{to ty2} \]

• 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:
  \[ \text{<result> = \text{trunc} <ty> <value> to <ty2>} \]

• Example:
  \[ \%x = \text{trunc i32} \%a \text{ to i8} ; \text{Truncate from 32 to 8 bits} \]
icmp Instruction

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

• Syntax:
\[ <\text{result}> = \text{icmp} \ <\text{cond}> \ <\text{ty}> \ <\text{op1}>, \ <\text{op2}> \]
...where \(<\text{cond}>\) is the condition such as eq, ne, ...

• Examples:
\[ \%b = \text{icmp} \ \text{eq} \ i32 \ \%x, \ \%y ; \ \%b = \%x == \%y \]
\[ \%c = \text{icmp} \ \text{sgt} \ i32 \ \%x, \ \%y ; \ \%c = \%x > \%y \]
**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:
  
  \[ \text{br} \ \text{label} \ \langle \text{dest} \rangle \]

  or

  \[ \text{br} \ \text{i1} \ \langle \text{cond} \rangle, \ \text{label} \ \langle \text{iftrue} \rangle, \ \text{label} \ \langle \text{iffalse} \rangle \]

- Examples:
  
  \[ \text{br} \ \text{label} \ \%bb0 \ \text{; Unconditional jump to bb0} \]

  \[ \text{; (x ? goto \%bb0 : goto \%bb1)} \]

  \[ \text{br} \ \text{i1} \ \%x, \ \text{label} \ \%bb0, \ \text{label} \ \%bb1 \]
phi Instruction

• Used for SSA form phi nodes

• Syntax:
  \[
  \text{<result>} = \text{phi} \ \text{<ty>} \ [ \text{<val0>}, \text{<label0>}], ...}
  \]

• 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*, ...)
declare i32 @main()

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
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:

  ```cpp
  // 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):
```c
{ 
    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);
}
```