# C 언어를 위한 최소 LLVM IR 명세: 의미론적 등가성 보고서

**초록:** 본 보고서는 C 프로그래밍 언어의 완전한 의미론적 표현을 달성하는 데 필요한 LLVM 중간 표현(Intermediate Representation, IR)의 최소 부분 집합에 대한 명확한 명세를 제공합니다. C 언어의 타입 시스템과 메모리 모델부터 연산자, 제어 흐름에 이르는 모든 구문을 근본적인 IR 등가물로 체계적으로 매핑함으로써, 이 문서는 최적화되지 않은 C 프론트엔드 컴파일의 기준선을 설정합니다. 이 명세는 순수한 의미론적 정확성과 표현력에 집중하기 위해 최적화 중심의 명령어, 메타데이터, 속성을 의도적으로 제외합니다.

## 섹션 1: 최소 IR 명세의 기본 개념

이 섹션은 비자명한 프로그램을 표현하는 데 필수적인 LLVM IR의 핵심 원칙을 확립합니다. LLVM 구조의 이면에 있는 "이유"를 설명하며, 그 설계가 어떻게 C와 같은 명령형 언어의 표현을 직접적으로 촉진하는지에 초점을 맞춥니다.

### 1.1 계층적 구조: 모듈, 함수, 기본 블록

LLVM IR의 구조는 단순한 구성을 넘어 C와 같은 구조적 프로그래밍 언어의 스코프 및 제어 흐름 개념을 직접적으로 반영합니다.

* **모듈 (Module):** LLVM IR의 최상위 컨테이너로, 단일 번역 단위(전처리 후의 .c 파일)에 해당합니다. 모듈은 전역 변수 정의, 함수 선언 및 정의를 포함합니다.1 모듈은 전역 심볼과 링크 타입을 위한 스코프 역할을 합니다.
* **함수 (Function):** 실행 가능한 코드를 담는 컨테이너로, 정의는 define 키워드로, 선언은 declare 키워드로 지정됩니다.3 각 함수는 시그니처(반환 타입 및 매개변수 타입)와 기본 블록으로 구성된 본문을 가집니다.
* **기본 블록 (Basic Block):** 종료(terminator) 명령어가 아닌 명령어들의 시퀀스와 정확히 하나의 종료 명령어로 구성됩니다.4 C의 모든 제어 흐름은  
  br, switch와 같은 종료 명령어에 의해 결정되는 이 블록들 간의 연결로 모델링됩니다. 기본 블록은 C의 if나 while 문보다 더 원시적인 제어 흐름의 원자적 단위입니다.

### 1.2 핵심 원칙으로서의 정적 단일 할당 (SSA) 형태

* **정의:** 모든 변수(레지스터)가 정확히 한 번만 값을 할당받고, 사용되기 전에 정의되어야 한다는 LLVM IR의 근본적인 제약 조건입니다.4 지역 변수는  
  % 접두사를 가집니다.
* **의미론적 영향:** C 변수는 변경 가능하지만(mutable), LLVM 가상 레지스터는 그렇지 않습니다. C의 변경 가능한 변수를 SSA 형태로 매핑하는 것은 프론트엔드의 주요 작업입니다. 최소한의 최적화되지 않은 표현을 위한 일반적인 전략은 alloca, load, store 패턴을 사용하여 메모리에서 변경 가능한 변수를 효과적으로 시뮬레이션하는 것입니다. 이는 프론트엔드에서 복잡한 SSA 구성을 즉시 필요로 하지 않게 합니다.7

### 1.3 제어 흐름 경로 병합에서 phi 명령어의 역할

* **기능:** phi 명령어는 제어 흐름이 있는 상황에서 SSA 형태를 유지 가능하게 하는 메커니즘입니다. 현재 블록에 도달하기 위해 어떤 경로가 사용되었는지에 따라 이전 블록 중 하나에서 값을 선택합니다.4
* **최소 표현:** 정교한 프론트엔드는 phi 노드를 직접 생성할 수 있지만, C의 최소 표현은 초기에 phi 없이 달성될 수 있습니다. 본 명세는 alloca/load/store 패턴을 변경 가능한 C 변수를 표현하는 가장 근본적인 방법으로 정의합니다. phi 명령어는 초기 IR 생성에 필수적인 요소가 아니라, mem2reg 최적화 패스가 alloca 패턴으로부터 생성하는 정규 SSA 표현으로 제시됩니다.6 따라서 이 최소 명세의 목적상,  
  alloca 접근 방식이 기준선으로 정의되며, phi는 그것의 SSA 형태 등가물로 설명됩니다. C의 변경 가능한 변수와 LLVM의 SSA 형태 사이의 긴장감은 컴파일러 설계의 핵심적인 아키텍처 결정을 드러냅니다. 프론트엔드는 메모리 기반 연산(alloca/load/store)을 내보내어 "단순하게" 유지하고, SSA 구성이라는 복잡한 작업을 표준화된 최적화 패스(mem2reg)에 위임할 수 있습니다. 이는 최소한의 C 표현이 phi 노드의 올바른 배치를 위해 필요한 지배 경계(dominance frontier) 문제를 프론트엔드에서 해결할 것을 엄격하게 요구하지 않음을 시사합니다. 최소 요구 사항은 스택 메모리를 사용하여 변수의 수명과 값의 변화를 올바르게 모델링하는 것입니다. 이로 인해 alloca 명령어는 C 프로그램의 의미를 처음부터 표현하는 데 있어 phi보다 더 근본적인 프리미티브가 됩니다.7

## 섹션 2: C 타입 시스템 매핑

이 섹션은 C의 타입 시스템과 LLVM IR에서 사용 가능한 타입 간의 직접적인 대응 관계를 상세히 설명합니다.

### 2.1 기본 타입 (Primitive Types)

* **정수 타입:** C의 char, short, int, long, long long은 LLVM의 iN 타입으로 매핑되며, 여기서 N은 비트 폭입니다 (예: i8, i16, i32, i64).10 LLVM IR은 타입 수준에서 부호 있는 정수와 부호 없는 정수를 구분하지 않으며, 이 구분은 해당 타입에 작용하는 명령어(예:  
  sdiv 대 udiv)에 의해 처리됩니다.13  
  \_Bool은 i1로 매핑됩니다.4
* **부동 소수점 타입:** C의 float, double, long double은 각각 float, double, x86\_fp80 (또는 대상 트리플에 따라 fp128, ppc\_fp128)으로 매핑됩니다.10
* **Void 타입:** C의 void 반환 타입은 LLVM의 void 타입으로 직접 매핑됩니다.10

### 2.2 복합 타입 (Aggregate Types)

* **배열:** C의 T a[N]은 LLVM 배열 타입 [N x <type>]으로 매핑됩니다.10 크기는 타입의 일부라는 점이 중요합니다.
* **구조체:** C의 struct S { T1 f1; T2 f2; }는 명명된(identified) 또는 리터럴(익명) LLVM 구조체 타입 %T = type { <type1>, <type2> }으로 매핑됩니다.10 LLVM 구조체는 필드 이름이 아닌 인덱스로 접근합니다.16

### 2.3 포인터 타입과 불투명 포인터 모델

* **포인터:** 모든 C 포인터 타입(T\*, void\*, 함수 포인터)은 최신 LLVM IR에서 단일의 불투명 포인터 타입 ptr로 매핑됩니다.18 포인터의 타입 정보는 이를 사용하는 명령어(예:  
  load, store, getelementptr)에 의해 강제되며, 이 명령어들은 가리키는 요소의 타입을 명시합니다. 이는 이전의 타입이 있는 포인터 모델(i32\* 등)로부터의 중요한 단순화입니다.
* **함수 포인터:** int (char, double) 함수에 대한 포인터는 ptr로 표현되지만, call 명령어에서 사용될 때는 호출 지점에서 올바른 함수 타입인 i32 (i8, double)\*로 캐스팅되어야 합니다.

| 표 2.1: C 데이터 타입과 LLVM IR 타입 매핑 |  |
| --- | --- |
| **C 타입** | **LLVM IR 타입** |
| char, signed char, unsigned char | i8 |
| short, signed short, unsigned short | i16 |
| int, signed int, unsigned int | i32 |
| long, signed long, unsigned long | i32 또는 i64 (데이터 레이아웃에 따라) |
| long long, signed long long, unsigned long long | i64 |
| \_Bool | i1 |
| float | float |
| double | double |
| long double | x86\_fp80, fp128, 또는 ppc\_fp128 |
| T \* (모든 포인터) | ptr |
| void (반환 타입) | void |
| T[N] (배열) | `` |
| struct { T1 f1; T2 f2; } | { T1, T2 } |

## 섹션 3: 메모리 모델 및 변수 표현

이 섹션은 C 변수들이 서로 다른 저장 기간을 가지며 LLVM IR 메모리 모델에서 어떻게 인스턴스화되는지를 상세히 설명합니다.

### 3.1 전역 변수

* **정의:** 정적 저장 기간을 갖는 C 변수(파일 스코프에서 정의되거나 static으로 선언된 변수)는 LLVM 모듈에서 @ 접두사를 가진 전역 변수로 표현됩니다.4
* **문법:** @var = [Linkage][global | constant] <type> <initializer>.19
* **링크 타입:** C 의미를 표현하는 데 필요한 최소 링크 타입은 private(static 변수용), internal, external(기본 전역 변수용)입니다.
* **접근:** 전역 변수는 포인터로 취급됩니다. 그 값에 접근하려면 명시적인 load 및 store 명령어가 필요합니다.19 LLVM IR은 주소(포인터)와 그 주소에 저장된 값을 엄격하게 분리합니다. C에서는  
  x = 5;와 같이 전역 변수를 작성하는 것이 단일 연산이지만, LLVM IR에서는 전역 @x가 메모리를 가리키는 포인터로 존재하고, 해당 메모리를 수정하기 위해 store i32 5, ptr @x 명령어가 필요한 두 단계로 나뉩니다. 이러한 명시성은 데이터 흐름과 메모리 접근 패턴을 분석에 투명하게 만듭니다.

### 3.2 지역 변수

* **할당:** 자동 저장 기간을 갖는 C 변수(지역 변수)는 alloca 명령어를 사용하여 함수의 스택 프레임에 할당됩니다.7
* **문법:** %ptr = alloca <type> [, align <alignment>].
* **결과:** alloca는 할당된 메모리에 대한 포인터(ptr)를 반환합니다. 이 포인터는 load 및 store와 함께 사용되어 C 지역 변수의 변경 가능한 동작을 시뮬레이션합니다. alloca 명령어는 C의 명령형, 스택 기반 변수 모델과 LLVM의 레지스터 기반 SSA 모델 사이의 근본적인 다리 역할을 합니다. C 함수는 변수들이 안정적인 메모리 주소를 갖는 스택 프레임에서 작동하며, alloca 명령어는 C 추상 기계의 이 부분을 직접 모델링합니다.7 모든 변경 가능한 지역 변수에 대해 메모리를 할당함으로써, 프론트엔드는 SSA에 대해 추론하지 않고도 올바른 코드를 생성할 수 있습니다.

### 3.3 메모리 접근 프리미티브: load와 store

* **load:** 메모리 위치에서 값을 읽습니다. 문법: %val = load <type>, ptr %ptr.7  
  <type>은 메모리에서 읽어올 데이터의 타입을 명시하여 접근 지점에서 타입 안전성을 강제합니다.
* **store:** 메모리 위치에 값을 씁니다. 문법: store <type> %val, ptr %ptr.7

## 섹션 4: C 연산자와 LLVM IR 명령어 매핑

이 섹션은 C의 표현식과 연산자가 LLVM IR 명령어로 어떻게 변환되는지에 대한 포괄적인 목록을 제공합니다.

### 4.1 이항 및 단항 산술 연산

* **매핑:** +, -, \*, /, %는 각각 add, sub, mul, sdiv/udiv, srem/urem으로 매핑됩니다.4 부호 있는(  
  s) 버전과 부호 없는(u) 버전 간의 선택은 피연산자의 C 타입에 따라 달라집니다. 단항 -는 sub <type> 0, %val로 구현됩니다.

### 4.2 비트 및 시프트 연산

* **매핑:** &, |, ^, <<, >>는 각각 and, or, xor, shl, lshr/ashr로 매핑됩니다.4 논리적 오른쪽 시프트(  
  lshr)와 산술적 오른쪽 시프트(ashr) 간의 선택은 시프트되는 C 피연산자의 부호에 따라 달라집니다. 단항 ~는 xor <type> %val, -1로 구현됩니다.

### 4.3 비교 연산

* **매핑:** C의 관계 연산자(==, !=, >, <, >=, <=)는 icmp(정수 비교) 및 fcmp(부동 소수점 비교) 명령어로 매핑됩니다.4
* **문법:** %result = icmp <cond> <type> %op1, %op2. <cond>는 각 C 연산자에 대해 지정됩니다 (예: eq, ne, sgt, ult). 결과는 항상 i1 불리언 값입니다.

### 4.4 논리 연산과 단락 평가 (&&, ||)

* **직접적인 명령어 없음:** C의 &&와 ||는 단락 평가(short-circuiting) 동작 때문에 and나 or와 같은 단일 LLVM 명령어로 매핑되지 않습니다.21
* **구현 패턴:** 이들은 조건부 분기(br)를 사용하여 구현되어야 합니다. A && B의 경우, IR은 먼저 A를 평가한 다음, br을 사용하여 A가 거짓이면 B의 평가를 건너뛰고, A가 참이면 B의 평가를 포함하는 블록을 실행합니다. 이는 섹션 5에서 코드 예제와 함께 자세히 설명될 것입니다.

### 4.5 타입 캐스팅 및 변환 명령어

이 하위 섹션은 모든 C 스타일 캐스트에 대한 명확한 명령어 목록을 제공합니다.

* trunc: 정수 축소 변환 (예: int에서 char로).23
* zext: 부호 없는 정수 확장 (예: unsigned char에서 int로).12
* sext: 부호 있는 정수 확장 (예: char에서 int로).12
* fptosi/fptoui: 부동 소수점에서 부호 있는/없는 정수로.23
* sitofp/uitofp: 부호 있는/없는 정수에서 부동 소수점으로.23
* bitcast: 값의 비트를 동일한 크기의 다른 타입으로 재해석 (예: int\*에서 float\*로).23
* ptrtoint/inttoptr: 포인터와 정수 간 변환.23

### 4.6 포인터 연산 및 멤버 접근: getelementptr 명령어

* **중심 역할:** getelementptr(GEP) 명령어는 C의 모든 주소 계산을 위한 통합 메커니즘입니다.16 이는 C의 ``(배열 인덱싱),  
  -> 및 .(구조체/공용체 멤버 접근), 포인터 연산(p + i)을 변환합니다.
* **핵심 원칙:** GEP는 주소 계산만 수행하며 **메모리에 접근하지 않습니다**.24 또한 GEP는 타입에 민감하여, 인덱스는 바이트가 아닌 요소를 기준으로 이동합니다. 관련된 타입의 크기는 최종 바이트 오프셋을 계산하는 데 사용됩니다.26
* **문법 및 예제:** %result = getelementptr <type>, ptr %base, <idx\_type> <idx1>,...
  + **배열 인덱싱 (a[i]):** getelementptr inbounds %T, ptr %a, i64 0, i64 %i로 매핑됩니다. 첫 번째 0은 배열 자체에 대한 포인터를 역참조하기 위한 것입니다.
  + **구조체 멤버 접근 (s.f):** getelementptr inbounds %S, ptr %s, i32 0, i32 <field\_index>로 매핑됩니다. 필드는 0부터 시작하는 인덱스로 접근합니다.16
  + **포인터 연산 (p+i):** getelementptr %T, ptr %p, i64 %i로 매핑됩니다.

| 표 4.1: C 연산자와 LLVM IR 명령어/패턴 매핑 |  |  |  |
| --- | --- | --- | --- |
| **C 연산자** | **피연산자 타입** | **LLVM IR 명령어/패턴** | **비고** |
| a + b | 정수 | add i32 %a, %b |  |
| a - b | 정수 | sub i32 %a, %b |  |
| a \* b | 정수 | mul i32 %a, %b |  |
| a / b | 부호 있는 정수 | sdiv i32 %a, %b | 부호에 따라 sdiv 또는 udiv 사용 |
| a % b | 부호 있는 정수 | srem i32 %a, %b | 부호에 따라 srem 또는 urem 사용 |
| a & b | 정수 | and i32 %a, %b |  |
| `a | b` | 정수 | or i32 %a, %b |
| a ^ b | 정수 | xor i32 %a, %b |  |
| ~a | 정수 | xor i32 %a, -1 |  |
| a << b | 정수 | shl i32 %a, %b |  |
| a >> b | 부호 있는 정수 | ashr i32 %a, %b | 부호에 따라 ashr 또는 lshr 사용 |
| a == b | 정수 | %cmp = icmp eq i32 %a, %b | 결과는 i1 타입. fcmp는 부동소수점용. |
| a!= b | 정수 | %cmp = icmp ne i32 %a, %b |  |
| a > b | 부호 있는 정수 | %cmp = icmp sgt i32 %a, %b |  |
| a && b | 불리언 | 조건부 br을 사용한 제어 흐름 | 단락 평가를 보장하기 위함. |
| `a |  |  |  |
| b` | 불리언 | 조건부 br을 사용한 제어 흐름 | 단락 평가를 보장하기 위함. |
| (T)a | 타입 변환 | trunc, zext, sext, bitcast 등 | 변환 종류에 따라 적절한 명령어 사용. |
| a[i] | 배열 | getelementptr | 주소 계산만 수행. |
| s.f | 구조체 | getelementptr | 0-기반 필드 인덱스 사용. |

## 섹션 5: 제어 흐름 추상화

이 섹션은 C의 구조적 제어 흐름 문이 LLVM의 원시적인 제어 흐름 그래프(기본 블록 및 종료 명령어)로 어떻게 분해되는지를 상세히 설명합니다.

### 5.1 조건부 분기: if-else

* **패턴:** if (cond) {... } else {... } 문은 다이아몬드 모양의 제어 흐름 그래프(CFG)로 변환됩니다. 진입 블록은 icmp 또는 fcmp를 사용하여 cond를 평가한 후, 조건부 br i1 %cond, label %true\_block, label %false\_block으로 분기합니다. %true\_block과 %false\_block은 각각의 본문을 포함하며, 무조건부 br label %merge\_block으로 끝납니다.5

### 5.2 루프 구문: for와 while

* **패턴:** 모든 C 루프(for, while, do-while)는 LLVM IR에서 정규 루프 구조로 변환되며, 이는 일반적으로 프리헤더(preheader), 헤더(header), 본문(body), 그리고 종료 블록(exit block)으로 구성됩니다.
  + **프리헤더:** 루프 변수를 초기화하고, 헤더로 무조건 점프합니다.
  + **헤더:** 루프 조건을 확인합니다. 순수 SSA 형태에서는 루프-전달 변수(loop-carried variables)를 위한 phi 노드를 포함합니다. 조건부 br은 본문으로 진입하거나 종료 블록으로 점프합니다.
  + **본문:** 루프의 연산을 포함하며, 헤더로 다시 무조건 점프하여 끝납니다(또는 변수를 증가시킨 후 점프하는 래치 블록).
* **phi의 역할:** for 루프의 i와 같은 변수에 대해, 헤더의 phi 노드는 %i.next = phi i32 [ 0, %preheader ], [ %i.inc, %body ]와 같은 형태가 되어 SSA를 유지하는 데 필수적인 역할을 합니다.4

### 5.3 다중 분기: switch-case

* **명령어:** C의 switch 문은 switch 종료 명령어로 직접 매핑됩니다.23
* **문법:** switch <intty> %value, label %default [ <intty> const, label %dest... ]. 이 명령어는 값과 기본 목적지, 그리고 상수 값과 목적지 레이블 쌍들을 인자로 받습니다. C switch는 각 case에 대한 기본 블록 집합과 이들에게 분배하는 switch 명령어로 변환됩니다.

## 섹션 6: 함수 표현 및 호출 규약

이 섹션은 함수의 정의부터 호출까지의 전체 생명주기를 다룹니다.

### 6.1 함수 정의(define) 및 선언(declare)

* **정의 문법:** define <ret\_type> @name(<arg\_list>) {... }.3
* **선언 문법:** declare <ret\_type> @name(<arg\_list>).3 이는  
  printf와 같은 외부 함수에 사용됩니다.

### 6.2 call 명령어와 인자 전달

* **문법:** %retval = call <ret\_type> @name(<arg\_list>).3
* **인자 전달:** 최소 명세에서는 인자가 값으로 전달됩니다. 복합 타입(구조체)의 경우, 대상 ABI는 이들을 다르게 전달하도록 요구할 수 있으며(예: 스택 또는 포인터를 통해), 이는 byval이나 sret과 같은 속성을 사용하여 LLVM에서 표현될 수 있지만, 기본적인 명령어는 call로 동일합니다.30

### 6.3 ret 명령어와 반환 값

* **문법:** 값을 반환하는 함수는 ret <type> %value, void 함수는 ret void를 사용합니다.4 이는 종료 명령어입니다.

### 6.4 함수 포인터 구현

* **표현:** 함수 포인터는 ptr 타입의 변수입니다.
* **호출:** 함수 포인터를 호출하려면, call 명령어의 피호출자(callee) 피연산자로 사용되어야 합니다. 호출의 시그니처는 호출되는 함수 타입과 일치해야 합니다: %retval = call i32 %func\_ptr(i32 %arg).3

## 섹션 7: 최소 C 언어 IR 명세의 종합

이 결론 섹션은 이전의 모든 섹션을 종합하여 최종적이고 통합된 명세를 제공합니다.

### 7.1 통합된 기능 매핑 및 요약

이 하위 섹션은 모든 C 기능과 그에 해당하는 LLVM IR 표현 간의 연결점을 요약하여, 명시성과 분해라는 핵심 주제를 강화합니다. C의 구조적 제어 흐름은 기본 블록과 분기문의 제어 흐름 그래프로 분해되며, 변경 가능한 변수는 메모리 연산을 통해 모델링되고, 복잡한 표현식은 간단한 3-주소 명령어 시퀀스로 변환됩니다.

### 7.2 최소 명령어 집합

보고서 전반에 걸쳐 상세히 설명된 바와 같이, 전체 C 언어를 표현하는 데 필요한 모든 LLVM IR 명령어 및 타입 생성자의 최종적이고 명확한 목록은 다음과 같습니다.

| 표 7.1: C를 위한 최소 LLVM IR 명령어 집합 |  |
| --- | --- |
| **카테고리** | **명령어 / 타입 생성자** |
| **메모리 연산** | alloca, load, store |
| **종료 명령어** | ret, br, switch |
| **산술 연산** | add, sub, mul, sdiv, udiv, srem, urem |
| **비트 연산** | and, or, xor, shl, lshr, ashr |
| **비교 연산** | icmp, fcmp |
| **타입 변환** | trunc, zext, sext, fptosi, fptoui, sitofp, uitofp, bitcast, ptrtoint, inttoptr |
| **주소 계산** | getelementptr |
| **함수 호출** | call |
| **SSA** | phi (메모리 모델의 SSA 등가물로서) |
| **타입 생성자** | iN, float, double, void, ptr, ``, {...} |

이 표는 "최소 LLVM IR 명세는 무엇인가?"라는 사용자의 질문에 대한 구체적인 답변입니다. C 프론트엔드를 구축하거나 LLVM IR의 의미론적 표현력을 연구하는 모든 이에게 명확하고 실행 가능한 체크리스트를 제공하며, 이 보고서 전체 분석의 정점을 이룹니다.

#### 참고 자료

1. LLVM Language Reference Manual - ROCm Documentation, 9월 27, 2025에 액세스, <https://rocm.docs.amd.com/projects/llvm-project/en/latest/LLVM/llvm/html/LangRef.html>
2. LLVM Documentation - Read the Docs, 9월 27, 2025에 액세스, <https://buildmedia.readthedocs.org/media/pdf/llvm/latest/llvm.pdf>
3. Function Definitions and Declarations — Mapping High Level ..., 9월 27, 2025에 액세스, <https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/basic-constructs/functions.html>
4. An Introduction to LLVM IR Syntax - Medium, 9월 27, 2025에 액세스, <https://medium.com/@mlshark/introduction-to-llvm-syntax-06a262f762eb>
5. Simple “if-then-else” Branching — Mapping High Level Constructs to ..., 9월 27, 2025에 액세스, <https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/control-structures/if-then-else.html>
6. What exactly PHI instruction does and how to use it in LLVM - Stack ..., 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/11485531/what-exactly-phi-instruction-does-and-how-to-use-it-in-llvm>
7. Local Variables — Mapping High Level Constructs to LLVM IR ..., 9월 27, 2025에 액세스, <https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/basic-constructs/local-variables.html>
8. Mapping High Level Constructs to LLVM IR Documentation, 9월 27, 2025에 액세스, <https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/_/downloads/en/latest/pdf/>
9. [LLVMdev] phi instuction example - Google Groups, 9월 27, 2025에 액세스, <https://groups.google.com/g/llvm-dev/c/GKp--MqLVLw>
10. LLVM Language Reference Manual, 9월 27, 2025에 액세스, <https://bcain-llvm.readthedocs.io/projects/llvm/en/latest/LangRef/>
11. Llvm.Types - Haskell.org Downloads, 9월 27, 2025에 액세스, <https://downloads.haskell.org/ghc/7.8.3/docs/html/libraries/ghc-7.8.3/Llvm-Types.html>
12. Other data type — Tutorial: Creating an LLVM Backend for the Cpu0 Architecture, 9월 27, 2025에 액세스, <https://jonathan2251.github.io/lbd/othertype.html>
13. A Quick Primer - Mapping High Level Constructs to LLVM IR - Read the Docs, 9월 27, 2025에 액세스, <https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/a-quick-primer/>
14. Arithmetic and Logic Instructions — Tutorial: Creating an LLVM ..., 9월 27, 2025에 액세스, <https://jonathan2251.github.io/lbd/otherinst.html>
15. High level types · llir/llvm, 9월 27, 2025에 액세스, <https://llir.github.io/document/user-guide/types/>
16. Structures — Mapping High Level Constructs to LLVM IR ..., 9월 27, 2025에 액세스, <https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/basic-constructs/structures.html>
17. Intro to Structures in LLVM and How to Emit a Structure Using MLIR | by Robert K Samuel, 9월 27, 2025에 액세스, <https://medium.com/@60b36t/structures-in-llvm-and-how-to-emit-a-structure-using-mlir-497f5132914e>
18. LLVM's IR Core Concepts - Values, Registers, Memory, 9월 27, 2025에 액세스, <https://blog.piovezan.ca/compilers/llvm_ir_p2/>
19. Global Variables - Mapping High Level Constructs to LLVM IR, 9월 27, 2025에 액세스, <https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/basic-constructs/global-variables.html>
20. How can I declare a global variable in LLVM? - Stack Overflow, 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/7787308/how-can-i-declare-a-global-variable-in-llvm>
21. How Does LLVM Compile 'e1 && e2'? - Juneyoung Lee, 9월 27, 2025에 액세스, <https://aqjune.github.io/posts/2021-10-4.the-select-story.html>
22. Short-circuit evaluation - Wikipedia, 9월 27, 2025에 액세스, <https://en.wikipedia.org/wiki/Short-circuit_evaluation>
23. LLVM Language Reference Manual — LLVM 22.0.0git documentation, 9월 27, 2025에 액세스, <https://llvm.org/docs/LangRef.html>
24. The Often Misunderstood GEP Instruction — LLVM 22.0.0git documentation, 9월 27, 2025에 액세스, <https://llvm.org/docs/GetElementPtr.html>
25. The Often Misunderstood GEP Instruction — LLVM 20.0.0git documentation, 9월 27, 2025에 액세스, <https://rocm.docs.amd.com/projects/llvm-project/en/latest/LLVM/llvm/html/GetElementPtr.html>
26. CS 6120: Making LLVM Address Calculation Safe(r) - Cornell: Computer Science, 9월 27, 2025에 액세스, <https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/gep/>
27. Deep diving into LLVM loop unroll | Interpreting compilers - Yashwant Singh, 9월 27, 2025에 액세스, <https://yashwantsingh.in/posts/loop-unroll/>
28. llvm::SwitchInst Class Reference - LLVM.org, 9월 27, 2025에 액세스, <https://llvm.org/doxygen/classllvm_1_1SwitchInst.html>
29. Control Flow · llir/llvm, 9월 27, 2025에 액세스, <https://llir.github.io/document/user-guide/control/>
30. The mess that is handling structure arguments and returns in LLVM - Yorick Peterse, 9월 27, 2025에 액세스, <https://yorickpeterse.com/articles/the-mess-that-is-handling-structure-arguments-and-returns-in-llvm/>