HW 1: LLVM Optimization

2021-01 인공지능플랫폼최적화

HW/SW Optimization for Machine Learning

박영준

HW #1 : LLVM Optimization

- 본 문서의 Step1-Step5의 내용을 직접 수행해보고 그 내용을 레포트로 작성하여 제출
- 레포트에는 다음의 내용이 필수적으로 포함되어야 함
  - [Step1] ReadIR 실행 결과 스크린샷
  - [Step2] PrintInst 실행 결과 스크린샷
  - [Step3] CountInst 실행 결과 스크린샷
  - [Step3] Exercise 1
  - [Step4] 원본 IR과 InsertInst의 결과 IR의 비교
  - [Step4] ReplaceInst를 적용하기 전과 적용한 결과 IR을 각각 바이너리로 컴파일하고 그 실행 결과를 비교
  - [Step4] Exercise 2
  - [Step5] MoveInst를 적용하기 전과 적용한 결과 IR을 각각 바이너리 컴파일하고 그 실행 결과를 비교
  - 위 리스트에 없는 Exercise나 Optional Project 등은 자유롭게 수행하되, 레포트에는 포함되지 않아도 무방함
    - (해당 과제들을 추가로 수행한다고 해도 추가 점수 없음)
- Deadline: 4/23 (Friday) 23:59:59
- Format: PDF or MS word file
- 문의: 강석원 (kswon0202@gmail.com)

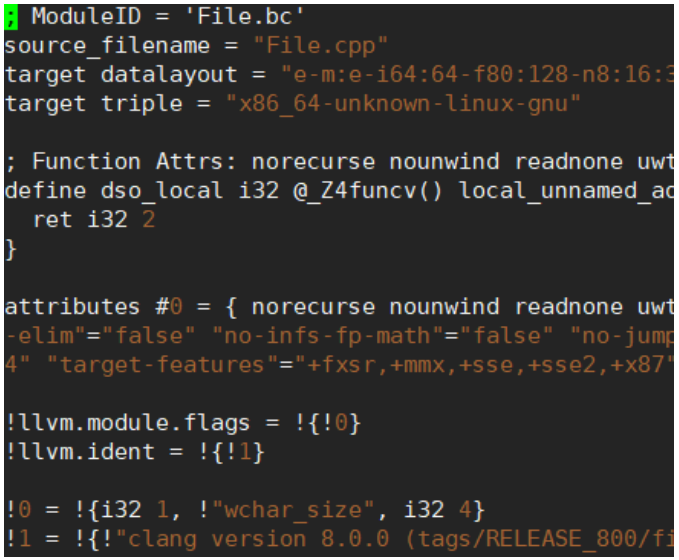
Getting Started

- LLVM을 개인 PC에 빌드 및 설치하기 위해서는 LLVM Getting Started를 참고하시기 바랍니다. LLVM을 소스코드로 부터 빌드하는 것은 컴퓨터 성능에 따라 1시간 – 3일 이상 소요될 수 있습니다. (본 튜토리얼에서는 LLVM과 Clang만 요구합니다.)
  https://releases.llvm.org/6.0.0/docs/GettingStarted.html
- (Recommended) Pre-Built Binary를 사용하여 빌드 없이 LLVM을 사용할 수 있습니다.
  https://releases.llvm.org/download.html
- (Recommended) OS: Ubuntu 16.04
  - 다른 OS에서도 가능합니다.
- 본 튜토리얼은 LLVM 6.0.1을 대상으로 작성되었습니다. 또한 튜토리얼 내 커맨드 및 소스 코드들은 컴파일러의 헤더 및 라이브러리, 바이너리들이 환경변수 등을 통해 절대 경로 지정 없이 접근 및 링킹 되는 환경을 가정합니다.
  - 본인 환경에 맞추어서 PATH나 LD_LIBRARY_PATH 등 환경변수를 지정하시기 바랍니다.
  - 참고: llvm/bin/llvm-config
- 보다 상위 버전의 LLVM에서는 더 많은 기능을 제공하고 있습니다. 단 일부 하위 호환성이 제공되지 않습니다. 자세한 내용은 LLVM에서 제공하는 Doxygen를 빌드하여서 확인 가능합니다. 최신 버전의 경우 다음 링크에서 접근할 수 있습니다.
  https://llvm.org/doxygen/index.html

The LLVM Compiler Infrastructure

- LLVM은 컴파일러를 위한 라이브러리 및 Tool Chain 모음으로써, 컴파일러(clang), 링커(llc), 디버거(ldb)등 다양한 서브 프로젝트들을 가지고 있습니다.
- LLVM은 LLVM IR이라는 중간 표현(Intermediate Representation, IR)을 사용하는데, Static Single Assignment(SSA) 형태의 코드 표현으로써, Human-readable Assembly(*.ll)과 Bitcode(*.bc) 두가지 포맷을 지원합니다.
- 이 튜토리얼에서는 LLVM IR 레벨에서 명령어를 처리하는 방법 위주로 설명합니다.

Bitcode (*.bc)



Human Readable Assembly (*.ll)

Step 1

- Step 1에서는 *clang, llvm-as, llvm-dis, llc*등의 툴을 사용하여 소스 코드 (*.c)를 IR 코드 (*.bc, *.ll)로, 그리고 다시 바이너리로 컴파일하는 방법을 익힙니다.
- 또한 LLVM의 API를 사용하여 이렇게 컴파일 된 IR 코드를 컴파일러 프로그램에서 (수정할 수 있도록) 로드 합니다.

- *clang*은 LLVM IR 기반 컴파일러로, C, C++등 다양한 프론트 엔드들을 지원합니다.
- *llvm-as*, *llvm-dis*는 IR 레벨의 Assembler와 Disassembler로 이를 사용하여 human-readable assembly 포맷과 bitcode 포맷의 IR 코드를 각각 다른 문법으로 바꿀 수 있습니다. 단, 파일에 (IR 문법 등의 이유로) 오류가 있다면 이 작업을 수행 할 수 없습니다.
- *llc*는 LLVM backend compile로 bitcode 포맷의 LLVM IR을 머신 어셈블리로 변환합니다.

Step 1

- Clang의 사용법은 아래와 같습니다. -I, -l, -L, -W, -c, -g 등 GCC에서도 통용되는 일반적인 컴파일 옵션을 대부분 지원합니다.

$ clang <source_file> [-o output_path]

```
$ clang HelloWorld.c -o HelloWorld
$ ./HelloWorld
```

```c
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

HelloWorld.c

- ***$ clang –emit-llvm -S <source_file> [-o output_path] # readable (*.ll)***

$ clang –emit-llvm -c <source_file> [-o output_path] # bitcode (*.bc)

```
$ clang -emit-llvm -S HelloWorld.c -o HelloWorld.ll
$ clang -emit-llvm -c HelloWorld.c -o HelloWorld.bc
```

Step 1

- llvm-as, llvm-dis를 사용하여 LLVM IR의 포맷을 변환 할 수 있습니다.

$ llvm-as <IR file> [-o=output path] # readable (*.ll) → bitcode (*.bc)

$ llvm-dis <IR file> [-o=output path] # bitcode(*.bc) → readable (*.ll)

- llvm-as, llvm-dis를 사용하여 LLVM IR 파일의 포맷을 변환하고 비교합니다.

```
$ llvm-as HelloWorld.ll -o=HelloWorld.2.bc
$ llvm-dis HelloWorld.bc -o=HelloWorld.2.ll
$ vimdiff HelloWorld.ll HelloWorld.2.ll
```

- llc를 사용하면 IR을 타겟 머신의 바이너리 (어셈블리)로 컴파일 할 수 있습니다.

$ llc <IR file> [-o output_path]

```
$ llc HelloWorld.ll -o HelloWorld.s
```

- clang을 사용하면 IR에서 실행가능한 바이너리로 컴파일 할 수 있습니다.
  (llc로 IR코드를 어셈블리로 컴파일하고 어셈블리를 돌린 후 링킹해서 실행가능한 바이너리를 얻을 수도 있습니다.)

```
$ clang HelloWorld.ll -o HelloWorld
```

Step 1

- LLVM 에서 제공하는 라이브러리 및 인터페이스들을 통해 LLVM IR을 명령어 수준에서 처리 할 수 있습니다.
- 이러한 LLVM IR을 처리하는 프로그램에서 LLVM 라이브러리들을 사용하기 위해, LLVM은 통합된 정보를 제공하기 위한 바이너리인 llvm-config를 제공합니다. llvm-config에서는 다양한 경로, 특정 기능을 사용하기 위해 필요한 종속 라이브러리 이름 등을 제공합니다.

$ clang++ <file> $(llvm-config --cxxflags --ldflags --system-libs --libs [name list])

- clang을 사용하여 IR을 읽어서 모듈이름을 출력하는 ReadIR.cpp를 컴파일합니다.

```
$ clang++ ReadIR.cpp -o ReadIR $(llvm-config --cxxflags --ldflags --system-libs --libs mcjit irreader)
```

- ReadIR을 실행하여 HelloWorld.ll과 HelloWorld.bc을 컴파일러 프로그램(ReadIR) 내에서 로드하고 모듈 이름을 출력합니다

```
$ ./ReadIR HelloWorld.ll

$ ./ReadIR HelloWorld.bc
```

Step 1

- ReadIR.cpp 내의 llvm::Module 클래스는 컴파일 및 최적화를 수행하기 위한 단위를 나타내는 클래스로 보통 하나의 파일을 의미합니다. llvm::LLVMContext 클래스는 전체 컴파일과정에서 일관된 자료형 및 전역 변수 등을 포함하는 컨택스트를 나타냅니다.

- 보다 자세한 LLVM 자료형 관련 내용들은 아래에서 확인 할 수 있습니다.
  http://llvm.org/doxygen/ (Note: 이 링크는 LLVM 최신 버전에 대한 문서임)

Step 1

```
40 llvm::LLVMContext* TheContext;
39 std::unique_ptr<llvm::Module> TheModule;
38
37 int main(int argc , char** argv)
36 {
35     std::string input_path;
34     llvm::SMDiagnostic err;
33
32     if(argc < 2)
31     {
30         std::cout << "Usage: ./ReadIR <ir_file_path>" << std::endl;
29         return -1;
28     }
27     input_path = std::string(argv[1]);
26
25     // Context owns and manages the core data of LLVM infrastructure, including the type and constant tables.
24     // new llvm::LLVMContext() allocate Context.
23     TheContext = new llvm::LLVMContext();
22     if( !TheContext )
21     {
20         std::cout<<"fail to allocate LLVMContext"<<std::endl;
19         return -1;
18     }
17
16     // Module is the top level container of LLVM intermediate Representation(IR) objects.
15     // Module take as input BitCode file and parse IR format.
14     TheModule = llvm::parseIRFile(input_path, err, *TheContext);
13     if( !TheModule )
12     {
11         std::cout << "Cannot open the file : "<< input_path << std::endl;
10         return -1;
 9     }
 8
 7     // Print Module Name
 6     std::cout << "Success reading & parsing the IR file." << std::endl;
 5     std::cout << "The module name is \"" << TheModule->getName().str() << "\"" << std::endl;
 4     std::cout << std::endl;
 3
 2     return 0;
 1 }
69
```

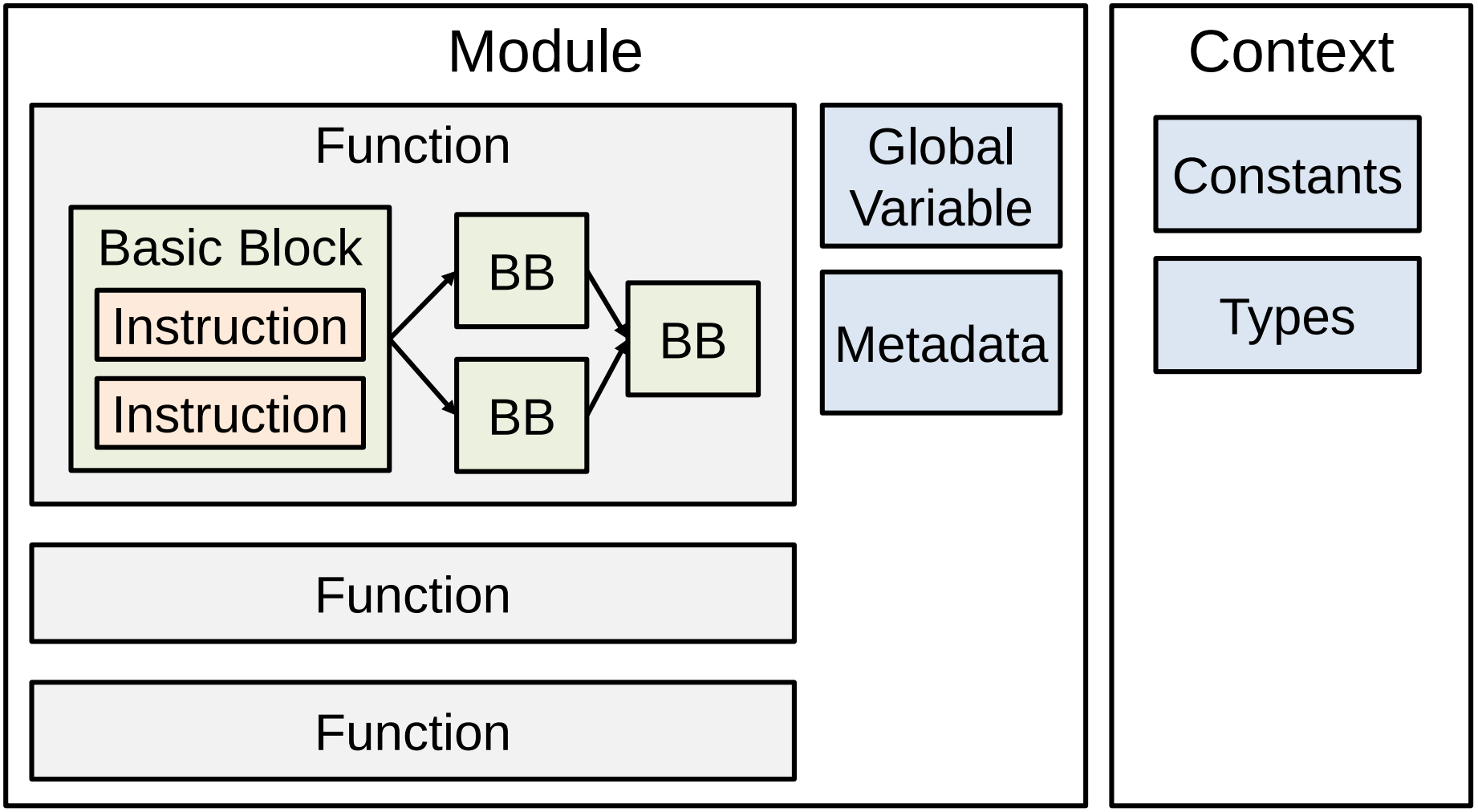
실제 IR 파일을 읽는 부분

Step 2

- Step 2에서는 LLVM의 기본적인 Module 구조를 이해하고, IR 내의 명령어들을 C++ iterator를 사용하여 순회 및 출력합니다.

- LLVM IR에서는 llvm::Module → llvm::Function → llvm::BasicBlock → llvm::Instruction의 계층 구조로 타겟 프로그램을 관리합니다.
- LLVM의 모델에 따르면 프로그램은 다음과 같은 구조로 되어있습니다.
  Module은 여러 개의 Function로 구성되어 있고,
  Function은 다시 여러 개의 Basic Block 으로 구성되어 있으며,
  Basic Block은 Instruction로 구성되어 있다.
  - Module = 모듈, 일반적으로 하나의 소스 파일
  - Function = 함수
  - Basic Block = Straight Forward Code Section, Branch나 Return같은 제어 명령어로 끝남
  - Instruction = 명령어
- 전역 변수를 의미하는 llvm::GlobalVariable 등은 llvm::Module 내에서 따로 관리 됩니다.
- 각 클래스들의 Iterator또는 다른 메소드들을 사용하여 위 계층구조를 접근 할 수 있습니다.
- Step 2에서는 Step 1에서 읽은 IR 코드의 함수와 명령어들을 출력합니다. 출력을 하기 위해서 llvm::raw_os_ostream 인스턴스를 사용합니다.

Step 2


Module
Function
Basic Block
Instruction
Instruction
BB
BB
BB
Function
Function
Global Variable
Metadata
Context
Constants
Types

Step 2

```
; ModuleID = 'Test.c'
source_filename = "Test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define i32 @func1() #0 {
  %1 = alloca i32, align 4
  store i32 4, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = icmp slt i32 %2, 3
  br i1 %3, label %4, label %7

; <label>:4:                                      ; preds = %0
  %5 = load i32, i32* %1, align 4
  %6 = add nsw i32 %5, 1
  store i32 %6, i32* %1, align 4
  br label %7

; <label>:7:                                      ; preds = %4, %0
  %8 = load i32, i32* %1, align 4
  ret i32 %8
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  ret i32 0
}
```

Module

Step 2

```
; ModuleID = 'Test.c'
source_filename = "Test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define i32 @func1() #0 {
  %1 = alloca i32, align 4
  store i32 4, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = icmp slt i32 %2, 3
  br i1 %3, label %4, label %7

; <label>:4:                                      ; preds = %0
  %5 = load i32, i32* %1, align 4
  %6 = add nsw i32 %5, 1
  store i32 %6, i32* %1, align 4
  br label %7

; <label>:7:                                      ; preds = %4, %0
  %8 = load i32, i32* %1, align 4
  ret i32 %8
}
```

Function

```
; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  ret i32 0
}
```

Function

Step 2

```
; ModuleID = 'Test.c'
source_filename = "Test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define i32 @func1() #0 {
  %1 = alloca i32, align 4
  store i32 4, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = icmp slt i32 %2, 3
  br i1 %3, label %4, label %7

; <label>:4:                                      ; preds = %0
  %5 = load i32, i32* %1, align 4
  %6 = add nsw i32 %5, 1
  store i32 %6, i32* %1, align 4
  br label %7

; <label>:7:                                      ; preds = %4, %0
  %8 = load i32, i32* %1, align 4
  ret i32 %8
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  ret i32 0
}
```

BasicBlock

BasicBlock

BasicBlock

Step 2

```
; ModuleID = 'Test.c'
source_filename = "Test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define i32 @func1() #0 {
  %1 = alloca i32, align 4
  %2 = load i32, i32* %1, align 4
  br i1 %3, label %4, label %7

; <label>:4:                                      ; preds = %0
  %5 = load i32, i32* %1, align 4
  %6 = add nsw i32 %5, 1
  store i32 %6, i32* %1, align 4
  br label %7

; <label>:7:                                      ; preds = %4, %0
  %8 = load i32, i32* %1, align 4
  ret i32 %8
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  ret i32 0
}
```

Instruction

Instruction

Instruction

Step 2

- PrintInst.cpp를 컴파일합니다.

```
$ clang++ PrintInst.cpp -o PrintInst $(llvm-config --cxxflags --ldflags --system-libs --libs mcjit irreader)
```

- 타겟 어플리케이션들을 IR 레벨로 컴파일합니다.

```
$ clang -emit-llvm -S Test.c -o Test.ll
```

- 프로그램을 통해 IR 파일을 읽고 명령어들을 출력합니다.

```
$ ./PrintInst Test.ll
```

Step 2

```
void TraverseModule(void)
{
    llvm::raw_os_ostream raw_cout( std::cout );

    // Module::iterator --> Function
    for( llvm::Module::iterator ModIter = TheModule->begin(); ModIter != TheModule->end(); ++ModIter )
    {
        llvm::Function* Func = llvm::cast<llvm::Function>(ModIter);

        // Print Function Name
        raw_cout << Func->getName() << '\n';

        // Function::iterator --> BasicBlock
        for( llvm::Function::iterator FuncIter = Func->begin(); FuncIter != Func->end(); ++FuncIter )
        {
            llvm::BasicBlock* BB = llvm::cast<llvm::BasicBlock>(FuncIter);

            // BasicBlock::iterator --> Instruction
            for( llvm::BasicBlock::iterator BBIter = BB->begin(); BBIter != BB->end(); ++BBIter )
            {
                llvm::Instruction* Inst = llvm::cast<llvm::Instruction>(BBIter);

                // Print Instruction
                raw_cout << '\t';
                Inst->print(raw_cout);
                raw_cout << '\n';
            }
        }
    }
}
```

llvm::Module::iterator = llvm::Function*

llvm::Function::iterator = llvm::BasicBlock*

llvm::BasicBlock::iterator = llvm::Instruction*

* llvm::cast<>는 iterator의 변환을 도와주는 함수임. 보통 일반적으로 llvm::isa<>와 같이 사용됨

Step 3

- Step3에서는 프로그램 내 특정 명령어의 수를 세어 출력하는 간단한 정적 프로파일러를 만듭니다.

- LLVM 6.0.1 기준으로, LLVM IR에는 총 64개의 명령어 Opcode가 있습니다.
  LLVM은 llvm::CallInst로 표현되는 Intrinsic이나 Vector 명령어 등 보다 더 많은 명령어들을 표현하고 있습니다.
  - llvm/include/llvm/IR/Instruction.def
- 자주 사용되는 몇가지 명령어들은 다음과 같습니다
  - BinaryOperator: add, sub, mul 과 같은 산술 연산이나, and, or과 같은 비교 연산 등 2개의 operand들을 연산하는 명령어들
  - ReturnInst, BranchInst 등: 제어 흐름관련 명령어들
  - CallInst: 함수 호출 명령어
  - CastInst: 타입 변환 명령어
  - AllocaInst: 스택(정적)에 메모리 할당하는 명령어
  - LoadInst, StoreInst: 메모리에 있는 데이터들을 접근하는 명령어
  - GetElementPtrInst: 배열 접근 등에서 메모리 접근하는 명령어

Step 3

- 전체 Add 명령어의 개수를 세려고 합니다.
- Step 2를 바탕으로 전체 명령어에 접근 할 수 있습니다. 여기에 추가로, llvm::Instruction의 메소드들을 사용하여 명령어가 어떤 종류의 것인지 파악 할 수 있습니다.

```
// Traverse Instructions in TheModule
void TraverseModule(void)
{
    int total_add_inst = 0;

    for( llvm::Module::iterator ModIter = TheModule->begin(); ModIter != TheModule->end(); ++ModIter )
    {
        llvm::Function* Func = llvm::cast<llvm::Function>(ModIter);

        for( llvm::Function::iterator FuncIter = Func->begin(); FuncIter != Func->end(); ++FuncIter )
        {
            llvm::BasicBlock* BB = llvm::cast<llvm::BasicBlock>(FuncIter);

            for( llvm::BasicBlock::iterator BBIter = BB->begin(); BBIter != BB->end(); ++BBIter )
            {
                llvm::Instruction* Inst = llvm::cast<llvm::Instruction>(BBIter);

                if( Inst->isBinaryOp() ) {}
            }
        }
    }
}
```

Step 3

- LLVM 6 기준 Add 명령어는 llvm::Instruction 클래스의 자식 클래스인 llvm::BinaryOperator 자료형으로 포함됩니다. 이전 슬라이드처럼 llvm::Instruction의 멤버 함수 isBinaryOp()를 사용하여 이를 확인 할 수 있습니다.
- 하지만 Add 명령어는 BinaryOpeartor의 Opcode로만 나타내어지기 때문에 (Add만을 위한 특별한 클래스가 없고, Sub, Mul등과 같이 BinaryOperator로 표현됨) 이 방법으로 해당 명령어가 Add 명령어인지는 확신 할 수 없습니다.

```
for(llvm::BasicBlock::iterator blockIter = funcIter->begin();
        blockIter != funcIter->end();
        blockIter++)
{
    llvm::Instruction* inst = llvm::cast<llvm::Instruction>(blockIter);

    if( inst->isBinaryOp() )
    {
    }
}
```

Step 3

- llvm::Instruction의 getOpcode() 메소드를 사용하여 Opcode를 직접 얻어서 현재 명령어가 어떤 명령어인지 확인 할 수 있습니다.

```
for( llvm::BasicBlock::iterator BBIter = BB->begin(); BBIter != BB->end(); ++BBIter )
{
    llvm::Instruction* Inst = llvm::cast<llvm::Instruction>(BBIter);

    if( Inst->getOpcode() == llvm::Instruction::Add ) { total_add_inst++; }
}
```

- CountInst.cpp를 바이너리로 컴파일하고, Test.c를 IR 레벨로 컴파일하여 실제 ADD 명령어의 수와 비교합니다.

```
$ clang++ CountInst.cpp -o CountInst $(llvm-config --cxxflags --ldflags --system-libs --libs mcjit irreader)
```

```
$ clang -emit-llvm -S Test.c -o Test.ll
```

```
$ ./CountInst ./Test.ll
```

Step 3

- -O3 옵션을 적용하면, Loop Unrolling과 같은 최적화가 적용되므로, 적용 전 후의 ADD 명령어 수가 다릅니다.

```
$ clang -O3 -emit-llvm -S Test.c -o Test.ll
```

```
$ ./CountInst ./Test.ll
```

- 다른 종류의 명령어들 또한 같은 방법으로 카운팅 할 수 있습니다.

- Exercise
  - Exercise 1: add, sub, mul, div의 개수를 모두 카운팅 합니다.
  - Exercise 2: 어떤 함수가 몇 번씩 호출되었는지 정적으로 카운팅 합니다.
    (Hint: llvm::CallInst 에서, getCalledFunction()을 통해 호출되는 함수를 접근 할 수 있습니다.)

Step 4

- Step 4에서는 IR 레벨에서 프로그램 내의 명령어를 삭제하거나 새로 생성하여 삽입합니다.
- Step 3을 바탕으로 전체 명령어 중 특정 종류의 명령어에 접근 할 수 있습니다. 이를 바탕으로 Step 4에서는 (1) 새로운 명령어를 생성하고 (2) 기존 명령어를 삭제하고 (3) 명령어간 종속성에 문제가 없도록 새로운 명령어로 기존 명령어를 대체하는 과정을 수행합니다.

Step 4

- LLVM에서는 명령어를 생성하기 위해 llvm::IRBuilder 클래스를 제공하고 있습니다. 이 IRBuilder 인스턴스를 사용할 수 있고, 각 명령어 클래스들에서 제공하는 static 메소드 Create를 사용하여서 명령어를 생성 할 수도 있습니다.
- 일반적으로 LLVM은 아래와 같이 수행할 명령어의 Operand들을 Create 메소드의 인자로 요구합니다. Operand에는 각 명령어 별로 요구하는 것에 따라 Value, Type 등이 될 수 있습니다.

```
/// Construct a binary instruction, given the opcode and the two
/// operands.  Optionally (if InstBefore is specified) insert the instruction
/// into a BasicBlock right before the specified instruction.  The specified
/// Instruction is allowed to be a dereferenced end iterator.
///
static BinaryOperator *Create(BinaryOps Op, Value *S1, Value *S2,
                              const Twine &Name = Twine(),
                              Instruction *InsertBefore = nullptr);
```

- 또한 결과 값을 가지는 명령어 (binary operator, non-void return function call 등) 또한 Instruction 객체(인스턴스)를 다른 Isntruction의 Operand로 사용함으로써 그 결과 값을 다른 명령어의 Operand로 사용할 수 있습니다. (llvm::Instruction 클래스는 llvm::Value 클래스의 자식 클래스입니다.)

Step 4

- InsertInst는 타겟 어플리케이션 내 ADD 명령어가 있다면 바로 그 앞에 1 + 1연산을 추가합니다.
  (즉 ADD 1, 1의 명령어를 삽입합니다)

```
// Traverse Instructions in TheModule
void TraverseModule(void)
{
    for( llvm::Module::iterator ModIter = TheModule->begin(); ModIter != TheModule->end(); ++ModIter )
    {
        llvm::Function* Func = llvm::cast<llvm::Function>(ModIter);

        for( llvm::Function::iterator FuncIter = Func->begin(); FuncIter != Func->end(); ++FuncIter )
        {
            llvm::BasicBlock* BB = llvm::cast<llvm::BasicBlock>(FuncIter);

            for( llvm::BasicBlock::iterator BBIter = BB->begin(); BBIter != BB->end(); ++BBIter )
            {
                llvm::Instruction* Inst = llvm::cast<llvm::Instruction>(BBIter);
                // if( Inst->isBinaryOp() ) {}
                if( Inst->getOpcode() == llvm::Instruction::Add )
                {
                    llvm::BinaryOperator::Create(
                            llvm::Instruction::Add, /* Opcode */
                            llvm::ConstantInt::get( llvm::IntegerType::get( *TheContext, 32 ), 1, true ),   /* S1 : i32 1 */
                            llvm::ConstantInt::get( llvm::IntegerType::get( *TheContext, 32 ), 1, true ),   /* S2 : i32 1 */
                            "addtmp",   /* Name */
                            Inst /* BeforeInst */ );
                }
            }
        }
    }
}
```

Step 4

- 타겟 어플리케이션과 작성한 프로그램을 각각 컴파일하고, 수행 후 원본과 처리 후 파일을 비교합니다.

```
$ clang++ InsertInst.cpp -o InsertInst $(llvm-config --cxxflags --ldflags --system-libs --libs mcjit irreader)
$ clang -emit-llvm -S Test.c -o Test.ll
$ ./InsertInst Test.ll Test.Processed.ll
$ vimdiff Test.ll Test.Processed.ll
```

- IR 레벨에서 명령어를 추가한 타겟 어플리케이션을 바이너리로 컴파일하고 실행합니다.
  (IR에 오류가 있다면 바이너리로 컴파일 되지 않습니다)

```
$ clang ./Test.Processed.ll -o Test
$ ./Test
```

Step 4

- 특정 명령어를 삭제하기 위해서는, 당연히 해당 명령어의 결과 값을 사용하는 명령어들의 종속 관계를 해결해 주어야 합니다. 예를 들어 A = 2 + 3를 계산하는 연산에서 명령어가 2 + 3을 연산하는 명령어 1번과 명령어 1번의 결과 값을 A에 저장(Store) 하는 명령어 2가가 있다고 가정할 때, 명령어 1의 결과 값이 명령어 2에서 사용되므로 명령어 1을 지우면 컴파일을 제대로 수행할 수 없습니다.
- 따라서 이 Step4에서는 타겟 어플리케이션 내에 있는 ADD 명령어들을 SUB 명령어로 바꾸는 과정을 수행하는데, 이를 위해 (1) ADD 명령어와 같은 피연산자들로 SUB 명령어를 생성하고 (2) ADD 명령어의 결과 값이 사용되는 곳(USE)들을 SUB 명령어의 결과를 사용하도록 변경하고, (3) 최후에 ADD 명령어를 삭제하는 과정을 거칩니다.
- IR 레벨의 소스에서 생기는 문제는 주로 Human-readable IR로 현재의 Module을 출력한 이후, llvm-as를 통해 bitcode로 포맷을 변환하면 확인 할 수 있습니다.

Step 4

- (1) 타겟 어플리케이션에서, 원래 ADD 명령어와 같은 Operand로, ADD 명령어 이전에 SUB 명령어가 수행되도록 코드를 변경합니다.

```
for( llvm::BasicBlock::iterator BBIter = BB->begin(); BBIter != BB->end(); ++BBIter )
{
    llvm::Instruction* Inst = llvm::cast<llvm::Instruction>(BBIter);
    // if( Inst->isBinaryOp() ) {}
    if( Inst->getOpcode() == llvm::Instruction::Add )
    {
        llvm::BinaryOperator::Create(
                llvm::Instruction::Sub, /* Opcode */
                Inst->getOperand(0),    /* S1 */
                Inst->getOperand(1),    /* S2 */
                "subtmp",   /* Name */
                Inst /* BeforeInst */ );
    }
}
```

```
%16 = load i32, i32* %15, align 4
%17 = add nsw i32 %12, %16
%18 = sext i32 %17 to i64
%19 = load i64, i64* %8, align 8
%20 = add i64 %18, %19
%21 = trunc i64 %20 to i32
%22 = load i32*, i32** %7, align 8
%23 = load i64, i64* %8, align 8
```

```
4   %16 = load i32, i32* %15, align 4
5   %subtmp = sub i32 %12, %16
6   %17 = add nsw i32 %12, %16
7   %18 = sext i32 %17 to i64
8   %19 = load i64, i64* %8, align 8
9   %subtmp1 = sub i64 %18, %19
10  %20 = add i64 %18, %19
11  %21 = trunc i64 %20 to i32
```

Sub 명령어 추가

Step 4

- (2) 생성한 Sub 명령어가 Add 명령어가 사용되는 곳에 대신하여 사용되도록 코드를 변경합니다.

```
for( llvm::BasicBlock::iterator BBIter = BB->begin(); BBIter != BB->end(); ++BBIter )
{
    llvm::Instruction* AddInst = llvm::cast<llvm::Instruction>(BBIter);
    // if( Inst->isBinaryOp() ) {}
    if( AddInst->getOpcode() == llvm::Instruction::Add )
    {
        llvm::Instruction* SubInst = llvm::BinaryOperator::Create(
                llvm::Instruction::Sub, /* Opcode */
                AddInst->getOperand(0), /* S1 */
                AddInst->getOperand(1), /* S2 */
                "subtmp",   /* Name */
                AddInst /* BeforeInst */ );
        AddInst->replaceAllUsesWith( SubInst );
    }
}
```

```
%16 = load i32, i32* %15, align 4
%17 = add nsw i32 %12, %16
%18 = sext i32 %17 to i64
```

```
1   %subtmp = sub i32 %12, %16
2   %17 = add nsw i32 %12, %16
3   %18 = sext i32 %subtmp to i64
```

"Add 명령어의 결과 값"을 사용하는 부분을
"Sub 명령어의 결과 값"을 사용하도록 변경

Step 4

- (3) ADD 명령어를 삭제합니다. 단, 현재 BBIter가 삭제할 명령어를 가리키고 있기 때문에, loop 내에서 삭제하면 오류가 발생합니다. loop 밖에서 삭제해야 합니다.

```
for( llvm::Function::iterator FuncIter = Func->begin(); FuncIter != Func->end(); ++FuncIter )
{
    llvm::BasicBlock* BB = llvm::cast<llvm::BasicBlock>(FuncIter);
    std::vector< llvm::Instruction* > AddInsts;

    for( llvm::BasicBlock::iterator BBIter = BB->begin(); BBIter != BB->end(); ++BBIter )
    {
        llvm::Instruction* AddInst = llvm::cast<llvm::Instruction>(BBIter);

        // if( Inst->isBinaryOp() ) {}
        if( AddInst->getOpcode() == llvm::Instruction::Add )
        {
            llvm::Instruction* SubInst = llvm::BinaryOperator::Create(
                    llvm::Instruction::Sub, /* Opcode */
                    AddInst->getOperand(0), /* S1 */
                    AddInst->getOperand(1), /* S2 */
                    "subtmp",   /* Name */
                    AddInst /* BeforeInst */ );
            AddInst->replaceAllUsesWith( SubInst );

            AddInsts.push_back( AddInst );
        }
    }

    for( int i=0, Size=AddInsts.size(); i<Size; ++i ) AddInsts[i]->eraseFromParent();
}
```

```
%17 = add nsw i32 %12, %16
%18 = sext i32 %17 to i64
```

```
1   %subtmp = sub i32 %12, %16
2   %18 = sext i32 %subtmp to i64
```

Add 명령어 삭제

Step 4

- 타겟 어플리케이션과 작성한 프로그램을 각각 컴파일하고, 수행 후 원본과 처리 후 파일을 비교합니다.

```
$ clang++ ReplaceInst.cpp -o ReplaceInst $(llvm-config --cxxflags --ldflags --system-libs --libs mcjit irreader
$ clang -emit-llvm -S Test.c -o Test.ll
$ ./ReplaceInst ./Test.ll ./Test.Processed.ll
$ vimdiff ./Test.ll ./Test.Processed.ll
```

- IR 레벨에서 명령어를 추가한 타겟 어플리케이션을 바이너리로 컴파일하고, 실행합니다.

```
clang Test.Processed.ll -o Test
./Test
```

Step 4

- Exercise
  - Exercise 1: 앞서의 Step 4에서 (2)의 코드를 삭제한 이후 ((3)은 수행), 어떤 문제가 발생하는지 확인합니다.
  - Exercise 2: A + B + C의 연산 패턴을 탐색하고, 이를 A * B – C로 변경합니다.
    (Hint: 연산 순서상, A + B + C는 (A + B) + C와 같은데, 이것은 A + B 명령어의 결과 값이 ADD 명령어의 Operand로 들어가는 것과 같습니다.)

Step 5

- Step 5에서는 IR 레벨에서 명령어의 위치를 변경하고, 메모리 명령어의 종속성 관계에 대해서 파악합니다.
- 일반적인 산술 연산 명령어와 다르게, 메모리 명령어 (load/store)의 종속 관계는 컴파일러 입장에서 대부분 알기 어렵습니다. 예를 들어 주소 X에서 값을 읽은 명령어와(load) 주소 Y에 값을 저장하는 명령어(store)가 있다고 할 때, 주소 X와 주소 Y가 같거나, 다르다는 보장이 없는 이상 두 명령어간 순서는 두 명령어간 def-use chain에서 종속 관계가 없다고 하더라도 쉽게 바뀔 수 없습니다.
- Step 5에서는 Store 명령어를 Store 명령어가 있는 BasicBlock 가장 마지막 부분으로 위치를 옮기는 작업을 수행합니다.
- 앞서 설명한 것의 이유로, Store 명령어의 위치 이동으로 이루어지는 종속성 문제는 컴파일러 레벨에서 특별한 오류를 발생시키지 않습니다.

Step 5

- 프로그램 내의 Store Instruction을 식별하도록 코드를 작성합니다.
- LLVM 6에서, Store Instruction은 llvm::StoreInst 자료형으로 표현됩니다.

```
// Traverse Instructions in TheModule
void TraverseModule(void)
{
    for( llvm::Module::iterator ModIter = TheModule->begin(); ModIter != TheModule->end(); ++ModIter )
    {
        llvm::Function* Func = llvm::cast<llvm::Function>(ModIter);

        for( llvm::Function::iterator FuncIter = Func->begin(); FuncIter != Func->end(); ++FuncIter )
        {
            llvm::BasicBlock* BB = llvm::cast<llvm::BasicBlock>(FuncIter);

            std::vector< llvm::StoreInst* > StoreInsts;

            for( llvm::BasicBlock::iterator BBIter = BB->begin(); BBIter != BB->end(); ++BBIter )
            {
                llvm::Instruction* Inst = llvm::cast<llvm::Instruction>(BBIter);

                //if( Inst->getOpcode() == llvm::Instruction::Store )
                if( llvm::isa< llvm::StoreInst >( Inst ) )
                {
                    StoreInsts.push_back( llvm::cast< llvm::StoreInst >( Inst ) );
                }
            }
        }
    }
}
```

Step 5

- Basic Block 내의 마지막 명령어를 탐색하는 코드를 추가합니다.
- LLVM IR에서 Basic Block의 마지막 명령어는 반드시 TerminatorInst 이여야 합니다. (Return, Br 등이여야 함)

```
void TraverseModule(void)
{
    for( llvm::Module::iterator ModIter = TheModule->begin(); ModIter != TheModule->end(); ++ModIter )
    {
        llvm::Function* Func = llvm::cast<llvm::Function>(ModIter);

        for( llvm::Function::iterator FuncIter = Func->begin(); FuncIter != Func->end(); ++FuncIter )
        {
            llvm::BasicBlock* BB = llvm::cast<llvm::BasicBlock>(FuncIter);

            std::vector< llvm::StoreInst* > StoreInsts;
            llvm::Instruction* LastInst;

            for( llvm::BasicBlock::iterator BBIter = BB->begin(); BBIter != BB->end(); ++BBIter )
            {
                llvm::Instruction* Inst = llvm::cast<llvm::Instruction>(BBIter);

                //if( Inst->getOpcode() == llvm::Instruction::Store )
                if( llvm::isa< llvm::StoreInst >( Inst ) )
                {
                    StoreInsts.push_back( llvm::cast< llvm::StoreInst >( Inst ) );
                }

                LastInst = Inst;
            }
        }
    }
}
```

Step 5

- Store 명령어들을 Basic Block 내의 마지막 명령어 이전으로 위치를 이동합니다,

```
llvm::BasicBlock* BB = llvm::cast<llvm::BasicBlock>(FuncIter);

std::vector< llvm::StoreInst* > StoreInsts;
llvm::Instruction* LastInst;

for( llvm::BasicBlock::iterator BBIter = BB->begin(); BBIter != BB->end(); ++BBIter )
{
    llvm::Instruction* Inst = llvm::cast<llvm::Instruction>(BBIter);

    //if( Inst->getOpcode() == llvm::Instruction::Store )
    if( llvm::isa< llvm::StoreInst >( Inst ) )
    {
        StoreInsts.push_back( llvm::cast< llvm::StoreInst >( Inst ) );
    }

    LastInst = Inst;
}

for( int i=0, Size=StoreInsts.size(); i<Size; ++i )
{
    StoreInsts[i]->moveBefore(LastInst);
}
```

Step 5

- 타겟 어플리케이션과 작성한 프로그램을 각각 컴파일하고, 수행 후 원본과 처리 후 파일을 비교합니다.

```
$ clang++ MoveInst.cpp -o MoveInst $(llvm-config --cxxflags --ldflags --system-libs --libs mcjit irreader)
$ clang -emit-llvm -S Test.c -o Test.ll
$ ./MoveInst ./Test.ll ./Test.Processed.ll
$ vimdiff ./Test.ll ./Test.Processed.ll
```

- 원본과 처리 후 실행 결과를 비교합니다.

```
$ clang Test.ll -o Test
$ clang Test.Processed.ll -o Test.Processed
$ ./Test
$ ./Test.Processed
```

- Exercise
  - Exercise 1: Store 명령어가 접근하는 주소를 바탕으로, 불필요한 Load를 삭제합니다.
    (Load와 Store 사이에 다른 메모리 접근 명령어가 없고, 주소가 같은 것을 완전히 보장 할 수 경우에만 Load 명령어의 결과 값 대신 Store에 저장하는 값 (레지스터에 저장된)을 사용 할 수 있습니다.)

Optional Project 1

- 이 프로젝트에서는 간단한 수학적 최적화를 수행합니다.
- A가 정수 데이터 일 때, A + 0 과 A * 1은 둘 다 결과 값이 A 입니다. 즉, 컴파일 단계에서 판단 할 수 있는 쓸모 없는 연산으로 일반적으로 삭제됩니다. 하지만 프로그램을 낮은 최적화 단계(O0)의 옵션으로 컴파일 하면 이러한 연산들이 여전히 IR 단계에서 남아있습니다. 이 프로젝트에서는 이러한 연산들을 삭제합니다.
- Input: 임의의 C 프로그램의 IR
- Output: A = A + 0와, A = A * 1 명령어가 지워진 IR
- Note: A는 32비트 정수(i32)만을 대상으로 합니다.
  결과 IR의 종속성 관계에 문제가 없어야 합니다.
  (즉, bitcode로 변환할 수 없는 경우 실패)

```
1  ; Function Attrs: noinline norecurs
2  define i32 @main() #4 {
3    %1 = alloca i32, align 4
4    %2 = alloca i32, align 4
5    store i32 0, i32* %1, align 4
6    store i32 1, i32* %2, align 4
7    %3 = load i32, i32* %2, align 4
8    %4 = add nsw i32 %3, 0
9    store i32 %4, i32* %2, align 4
10   %5 = load i32, i32* %2, align 4
11   ret i32 %5
12 }
```

A = A + 0 의 명령어가 있는 IR

Optional Project 2

- 이 프로젝트에서는 간단한 동적 프로파일러를 개발합니다.
- 컴파일 과정에서 분석하는 정적(static) 프로파일링은 루프 등에서 반복적으로 실행되는 명령어를 커버할 수 없다는 단점이 있습니다. 이를 해결하기 위해, 동적으로 실제 실행되는 명령어를 IR 레벨에서 카운팅 합니다.
- Input: 임의의 C 프로그램의 IR. 0으로 초기화 된 I32 전역 변수 add_inst_count 가 프로그램 내에 있다고 가정합니다.
- Output: 프로그램 내에 add 명령어가 실행 될 때마다 add_inst_count += 1을 add 바로 직전 (또는 직후)에 실행하도록 추가한 IR.
- Note: 전역 변수의 값을 업데이트 하기 위해서는 load-add-store의 과정이 필요합니다.
  프로그램의 마지막 부분에서 add_inst_count을 출력할 것입니다.