컴파일러

2025-12-04

TISC (Ternary Symbol Instruction Set Computer) 기반의 언어(IRS)와 컴파일러의 제안

공과대학 컴퓨터공학부 - 202111308 손승우(3 학년)

Part 1. Introduction

본 문서는 3 개의 심볼(I, N, S)만으로 구성된 TISC 구조를 제안하고, 이를 기반으로 한 새로운 프로그래밍 언어인 “INS”의 구조와 수학적 특성을 제시하기 위해 작성되었다.

이를 위해 이 언어모델이 어떻게 순환 테이프(Circular Tape)와 조건부 건너뛰기(Conditional Skip) 메커니즘을 통해 튜링 완전(Turing Complete)함을 달성하고, 이를 해석하는 컴파일러의 제작을 통하여 새로운 언어 시스템을 제시한다.

Part 2. Architecture Definition of TISC(INS)

TISC(Ternary symbol Instruction Set Computer)란, 3 개의 심볼만으로 명령을 수행하는 기계이다. 이는 단 하나의 명령어 만을 제공하는 기계인 OISC(One Instruction Set Computer)와 민스키 머신(Minsky Register Machine)의 철학에서 영감을 받아 본 연구를 통해 새롭게 제안한다.

2.1. Definition of TISC System

TISC 시스템의 엄밀한 정의를 위해, 제안하는 기계(Machine)를 다음과 같은 5-튜플(Tuple)의 수학적 모델로 정의한다.

이 머신은 $M = < T_d, T_p, p, pc, \Sigma >$로 정의한다.

- T_d (Data Tape): 정수($\mathbb{Z}$)를 저장하는 순환 테이프. 튜링 완전성에 근접하기 위해 각 셀은 충분히 큰 모듈러스 M을 갖는 정수 공간 $\mathbb{Z}_m$ 으로 정의한다
- T_p (Program Tape): 명령어들이 저장된 길이 L의 순환 테이프. (pc가 L에 도달하면 0으로 순환됨)
- p (Data Pointer) : T_d 상에서 현재 데이터가 위치한 셀을 가리키는 포인터.
- pc (Program Counter) : T_p상에서 현재 실행할 명령어를 가리키는 인덱스.
- Σ (Alphabet) : 명령어 집합 {I,N,S}

각 심볼은 다음과 같은 단일 동작을 수행한다.

Symbol	Action	Description
I	$Val \leftarrow Val + 1$	현재 포인터가 가리키는 값을 1 증가 (Increment)
N	$P \leftarrow P + 1$	포인터를 오른쪽으로 한 칸 이동 (Move Next)
S	$if\ Val == 0$ $then\ skip\ next$ $(pc \leftarrow pc + 2)$	조건부 건너뛰기 (Skip) 현재 데이터 값(Val)이 0 이면, 바로 뒤에 오는 명령어 하나를 실행하지 않고 무시한다.

2.2. Proof of Turing Completeness

본 장에서는 제안된 TISC 시스템(INS 언어)이 계산 이론적으로 느슨한 튜링 완전(Loose Turing Complete)함을 증명한다. 이는 물리적인 메모리가 유한할 수밖에 없는 현실적인 제약 하에서, 충분히 큰 메모리 공간(M, N)이 주어졌을 때 이 시스템이 튜링 머신과 동등한 계산 능력을 발휘함을 의미한다. 증명은 이 언어가 튜링 완전함이 이미 알려진 2-카운터 민스키 머신(2-Counter Minsky Machine)의 모든 동작을 시뮬레이션할 수 있음[1]을 보이는 Reduction 방식을 통해 수행된다.

Definition 2.2.1 (Operational Semantics)

증명을 위해 기계의 상태 전이를 다음과 같이 정의한다.

1. **순환 실행 (Circular Execution):** 명령어 포인터 pc는 프로그램 테이프의 끝(L)에 도달하면 자동으로 0으로 순환한다. 즉, 프로그램은 명시적인 종료 명령이 없는 한 무한히 반복 실행된다.
2. **조건부 건너뛰기 (Conditional Skip):** 명령어 S 는 현재 포인터가 가리키는 값(Val)이 0 일 때, pc를 2 증가시켜 바로 다음 명령어를 무시(No-op)한다. (0 이 아니면 pc를 1 증가시켜 정상 실행한다.)

[1] Minsky, M. L. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall. p. 255-258. (Chapter 14.1 Universal Program Machines with Two Registers)

Lemma 2.2.1 (Existence of Decrement via Overflow)

기계 M의 명령어 집합 Σ 에는 가산 명령어 I만 존재하지만, 유한한 데이터 값 공간($\mathbb{Z}_m$)의 특성인 오버플로우 (Overflow)를 이용하여 감산 연산 DEC를 구현할 수 있다.

Proof of Lemma 2.2.1

테이프의 각 셀은 0부터 $M-1$까지의 정수를 저장한다. 따라서 연산 결과가 M에 도달하면 정수 오버플로우(Integer Overflow)가 발생하여 0 으로 순환하는 모듈러 연산 $(mod\ M)$이 수행된다. 수학적으로 값 1을 뺀다는 것은, 덧셈 역원인 $M-1$을 더하는 것과 동치이다.

$$Val\ -\ 1 \equiv\ Val\ +\ (M\ -\ 1)\ \ (mod\ M)$$

즉, 명령어 I를 $M-1$회 수행하면 의도적인 오버플로우가 발생하여, 결과적으로 원래 값에서 1이 감소된 값을 얻게 된다. 따라서 가산기(Adder)만으로 감산기(Subtractor)의 기능을 완벽히 대체한다. ■

Definition 2.2.2 (Virtual Program Counter, vPC)

테이프의 특정 셀 $T[0]$를 기계의 상태를 저장하는 가상 프로그램 카운터(virtual Program Counter), vPC로 정의한다.

Lemma 2.2.2 (State-Guarded Execution via Skip)

조건부 건너뛰기 명령어 S 를 사용하여, 특정 상태 k에서만 코드 블록이 유효하게 실행되도록 하는 Guard Mechanism 을 구현할 수 있다.

Proof of Lemma 2.2.2

임의의 연산 A를 상태 k에서만 실행하고 싶다면, 다음과 같은 논리를 구성한다.

1. **플래그 설정:** 실행 여부를 결정할 플래그 변수 F를 1로 설정한다.
2. **상태 검사:** 현재 상태(vPC)와 목표 상태 k의 차이 $\delta\ =\ vPC\ -\ k$를 계산한다.
3. **조건부 무효화:**
    - δ 값을 검사한다.

- 명령어 S를 실행한다.
- S 바로 뒤에 F를 0으로 만드는 명령어를 배치한다.
- 만약 상태가 일치하면 ($\delta = 0$), S 가 작동하여 뒤의 명령어를 건너뛰므로 F는 1로 유지된다.
- 만약 상태가 불일치하면 ($\delta \neq 0$), S가 작동하지 않아 뒤의 명령어가 실행되고 F는 0 이 된다.

4. **조건부 실행:** 연산 A를 F번 반복 수행하도록 루프를 구성한다. ($F = 1$이면 실행, $F = 0$이면 실행하지 않음).

이로써 프로그램이 순환할 때마다 자신의 상태에 맞는 코드만 선별적으로 실행하는 제어 구조가 완성된다. ■

Theorem 2.2.3. (Simulating Minsky Machine)

TISC 기계 M은 임의의 2-카운터 민스키 머신 M_{Minsky}를 시뮬레이션할 수 있다.

Proof of Theorem 2.2.3.

민스키 머신은 두 개의 레지스터 r_1, r_2와 명령어 집합 $\{INC, DEC, JZ\}$로 구성된다. 이를 M으로 다음과 같이 Mapping 한다.

1. 메모리 : 테이프의 특정 셀을 r_1, r_2, vPC로 할당한다.
2. 명령어 시뮬레이션 :
   - $INC(r), DEC(r)$: Lemma 2.2.2 의 가드 블록 내부에서 포인터 이동(N) 후 I 또는 Decrement Operation (Lemma 2.2.1)을 수행한다. 이후 vPC를 다음 값으로 갱신한다.
   - $JZ(r, target)$: 레지스터 값이 0인지 검사하여(플래그 로직 활용), 0이면 vPC를 타겟 주소로, 아니면 다음 주소로 갱신한다. (Lemma 3.2)
3. 실행 모델 : 프로그램 테이프 T_p는 무한히 회전하며, 매 회전마다 vPC 와 일치하는 단 하나의 논리적 블록만이 활성화되어 민스키 머신의 한 스텝을 수행한다.

모든 필수 구성 요소가 M상에서 구현 가능하므로, $M \cong M_{Minsky}$(계산적으로 동등)이다. ■

Corollary 2.2.4. (Loose Turing Completeness)

TISC(INS)는 $M \cong M_{Minsky} \cong Turing\ Machine$ 에 의하여[2] Universal Turing Machine 과 계산적으로 동등하며, 메모리 크기 M, N이 입력 문제의 복잡도에 비례하여 충분히 크다면, 튜링 머신이 수행할 수 있는 모든 계산을 수행할 수 있다. 이는 선형 유계 오토마타(LBA)의 상한을 가지면서도 논리적으로는 튜링 완전함을 의미한다. 이로써 TISC(INS)는 이론적으로 현대 컴퓨터가 수행 가능한 모든 알고리즘을 처리할 수 있다.

Definition 2.2.5. (Halting Condition)

제안된 기계는 순환 테이프 구조로 인해 물리적으로 영원히 실행(Infinite Execution)된다. 따라서 계산의 정지(Computation Halting)는 기계의 물리적 정지가 아닌, 가상 프로그램 카운터(vPC, 즉 $T[0]$)가 특정한 종료 상태(Terminal State, H)에 도달하는 것으로 정의한다. 기계가 상태 H에 도달하면 더 이상의 유효한 상태 전이를 수행하지 않고 결과값을 보존하는 'Trap State'에 진입한 것으로 간주하여, 이를 정지(Halt)로 정의한다.

Definition 2.2.6. (Language Acceptance)

입력 w에 대하여 기계가 유한한 시간 t 후에 논리적 정지 상태(State H)에 도달하고, 그때 지정된 결과 셀(예: $T[1]$)의 값이 1이라면 기계는 입력을 Accept 한 것으로 간주한다. 반면, 기계가 영원히 상태 H에 도달하지 못하거나(Infinite Loop), 정지했을 때 결과 셀의 값이 0이라면 입력을 거부(Reject)한 것으로 간주한다.

정리

1. 기계가 유한한 시간 내에 상태 H에 도달하고 결과값이 1 이면, 입력을 수락(Accept)한다.
2. 기계가 유한한 시간 내에 상태 H에 도달하고 결과값이 0 이면, 입력을 거부(Reject)한다.
3. 기계가 영원히 상태 H에 도달하지 못한다면(Infinite Loop), 이 또한 입력을 거부(Reject)한 것으로 간주한다.

[2] Recursive Unsolvability of Post's Problem of 'Tag' and other Topics in Theory of Turing Machines" (Marvin Minsky, 1961)

[Note]

1. 튜링 완전한 시스템의 특성상(Halting Problem), 임의의 입력에 대해 기계가 정지할지 여부를 사전에 판별하는 것은 불가능하다.[3] 본 정의는 기계가 정지하지 않는 경우를 포함하여 '언어의 인식(Recognition)'을 정의한 것이며, 이는 표준 튜링 머신의 정의와 일치한다.
2. 민스키 머신의 'HALT' 명령어는 TISC 시스템에서 vPC를 종료 상태 H로 설정하는 동작으로 매핑된다. 이후 기계는 물리적으로는 계속 회전하지만, vPC가 H이므로 더 이상 어떠한 가드 블록(Guarded Block)도 활성화되지 않아(모두 Skip 되어) 데이터 테이프의 값이 변하지 않는 안정 상태(Stable State)를 유지하게 된다.

2.3. Comparative Analysis of TISC Architecture

본 절에서는 제안된 TISC (INS) 시스템의 아키텍처적 특성을 규명하기 위해, 기존의 극한적 축소 명령어 집합 컴퓨터(OISC) 및 난해 프로그래밍 언어(Esoteric Programming Language)들과 비교 분석한다. 특히 명령어의 구조, 메모리 접근 방식, 그리고 제어 흐름의 메커니즘을 중점적으로 다룬다.

2.3.1. Comparison with OISC (Subleq)

가장 대표적인 OISC 인 subleq[4] (Subtract and Branch if Less than or Equal to Zero)와 TISC 를 비교하면, 명령어의 개수보다는 명령어의 밀도(Density)와 Operand 의 유무에서 근본적인 차이가 발생한다.

1. **피연산자의 부재 (Zero-Operand Architecture):**
   - **OISC:** 단 하나의 명령어(subleq)를 가지지만, 동작을 위해 3 개의 피연산자(A, B, C)를 필요로 한다. 즉, 하나의 명령어가 [Opcode + Address1 + Address2 + JumpTarget]의 패킷 구조를 갖는다.

[3] Turing, A. M. (1936). "On Computable Numbers, with an Application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society*.

[4] Mazonka, O., & Kolodin, A. (2011). "A Simple Multi-Processor Computer Based on Subleq". *arXiv preprint arXiv:1106.2593*.

- **TISC (INR):** I, N, S 세 개의 심볼은 그 자체로 완결된 명령어이며, 피연산자를 갖지 않는다(Zero-Operand). 이는 하드웨어 디코더의 복잡성을 획기적으로 낮춘다.

2. **메모리 접근 패러다임 (Random vs. Sequential):**
   - **OISC:** 폰 노이만 구조의 전형인 임의 접근(Random Access) 방식을 따른다. 임의의 메모리 주소 번지를 즉시 참조할 수 있다.
   - **TISC (INR):** 튜링 머신 기반의 순차 접근(Sequential Access) 방식을 따른다. 특정 주소로 이동하기 위해서는 포인터 이동 명령어(N)를 물리적으로 수행해야 한다. 이는 하드웨어의 주소 버스(Address Bus)를 제거하고 단순한 카운터만으로 구현 가능함을 의미한다.

2.3.2. Comparison with Brainfuck

TISC 는 테이프와 포인터를 사용한다는 점에서 Brainfuck 언어와 유사성을 갖지만, 제어 흐름(Control Flow)의 구현 방식에서 차이가 있다.

- **Brainfuck:** 중첩 가능한 루프 명령어 [,]를 제공한다. 이는 "현재 값이 0 이면 짝이 맞는 괄호 뒤로 점프"하는 Local 분기를 수행하며, 스택 구조가 필요하다.
- **TISC (INS):** 오직 조건부 건너뛰기 명령어 S만을 제공한다. S는 현재 데이터 값(Val)이 0 이면, 바로 뒤에 오는 명령어 하나를 실행하지 않고 무시한다.

2.3.4. Summary of Characteristics

다음 표는 TISC 시스템이 기존 시스템들과 구별되는 구조적 특징을 요약한다.

Feature	OISC (Subleq)	Brainfuck	TISC (INS)
Instruction Count	1	8	**3**
Operands per Inst.	3 (Source, Dest, Jump)	0	**0**
Memory Access	Random Access	Sequential (Tape)	**Sequential (Circular Tape)**
Control Flow	Conditional Jump (Arbitrary)	Structured Loop (Stack)	**Conditional Skip (Local/State-based)**

Hardware Reqs	ALU + Address Bus	Counter + Stack	**Counter (No Stack)**

Part 3. Compiler Design and Implementation

본 장에서는 앞서 제안된 TISC (INS) 아키텍처를 실제 x86-64 환경에서 동작 가능한 기계어로 변환하는 컴파일러의 설계 및 구현 상세를 다룬다. 컴파일러는 어휘 분석(Lexical Analysis), 구문 분석(Syntax Analysis), 코드 생성(Code Generation)의 3 단계 파이프라인으로 구성된다.

3.1. Overall Architecture

INS 컴파일러는 소스 코드를 입력받아 x86-64 Assembly 코드를 출력하는 전방(Frontend) 컴파일러 구조를 갖는다.

- **Scanner:** 소스 코드 스트림을 읽어 유효한 토큰(I, N, S)으로 분리하고 불필요한 공백 및 기타 문자는 무시(Ignore)한다.
- **Parser:** 토큰 시퀀스를 검증하고 추상 구문 트리(AST)를 구성한다.
- **Code Generator:** AST 를 순회하며 TISC 의 논리적 동작을 등가의 x86 명령어로 변환한다.

3.2. Lexical and Syntax Analysis

어휘 분석기(scanner.l)는 입력 스트림에서 [iI], [nN], [sS] 정규 표현식에 매칭되는 문자를 찾아 각각 TOKEN_I, TOKEN_N, TOKEN_S 로 변환한다. 구문 분석기(parser.y)는 문맥 자유 문법(Context-Free Grammar)을 기반으로 명령어의 리스트를 순차적 AST 노드로 구성하며, 이때 각 명령어는 정수형 Opcode (I=1, N=2, S=3)로 매핑되어 저장된다.

3.3. Code Generation and Memory Model

코드 생성기(codegen_x86.c)는 본 컴파일러의 핵심 모듈로, 이론적 TISC 모델을 물리적 하드웨어에 매핑한다.

3.3.1. Memory Mapping (8-bit Data Tape Model)

Proof 2.2.1 에서 제시된 '오버플로우를 이용한 감산(Existence of Decrement via Overflow)'을 현실적인 시간 내에 수행하기 위해, 데이터 테이프(T_d)는 8-bit 정수 공간(Z_{256})으로 모델링되었다.

[Remark]

엄밀한 튜링 완전성은 무한한 테이프와 무한한 정수 집합을 요구하나, 본 구현에서는 물리적 제약과 실행 효율성을 위해 8-bit 모델을 채택하였다. 따라서 구현된 시스템은 이론적 TISC 모델의 유한한 부분집합(Finite Subset)을 시뮬레이션하는 것으로 한정된다. 이론적 TISC 모델은 Proof 2.2.3 과 그에 따른 Collorary 2.2.4 에 의하여 튜링 완전성을 증명하였다.

- **Data Tape :** .data 섹션에 65,536 바이트 크기의 data_tape 배열을 할당하고 0 으로 초기화한다.
- **Register Mapping :** %r15 는 데이터 테이프의 베이스 주소, %r13 은 데이터 포인터(p), %r12 는 프로그램 카운터(pc)로 매핑된다.

3.3.2. Instruction Translation

각 INS 명령어는 x86 명령어와 1:1 또는 1:N 관계로 변환된다.

- **I (Increment) :** 8-bit 데이터 모델에 따라 incb (%r15, %r13, 1) 명령어를 사용하여 1 바이트 단위 연산을 수행한다. 이는 $255 + 1 \rightarrow 0$ 의 오버플로우를 자연스럽게 유도한다.
- **N (Move Next) :** incq %r13 후 andq $0xFFFF, %r13 을 수행하여, 포인터가 테이프 크기(65,536)를 벗어날 경우 0 번지로 순환하도록 구현하였다.
- **S (Conditional Skip) :** movzbl 명령어로 현재 포인터의 1 바이트 값을 읽어와 0 인지 검사한다. 조건 만족 시 %r12(pc)에 2 를 더하여 다음 명령어를 건너뛴다.

3.4. Implementation Constraints and Limitations

이론적 모델(Theoretical Model)을 유한한 자원을 가진 실제 컴퓨터(Physical Machine)로 옮기는 과정에서, 논리적 정합성을 유지하기 위해 다음과 같은 제약 사항을 정의하였다.

1. **Practicality of Decrement**
   Lemma 2.2.1 에 따르면 값 1 을 감소시키기 위해 $M - 1$회의 가산이 필요하다. 64-bit 정수 체계($M = 2^{64}$)에서는 이 연산이 천문학적인 시간을 소요하여 사실상 실행 불가능하다. 본 구현에서는 데이터 셀을 8-bit($M = 2^8 = 256$)로 제한하여, 255 회의 가산으로 감산을 수행할 수 있도록 최적화하였다.
2. **물리적 무한성과 논리적 정지 (Definition 2.2.5)**
   TISC 는 물리적으로 멈추지 않는 기계이다. 이를 구현하기 위해 컴파일러는 pc가 프로그램 끝에 도달하면 다시 0 으로 초기화하는 순환 로직을 삽입하였다. 대신, 논리적 정지(Halt)를 감지하기 위해 매 사이클마다 가상 프로그램 카운터($T[0]$)를 검사한다. 본 구현에서는 종료 상태 H를 255 (0xFF)로 정의하였으며, $T[0] == 255$가 되는 순간 에뮬레이터는 실행을 중단하고 결과를 출력한다.
3. **운영체제 호환성 (OS Compliance)**
   Windows x64 환경(MSYS2)에서의 안정적인 동작을 위해 Microsoft x64 호출 규약을 준수하였다. 함수 프롤로그에서 subq $40, %rsp 를 수행하여 Shadow Space 를 확보함으로써 외부 입출력 함수(printf) 호출 시의 스택 오염을 방지하였다.

Part 4. Experiment and Verification

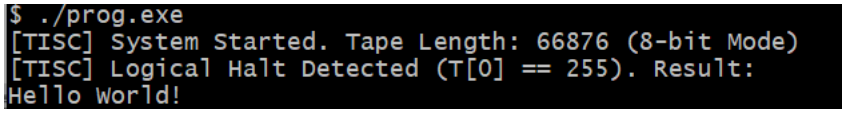
4.1. Experimental Environment

본 컴파일러의 검증은 다음 환경에서 수행되었다.

- **OS:** Windows 10 x64
- **Environment:** MSYS2 UCRT64
- **Assembler/Linker:** GNU Assembler (Gas), GCC

4.2. Execution Results

"Hello World!" 문자열을 출력하는 compliant_hello.ins 프로그램을 작성하여 컴파일 및 실행하였다.
이 프로그램은 문자열 출력 후 포인터를 순환시켜 $T[0]$에 값 255 를 기록하도록 설계되었다.
실행 결과, 시스템은 다음과 같이 테이프 길이(66,876)를 초기화한 후 물리적 루프를 수행하다가,
논리적 종료 상태($T[0] == 255$)를 감지하고 정상적으로 메시지를 출력한 뒤 종료되었다.



5. Conclusion and Contributions

본 연구는 3 개의 심볼(I, N, S)만으로 구성된 초소형 명령어 집합 컴퓨터인 **TISC (INS)** 아키텍처를 제안하고, 이를 해석하는 최적화된 컴파일러를 구현함으로써 이론적 계산 모델을 실제 컴퓨팅 환경에서 실증하였다. 본 프로젝트의 학술적 성과와 교육적 가치는 다음과 같이 요약된다.

5.1. Academic Achievements

첫째, **미니멀리즘 컴퓨팅 모델의 이론적 확립 및 증명**을 달성하였다. OISC 와 민스키 머신에서 영감을 받아 설계된 TISC 시스템은 단 3 개의 명령어와 피연산자가 없는(Zero-Operand) 구조만으로도 튜링 완전(Turing Complete)함을 보였다. 특히, 조건부 건너뛰기(Conditional Skip) 메커니즘을 통해 제어 흐름을 구현하고, 오버플로우를 이용한 감산(Decrement via Overflow)을 수학적으로 증명함으로써, 제한된 명령어 집합이 계산 복잡도를 제약하지 않음을 확인하였다.

둘째, **이론과 현실의 간극(Gap)을 해결하는 시스템 설계**를 제시하였다. 무한한 자원을 가정하는 이론적 튜링 머신을 유한한 물리적 하드웨어(x86-64)에 매핑하기 위해, 순환 테이프(Circular Tape) 모델과 8-bit 데이터 모델을 도입하였다. 이는 2^{64}회의 연산이 필요한 64-bit 감산의 비효율성을 2^{8}회 연산으로 단축시킴으로써, 이론적 증명을 현실적인 시간 내에 동작 가능한 프로그램으로 구현해냈다는 점에서 기술적 의의가 있다.

셋째, **순환 실행 모델에서의 정지 문제(Halting Problem) 재정의**이다. 물리적으로 영원히 회전하는 기계적 특성을 유지하면서도, 가상 프로그램 카운터(vPC)의 상태 감지를 통해 논리적 정지(Logical Halt)를 판별하는 하이브리드 실행 모델을 정립하였다. 이는 추상적인 정지 문제를 구체적인 구현 레벨에서 어떻게 다룰 수 있는지에 대한 새로운 시각을 제시한다.

5.2. Educational Benefits

본 프로젝트는 컴퓨터 과학 교육 측면에서 다음과 같은 이점을 제공한다.

- **컴파일러 파이프라인의 전 과정 체득:** 어휘 분석(Scanner), 구문 분석(Parser), 코드 생성(Code Generator)으로 이어지는 컴파일러의 표준 파이프라인을 직접 설계하고 구현함으로써 언어 처리기의 내부 동작 원리를 깊이 있게 이해할 수 있었다.
- **시스템 레벨 프로그래밍 능력 함양:** 추상 구문 트리(AST)를 x86-64 어셈블리어로 변환하는 과정에서 레지스터 할당, 스택 프레임 관리, 그리고 운영체제별 호출 규약(Windows x64 ABI/Shadow Space) 준수 등 저수준 시스템 프로그래밍의 핵심 개념을 습득하였다.
- **계산 이론의 구체화:** 추상적으로만 다루어지던 Turing-Completeness 를 직접 제작한 언어체계에 대해 Reduction(환원)을 통해 증명하며, 이를 직접 동작하는 코드로 구현해봄으로써, "계산 가능하다(Computable)"는 것의 의미를 공학적 관점에서 체감하는 계기가 되었다.

결론적으로, 본 TISC (INS) 프로젝트는 극한의 제약 조건을 가진 난해 프로그래밍 언어(Esoteric Language)를 설계하고 이를 현대적인 컴파일러 기술로 구현해냄으로써, 컴퓨터 구조와 계산 이론, 그리고 시스템 프로그래밍을 아우르는 융합적 사고 능력을 증진시키는 데 기여하였다.

문서 끝.