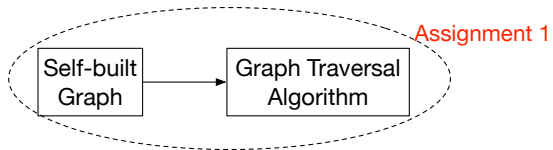


# Week 4 LLVM Compiler and Its Intermediate Representation

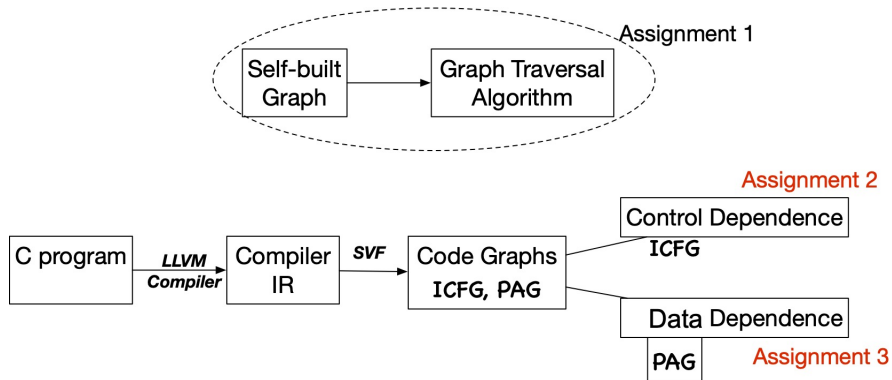
Yulei Sui

University of Technology Sydney, Australia

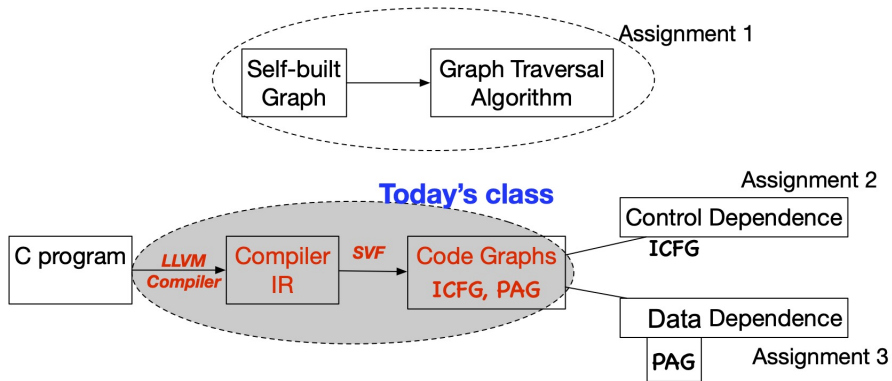
# Where We Are Now and Today's Class



# Where We Are Now and Today's Class



# Where We Are Now and Today's Class



# What is LLVM ?

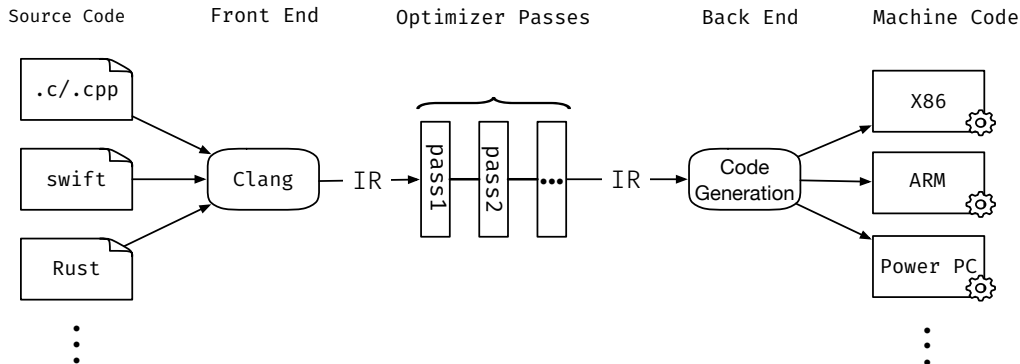
**Low-Level Virtual Machine (LLVM) compiler infrastructure is a collection of compiler and tool-chain technologies.**

- Originally started in 2000 from UIUC. An **open-source project** and supported and contributed by a range of high-tech companies such as Apple, Google, Intel, ARM.
- Modern compiler infrastructure can be used to develop a **front-end for any programming language** and a **back-end for any instruction set architecture**.
- A set of **reusable software modules** to quickly design your own compiler or software tool chains.
- **Language-independent intermediate representation (IR)** used for a variety of purposes, such as compiler optimizations, static analysis and bug detection.
- **More information on LLVM's website:** <https://llvm.org/>

# Why Learn LLVM or Learn Compilers in General?

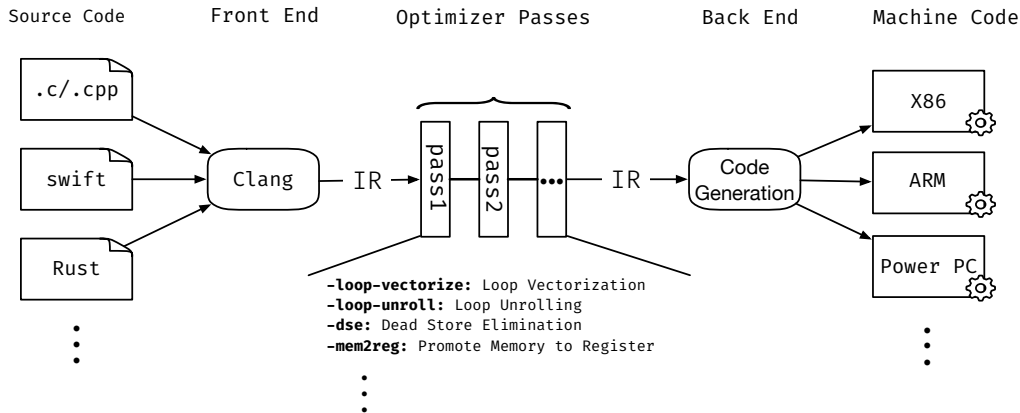
- An essential part of the standard curriculum in computer science.
- One of the most complex system for building virtually all other software.
- Sharpen your software design and implementation skills.
- Widely used by many major software companies. In-demand skills and competitive salaries in job market.

# LLVM's Architecture



\*IR: Human-readable LLVM IR (.ll files) or dense 'bitcode' binary representation (.bc files)

# LLVM's Architecture



\*IR: Human-readable LLVM IR (.ll files) or dense 'bitcode' binary representation (.bc files)



# LLVM Intermediate Representation (IR)

**LLVM IR is LLVM's code representation which is generated by its front-end clang when compiling a program** (<https://llvm.org/docs/LangRef.html>)

- **Language independent.** Not machine code, but one step just above assembly

# LLVM Intermediate Representation (IR)

**LLVM IR is LLVM's code representation which is generated by its front-end clang when compiling a program** (<https://llvm.org/docs/LangRef.html>)

- **Language independent.** Not machine code, but one step just above assembly
- **Clear lexical scope**, such as modules, functions, basic blocks, and instructions

# LLVM Intermediate Representation (IR)

**LLVM IR is LLVM's code representation which is generated by its front-end clang when compiling a program** (<https://llvm.org/docs/LangRef.html>)

- **Language independent.** Not machine code, but one step just above assembly
- **Clear lexical scope**, such as modules, functions, basic blocks, and instructions
- **3-address code style** in **static single assignment (SSA)** form

# LLVM Intermediate Representation (IR)

**LLVM IR is LLVM's code representation which is generated by its front-end clang when compiling a program** (<https://llvm.org/docs/LangRef.html>)

- **Language independent.** Not machine code, but one step just above assembly
- **Clear lexical scope**, such as modules, functions, basic blocks, and instructions
- **3-address code style in static single assignment (SSA) form**
  - Variables are strongly typed
  - Global variable (symbol starting with '@')
  - Stack/register variable (symbol starting with '%')
  - Three addresses and one operator.
    - For example, 'a = b op c', where 'a', 'b', 'c' are either programmer defined variables (e.g., heap, global or stack), constants or compiler-generated temporary names. 'op' stands for an operation which is applied on 'a' and 'b'.

# Compiling a C Program to Its LLVM IR

## Clang/LLVM compiler options

- Compile a C program 'swap.c' to a human readable IR 'example.ll'.
  - `clang -c -S -emit-llvm swap.c -o swap.ll`
- Compilation without optimisation.
  - `clang -c -S -Xclang -disable-00-optnone -emit-llvm swap.c -o swap.ll`
- Keep the variable names.
  - `clang -c -S -fno-discard-value-names -Xclang -disable-00-optnone -emit-llvm swap.c -o swap.ll`
- Convert the LLVM IR to SSA form's IR for later static analysis.
  - `opt -S -mem2reg swap.ll -o swap.ll`

# Compiling a C Program to Its LLVM IR

## An example

```
void swap(char **p, char **q){
    char* t = *p;
    *p = *q;
    *q = t;
}

int main(){
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}
```

swap.c

compile



```
define void @swap(i8** %p, i8** %q) #0 {
entry:
    %0 = load i8*, i8** %p, align 8
    %1 = load i8*, i8** %q, align 8
    store i8* %1, i8** %p, align 8
    store i8* %0, i8** %q, align 8
    ret void
}
```

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```

swap.ll

# C code to LLVM's IR

## An example

```
void swap(char **p, char **q){
    char* t = *p;
    *p = *q;
    *q = t;
}

int main(){
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}
```

swap.c

compile



```
define void @swap(i8** %p, i8** %q) #0 {
entry:
    %0 = load i8*, i8** %p, align 8
    %1 = load i8*, i8** %q, align 8
    store i8* %1, i8** %p, align 8
    store i8* %0, i8** %q, align 8
    ret void
}
```

Function

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```

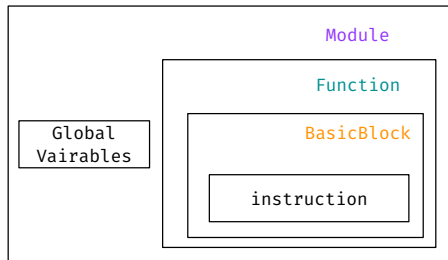
BasicBlock

Instruction

swap.ll

# LLVM Intermediate Representation (IR)

## Structure Organization



LLVM-IR Scopes

**Module** contains **Functions** and **Global Variables**

- Whole module is the unit of translation, analysis and optimization.

**Function** contains **BasicBlocks** and **Arguments**, which correspond to functions.

**BasicBlock** contains list of instructions.

- Each block is contiguous chunk of instructions

Instruction is opcode + vector of operands in '3-address' style

- All operands have types
- Instruction result is typed



# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main()
{
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}

define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```

%a1 = alloca i8, align 1

register  
variable

identifiers:

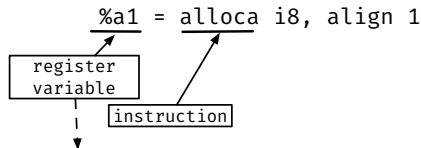
- [% / @] [-a-zA-Z\$. \_] [-a-zA-Z\$. \_0-9]
- % is for local variable
- @ is for global
- temporary variables are numbered

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main()
{
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}

define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```



identifiers:

`[% / @] [-a-zA-Z$. _] [-a-zA-Z$. _0-9]`

- % is for local variable

- @ is for global

- temporary variables are numbered

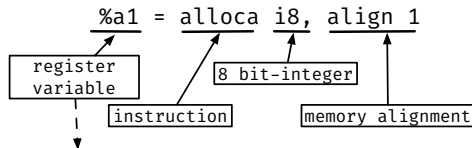
`alloca`: instruction allocates `i8` (`sizeof`) bytes of memory on run-time stack

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main()
{
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}

define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```



identifiers:

`[% / @] [-a-zA-Z$. _] [-a-zA-Z$. _0-9]`

- % is for local variable

- @ is for global

- temporary variables are numbered

`alloca`: instruction allocates `i8` (`sizeof`) bytes of memory on run-time stack

`align`: indicates the memory operation should be aligned to 1 byte

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main()
{
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}

define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```

`%a = alloca i8*, align 8`



allocate 8-bit integer pointer for a

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main()
{
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}
```

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```

%b1 = alloca i8, align 1



allocate 8-bit integer for b1

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main()
{
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}
```

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```

`%b = alloca i8*, align 8`



allocate 8-bit integer pointer for b

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main()
{
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}

define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```

store i8\* %a1, i8\*\* %a align 8

↑  
instruction

↙  
8-bit integer typed pointer %a1

store the pointer %a1 to the memory location that %a points to

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main()
{
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}
```

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```

store i8\* %b1, i8\*\* %b align 8

↑ instruction

↙ 8-bit integer typed pointer %b1

store the pointer %b1 to the memory location that %b points to



# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main()
{
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}
```

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1
    %a = alloca i8*, align 8
    %b1 = alloca i8, align 1
    %b = alloca i8*, align 8
    store i8* %a1, i8** %a, align 8
    store i8* %b1, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
```

call void @swap(i8\*\* %a, i8\*\* %b)

function call    function name    typed params

call instruction will be  
used to build control flow.

# LLVM Documentations

- LLVM Language Reference Manual <https://llvm.org/docs/LangRef.html>
- LLVM Programmer's Manual  
<https://llvm.org/docs/ProgrammersManual.html>
- Writing an LLVM Pass <http://llvm.org/docs/WritingAnLLVMPass.html>
- Tutorials for Clang/LLVM  
<https://freecompilercamp.org/clang-llvm-landing>
- LLVM Tutorial IEEE SecDev 2020 [https://cs.rochester.edu/u/ejohns48/secdev19/secdev20-llvm-tutorial-version4\\_copy.pdf](https://cs.rochester.edu/u/ejohns48/secdev19/secdev20-llvm-tutorial-version4_copy.pdf)