情報工学実験 II 報告書

	点

実験クール (1・2・3・4)	1	実験回 (1・2・3)	2	クラス (A・B)	A

実験番号 4　実験題目 **簡単なコンパイラ**

グループ (A・B・C・D)	D	学籍番号	EP20050	氏名	小池 正基

	実験実施日	実験項目（実施概要）
1	2023／10／02	LLVM 中間言語の概要，課題 1
2	2023／10／16	課題 2
3	／　／	

共同実験者（学籍番号）

	中間①提出日	中間②提出日	完成提出日	完成再提出①	完成再提出②
提出期日（予定日）	2023／10／10	2023／10／17	20　／　／	20　／　／	20　／　／
提出日	2023／10／09	2023／10／16	20　／　／	20　／　／	20　／　／
ルーブリック評価点					

コメント

1　実験の目的

本実験では，極めて小規模な（とはいえ十分に汎用的な）C 言語サブセットのコンパイラを LLVM コンパイラ基盤と C 言語を用いて実現することで，

- コンパイラが原始言語を中間言語に翻訳する仕組みを具体的に理解する
- 広く利用されている LLVM コンパイラ基盤の中間言語の初歩を理解する
- C 言語についてよりよく理解し使いこなせるようになる

ことを目的とする．

2　実験の前提知識（理論）と環境

2.1　実験環境

本実験では Windows11 の WSL を用いた仮想環境の Ubuntu22.04 LTS を用いる．LLVM 中間言語の実行には対応したバックエンドを持つ C 言語コンパイラが必要であるため，clang-15 のインストールを行う．clang-15 及び LLVM ツールチェーンのインストールには以下のコマンド 1 を Windows Terminal で実行した．

Source Code 1: 環境構築時に用いたコマンド

```
1                sudo apt update && sudo apt install -y wget curl git make gcc cgdb
                 ↪  clang-15 llvm-15 graphviz valgrind global w3m bc xsel
                 ↪  librsvg2-bin nkf
```

コマンド 1 は，sudo により管理者権限を付与した後，apt update によりインストールされているソフトウェアの更新を行い， apt install によって管理者権限を付与した状態でいくつかのアプリのインストールを行うコマンドである．-y オプションは全ての選択肢に’yes’ と回答するものである．以下にインストールしたアプリの一覧と簡単な説明を示す．

wget　：ファイルをダウンロードするためのコマンドラインツール．
curl　：URL からデータを転送するコマンドラインツール．
git　：バージョン管理システム．
make　：ビルド自動化ツール．

gcc ：C コンパイラ．
cgdb ：C デバッガ．
clang-15 ：LLVM プロジェクトを実行できる C 言語コンパイラ（バージョン 15）．
llvm-15 ：LLVM コンパイラインフラストラクチャ（バージョン 15）．
graphviz ：グラフ描画ソフトウェア．
valgrind ：C 言語による不正なメモリアクセスの検出ツール．
global ：ソースコードタグシステム．
w3m ：テキストベースの Web ブラウザ．
bc: ：任意精度の計算機言語．
xsel ：GUI と CUI のクリップボードを連携するツール．
librsvg2-bin ：SVG ファイルを他のフォーマットに変換するツール．
nkf ：日本語文字コード変換ツール．

2.2 実験で用いるツール

本実験では数行〜数百行のプログラム編集が必要であるため，エディタに VScode を用いることとする．

2.3 原始言語（Pico 言語）の仕様

Pico 言語とは，ブリュッセル大学のソフトウェア言語研究室で学生の学習向けに開発されたプログラミング言語である．Pico とは‘小さい’の意であり，その名前の通り簡素さに重点が置かれている．例えば，条件分岐に if のみが用いられている点，データ型に文字列，整数，実数，テーブルの 4 種類しかない点などが挙げられる [1]．

2.4 LLVM 中間言語の仕様

C 言語などのプログラムファイルをコンパイルする際，機械語に直接変換することは稀であり，基本的には中間言語と呼ばれる，高水準言語と低水準言語の橋渡しを行うプログラムに変換される．このときに用いられている中間言語の一つが LLVM 中間言語（LLVM Intermediate Representation：LLVM-IR）である．この LLVM-IR をコンパイラ LLVM によって機械語に翻訳することで，機械語で書かれたプログラムに変換することができる．

LLVM を用いるプログラム言語には go や Swift などがあるが，任意のプログラム言語に対応可能なコンパイラ基盤であるため，コンパイラしたい言語を LLVM-IR に変換することができれば，その後の機械語へのコンパイルまでを自動で実行することができる．この

性質から，独自のコンパイラを作成する場合非常に有用であり，コンパイラが原始言語を中間言語に翻訳する仕組みを理解する上で最適である．

本実験では，Pico 言語で書かれたプログラムを LLVM-IR のコードに変換して出力するコンパイラを C 言語で作成する．

3 課題 1

以下に示す C 言語のコード 2 は正整数 n の（正の）約数を計算する．これを LLVM-IR のコードになおせ．ただし，LLVM-IR のコードでは load 命令は 2 回まで使ってよい．でき上がった LLVM-IR のコードはコンパイル・実行してテストせよ．

Source Code 2: div.c

```c
#include <stdio.h>
#include <limits.h>

int main()
{
    int n = 12;           // 典型的な例
    //int n = 1;            // 極端に小さな例
    //int n = INT_MAX - 1; // 極端に大きい例
    //int n = INT_MAX;      // どんな結果になるか？理由も考察してみるとよい
    //int n = 99999989;     // 素数の例
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) printf("%d\n",i);
    }
    return 0;
}
```

1-1.

問題

div.c をフローチャートを用いて書き直せ．

回答

div.c の処理をフローチャートを用いて記述したものを図 1 に示す．また，図 1 のフローチャートを goto 文を用いたフローチャートに変更したものを図 2 に示す．

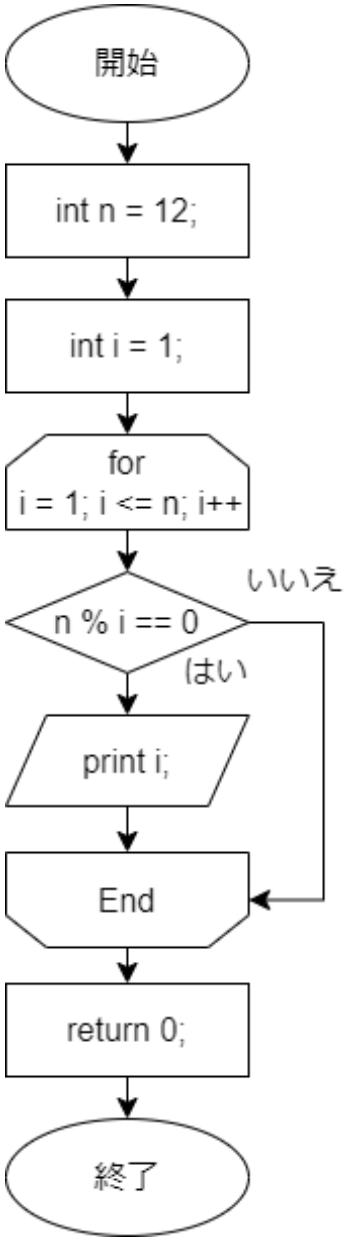

図 1: div.c のフローチャート

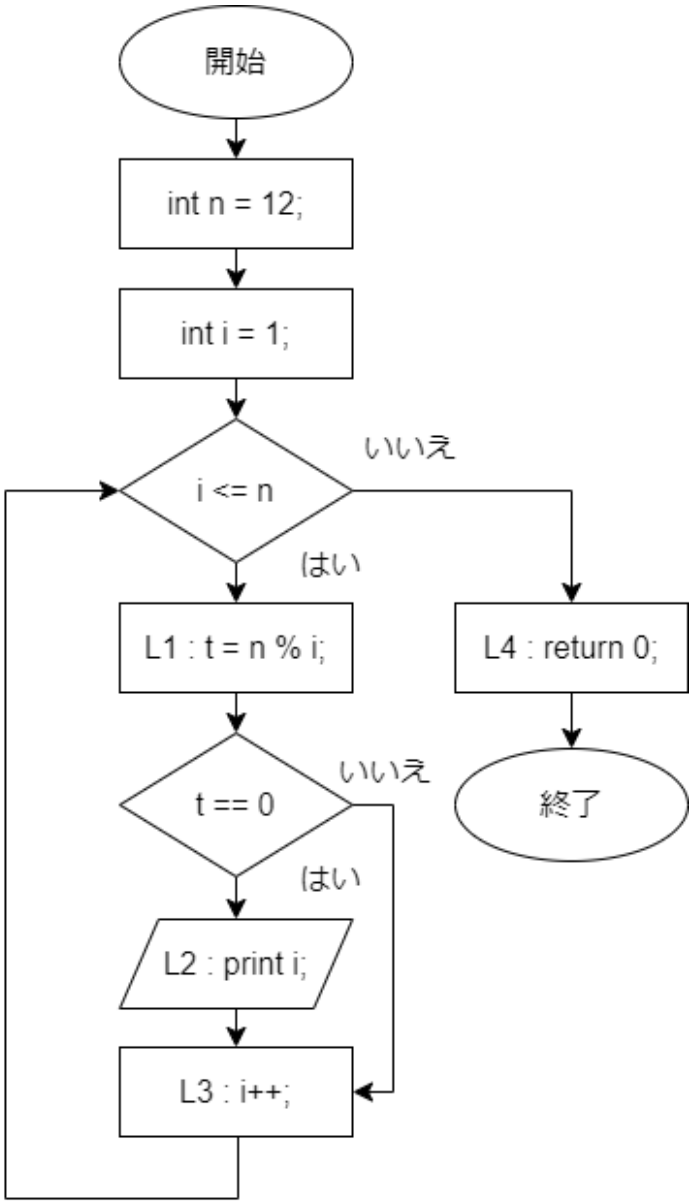

図 2: goto を用いた div.c のフローチャート

解説

div.c では正整数 n の正の約数を求めるため，n を 1〜n までの数値で割ったときの余りが 0 ならその数値を出力する．余りが 0 以外ならば今回のループを終了し，新しい数値で再度ループする．

1-2.

問題

フローチャートを goto 文で表した C 言語のコードに書き換えよ．

回答

goto 文を用いた正整数 n の正の約数を計算するプログラムをコード 3 に示す．

Source Code 3: kadai1-2.c

```c
#include <stdio.h>
#include <limits.h>

int main(){
    int n = 12;
    int i = 0;
    int t;
    goto L1;

L1:
    i++;
    if(i <= n) goto L2; else goto L4;

L2:
    t = n % i;
    if(t == 0) goto L3; else goto L1;

L3:
    printf("%d\n",i);
    goto L1;

L4:
    return 0;
}
```

解説

コード 3 では，L1 で for 文のループ判定を行い，L2 で if 文の真偽値判定を行っている．true だった場合は L3 で結果の出力を行い，false だった場合は L1 のループ判定に移る．最終的にループが終了したら L4 に移動し，処理を終える．

1-3.

問題

1-2 のコードをもとに，LLVM-IR のコードを作成し，CFG として図示せよ．

回答

LLVM-IR で記述した正整数 n の正の約数を計算するプログラムをコード 4 に示す．

Source Code 4: kadai1-3.ll

```
1  declare i32 @printf(ptr, ...)
2  @fmt = private constant [4 x i8] c"%d\0A\00"
3
4  define i32 @main() {
5      %n = alloca i32
6      store i32 12, ptr %n
7      %i = alloca i32
8      store i32 0, ptr %i
9      br label %L.1
10
11 L.1:
12 ;for 冒頭
13     %t.1 = load i32, ptr %n
14     %t.2 = load i32, ptr %i
15     %t.3 = add  i32  %t.2, 1
16     store  i32  %t.3,ptr %i
17     %t.4 = icmp sle i32 %t.3, %t.1
18     br i1 %t.4, label %L.2, label %L.4
19
20 L.2:
21     %t.5 = load i32, ptr %n
22     %t.6 = load i32, ptr %i
23     %t.7 = srem i32  %t.5, %t.6
24     %t.8 = icmp eq   i32  %t.7, 0
25     br i1 %t.8, label %L.3, label %L.1
26
```

```
27  L.3:
28      %t.9 = load i32, ptr %i
29      %t.10 = getelementptr [4 x i8], ptr @fmt, i32 0, i32 0
30      %t.11 = call i32 (ptr, ...) @printf(ptr %t.10, i32 %t.9)
31      br label %L.1
32
33  L.4:
34  ; for直後
35      ret i32 0
36  }
```

コード 4 を CFG として図示したものを図 3 に示す.

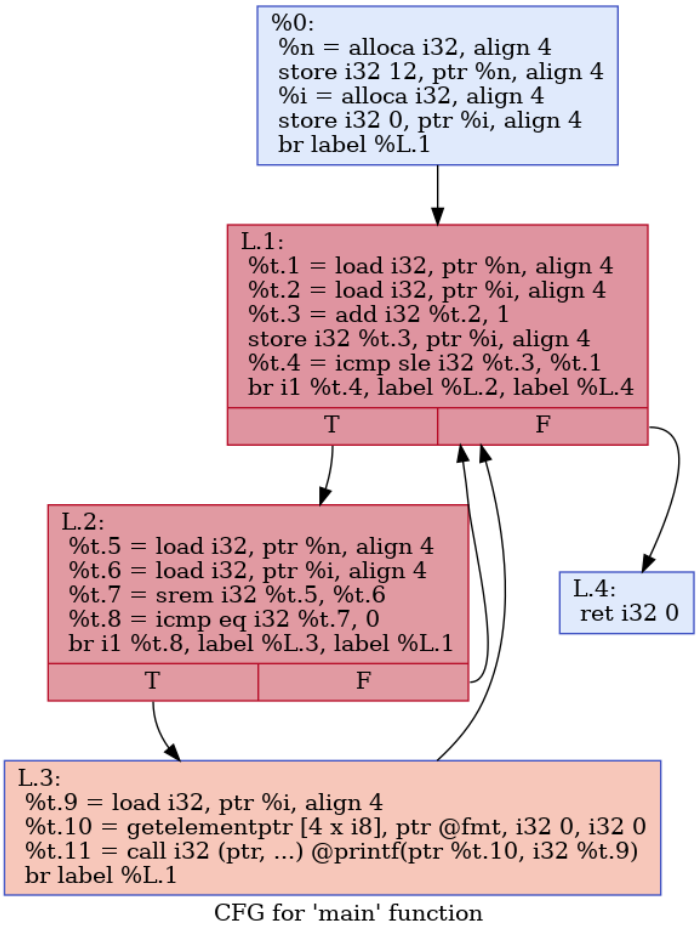

図 3: kadai1-3.ll の CFG

解説

コード 4 では，alloca を用いてローカル変数の割り当てを行い，その変数の値を都度新しい一時変数に読み込むことで参照，計算している．ptr はデータのアドレスであり，変数の値の変更が必要な場合は，store を用いて更新を行う．for,if 文は icmp を用いた条件分岐のみで構成されており，L1 が for の判定，L2 が if 文の判定を行っている．L3 が約数の出力を担い，@fmt 配列へのポインタを格納した%t.10 を用いて call によって外部関数 printf を呼び出している．for 文が終了したら L4 に移動し，ret 0 によって処理を正常に終了する．

1-4.

問題

CFG をもとに変数 i,n の流れを追い，不要な load 命令の使用箇所と不要な理由を指摘せよ．

回答

CFG を示す図 3 から，%t.5，%t.6，%t.9 が不要である．

解説

CFG を示す図 3 を見ると，変数%n,%i が複数の基本ブロック内で何回も load されていることがわかる．しかし，ローカル変数 n,i は一度一時変数に割り当てれば以降呼び出す必要はなく，今回の場合は%n を%t.1，%i を%t.2 として利用可能である．さらに言えば，変数内の値の変化があり複数回 load しなければならない%i と違い，%n は変化しないため%t.1 も不要であるが，今回は視認性を重視し省かない．

1-5.

問題

不要な load 命令を除去した完成版の LLVM-IR コードを示せ。

回答

不要な load 命令を省き，コード内での load 命令を 2 回だけ行うよう改良したプログラムをコード 5 に示す．

Source Code 5: kadai1-5.ll

```
declare i32 @printf(i8*, ...)
@fmt = private constant [4 x i8] c"%d\0A\00"

define i32 @main() {
    %n = alloca i32
    store i32 99999989, i32* %n
    %i = alloca i32
    store i32 0, i32* %i
    br label %L.1

L.1:
    %t.1 = load i32, i32* %n
    %t.2 = load i32, i32* %i
    %t.3 = add i32 %t.2, 1
    store i32 %t.3, i32* %i
    %t.4 = icmp sle i32 %t.3, %t.1
    br i1 %t.4, label %L.2, label %L.4

L.2:
    %t.5 = srem i32 %t.1, %t.3
    %t.6 = icmp eq i32 %t.5, 0
    br i1 %t.6, label %L.3, label %L.1

L.3:
    %t.7 = getelementptr [4 x i8], [4 x i8]* @fmt, i32 0, i32 0
    %t.8 = call i32 (i8*, ...) @printf(i8* %t.7, i32 %t.3)
    br label %L.1

L.4:
    ret i32 0
}
```

解説

問題 1-4 で言及した通りに%t.5，%t.6，%t.9 を省くことで，load 命令が 2 つに減り，一時変数の数も 3 つ減少した．

1-6.

問題

正整数 n の（正の）約数を正しく計算していることを網羅的にテストせよ。

テスト手順

正整数 n の正の約数を正しく計算できているか確認するため，以下の数値を n に代入し，テストを行う．

- n = 1
- n = 12
- n = 15
- n = 99999989
- n = INT_MAX - 1 = 2147483646
- n = INT_MAX = 2147483647

テスト結果

テスト結果をコード 6，7，8，9，10，11 に示す．ここで，出力が多いプログラムの実行結果は一部表示形式を改変して示すものとする．

Source Code 6:
n=1 の実行結果

```
:~/jikkenii/work$ lli-15
↪  kadai1-5.ll;
1
```

Source Code 7:
n=12 の実行結果

```
:~/jikkenii/work$ lli-15
↪  kadai1-5.ll;
1
2
3
4
6
12
```

Source Code 8:
n=15 の実行結果

```
:~/jikkenii/work$ lli-15
↪  kadai1-5.ll;
1
3
5
15
```

Source Code 9:

n=99999989 の実行結果

```
:~/jikkenii/work$ lli-15 kadai1-5.ll;
1
99999989
```

Source Code 10:

n=2147483646 の実行結果

```
:~/jikkenii/work$ lli-15 kadai1-5.ll;
1, 2, 3, 6, 7, 9, 11, 14, 18, 21, 22, 31,
33, 42, 62, 63, 66, 77, 93, 99, 126,
151, 154, 186, 198, 217, 231, 279,
302, 331, 341, 434, 453, 462, 558,
651, 662, 682, 693, 906, 993, 1023,
1057, 1302, 1359, 1386, 1661, 1953,
1986, 2046, 2114, 2317, 2387, 2718,
2979, 3069, 3171, 3322, 3641, 3906,
4634, 4681, 4774, 4983, 5958, 6138,
6342, 6951, 7161, 7282, 9362, 9513,
9966, 10261, 10923, 11627, 13902, 14043,
14322, 14949, 19026, 20522, 20853,
21483, 21846, 23254, 25487, 28086,
29898, 30783, 32767, 32769, 34881,
41706, 42129, 42966, 49981, 50974,
51491, 61566, 65534, 65538, 69762,
71827, 76461, 84258, 92349, 98301,
99962, 102982, 104643, 112871, 143654,
149943, 152922, 154473, 184698, 196602,
209286, 215481, 225742, 229383, 294903,
299886, 308946, 338613, 349867, 360437,
430962, 449829, 458766, 463419, 549791,
589806, 646443, 677226, 699734, 720874,
790097, 899658, 926838, 1015839, 1049601,
1081311, 1099582, 1292886, 1549411,
1580194, 1649373, 2031678, 2099202,
2162622, 2370291, 3098822, 3148803,
3243933, 3298746, 3848537, 4648233,
4740582, 4948119, 6297606, 6487866,
7110873, 7697074, 9296466, 9896238,
10845877, 11545611, 13944699, 14221746,
17043521, 21691754, 23091222, 27889398,
32537631, 34087042, 34636833, 51130563,
65075262, 69273666, 97612893, 102261126,
119304647, 153391689, 195225786,
238609294, 306783378, 357913941,
715827882, 1073741823, 2147483646,
```

Source Code 11: n=2147483647 の実行結果

```
:~/jikkenii/work$ lli-15 kadai1-5.ll;
1
2147483647
-2147483647
-1
PLEASE submit a bug report to https://github.com/llvm/llvm-project/issues/ and include the crash backtrace.
Stack dump:
0.  Program arguments: lli-15 kadai1-5.ll
Stack dump without symbol names (ensure you have llvm-symbolizer in your PATH or set the environment var
`LLVM_SYMBOLIZER_PATH` to point to it):
/usr/lib/llvm-15/bin/../lib/libLLVM-15.so.1(_ZN4llvm3sys15PrintStackTraceERNS_
11raw_ostreamEi+0x31)[0x7f91214043b1]
/usr/lib/llvm-15/bin/../lib/libLLVM-15.so.1(_ZN4llvm3sys17RunSignal
HandlersEv+0xee)[0x7f91214020fe]
/usr/lib/llvm-15/bin/../lib/libLLVM-15.so.1(+0xf048d6)[0x7f91214048d6]
/lib/x86_64-linux-gnu/libc.so.6(+0x42520)[0x7f911ffd2520]
[0x7f911ddd0041]
/usr/lib/llvm-15/bin/../lib/libLLVM-15.so.1(_ZN4llvm3orc9runAsMainEPFiiPPcENS_
8ArrayRefINSt7
__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEEEENS
_8OptionalINS_9StringRefEEE+0x4b5)[0x7f9122f7ed55]
lli-15(_Z9runOrcJITPKc+0x2fb7)[0x7f9127567257]
lli-15(main+0x329)[0x7f9127561c59]
/lib/x86_64-linux-gnu/libc.so.6(+0x29d90)[0x7f911ffb9d90]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0x80)[0x7f911ffb9e40]
lli-15(_start+0x25)[0x7f912755ec45]
浮動小数点例外（コアダンプ）
```

考察

n=2147483647 以外の実行結果は，正常に正整数 n の正の約数を求められていることがコード 6，7，8，9，10 からわかる．しかし，コード 11 の n=2147483647 の場合のみ出力値がマイナスとなった後に例外が発生し，プログラムがクラッシュしている．この原因として考えられるのは，n が 32 ビット符号付き整数型 (int) で表現可能な最大の正整数であることである．この正整数 n の正の約数を求めるため，i の値は n の値まで+1 され続けることになる．その結果，最後のループで int 型で定義可能な最大の数に到達し，最後のループ判定で i の値にさらに 1 が追加され，オーバーフローが発生する．オーバーフロー前の整数 i は 0111 1111 1111 1111 1111 1111 1111 1111 で表現されていたが，オーバーフローにより 1000 0000 0000 0000 0000 0000 0000 0000 となり，これは-2147483648 を表す数値である．これにより，-2147483648 からもう 1 度 n の約数を求める処理が繰り返され，-2147483647 と-1 が出力された．最後に，i が-1 の次に 0 となり，0 による除算を実行したためゼロ除算エラーが発生し，プログラムがクラッシュしたと考察する．以上の理由から，課題 1 の正整数 n の正の約数 i を求めるプログラムは，$1 \leq n \leq 2147483646$ の範囲内において妥当である．

4　課題 2

問題

バイナリ法によって冪剰余を求めるコード example/bpmod.pc を以下の手法でコンパイル，実行したときに要する時間を比較検討する．

picoc -O0　：picoc.c を用いて pico 言語を LLVM-IR に変換し，機械語に翻訳する手法．ここで，-O0 は最適化無効のオプションである．

picoc -O2　：picoc.c を用いて pico 言語を LLVM-IR に変換し，最適化を有効にして機械語に翻訳する手法．

picoc_x64_mem　：配布ファイル picoc_x64_mem.c を用いて pico 言語を機械語に直接変換する手法．式の途中結果をメモリに保存する．

picoc_x64_reg　：配布ファイル picoc_x64_mem.c を用いて pico 言語を機械語に直接変換する手法．式の途中結果をレジスタに保存する．

gcc -O0　：bpmod.pc の C 言語版 bpmod.c を gcc でコンパイルする手法．pico 言語の print() を printf() に変換するだけで実装可能である．ここ

で，-O0 は最適化無効のオプションである．

gcc -O2　：bpmod.c を gcc の最適化オプションを有効にしてコンパイルする手法．

今回用いるプログラム bpmod.pc をコード 12 に示す．

Source Code 12: bpmod.pc

```
1  int main()
2  {
3      int x; int n; int m;
4      x = 2; n = 1234567890; m = 1234;
5
6      int a;
7      a = 1;
8      while (0 < n) {
9          int q; int r; int c; int t;
10         // --- q = n / 2, r = n % 2 の計算
11         q = 0; r = n;
12         while (r < 2 < 1) { // r >= 2
13             q = q + 1;
14             r = r - 2;
15         }
16         c = r;
17         while (c) { // if (n % 2) のシミュレート
18             // --- a = (a * x) % m の計算
19             r = 0; t = x;
20             while (t) { r = r + a; t = t - 1; }
21             // r == a * x
22             while (r < m < 1) r = r - m;
23             // r == a * x % m
24             a = r;
25             c = 0;
26         }
27         // --- x = (x * x) % m の計算
28         r = 0; t = x;
29         while (t) { r = r + x; t = t - 1; }
30         // r == x * x
31         while (r < m < 1) r = r - m;
32         // r == x * x % m
33         x = r;
34
35         // --- n = n / 2 の計算
```

```
36          n = q;
37      }
38      print a; // 966
39  }
```

実験手順

本実験ではプログラムの実行時間計測に bash のビルトイン版 time コマンドを用いる．また，計測時間には誤差が発生するため，全ての計測を 3 回ずつ行い，その平均値を最終結果とする．本実験に用いる自作コンパイラを実行するため，以下の手順でコマンド操作を行う．

1. コンパイラ用ファイルのビルド:

```
:~/jikkenii$ make picoc
```

2. 指定された形式で bpmod.pc をコンパイル:

```
:~/jikkenii$ ./picoc < example/bpmod.pc > bpmod.ll
```

3. LLVM ファイルを機械語にコンパイル:

```
:~/jikkenii$ clang-15 -Wno-override-module -O2 bpmod.ll
```

4. 時間を計測しながらプログラムを実行:

```
:~/jikkenii$ time ./a.out
```

また，gcc によるコンパイルは以下のように行う．

1. gcc によって bpmod.c をコンパイル:

```
:~/jikkenii$ gcc -O0 example/bpmod.c
```

2. 時間を計測しながらプログラムを実行:

```
:~/jikkenii$ time ./a.out
```

回答

それぞれの手法で 1 回実行したときのコマンドラインをコード 13，14，15．16．17，18 に示す．

Source Code 13:
picoc -O0 の実行結果

```
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ make
gcc -Wall -std=c11 -O0
↪ -g3 -c -o
↪ picoc.o picoc.c
gcc -Wall -std=c11 -O0
↪ -g3 scan.o
↪ hashmap.o
↪ tidwall_hashmap.o
↪ picoc.o -o picoc
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ ./picoc
↪ < example/bpmod.pc
↪ > bpmod.ll
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$
↪ clang-15 -Wno-overr
↪ ide-module -O0
↪ bpmod.ll
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ time
↪ ./a.out
966

real 0m2.497s
user 0m2.484s
sys 0m0.000s
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$
```

Source Code 14:
picoc -O2 の実行結果

```
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ make
gcc -Wall -std=c11 -O0
↪ -g3 -c -o
↪ picoc.o picoc.c
gcc -Wall -std=c11 -O0
↪ -g3 scan.o
↪ hashmap.o
↪ tidwall_hashmap.o
↪ picoc.o -o picoc
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ ./picoc
↪ < example/bpmod.pc
↪ > bpmod.ll
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$
↪ clang-15 -Wno-overr
↪ ide-module -O2
↪ bpmod.ll
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ time
↪ ./a.out
966

real 0m0.027s
user 0m0.016s
sys 0m0.000s
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$
```

Source Code 15:
picoc_x64_mem の実行結果

```
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ make
↪ picoc_x64_mem
gcc -Wall -std=c11 -O0
↪ -g3 -c -o
↪ picoc_x64_mem.o
↪ picoc_x64_mem.c
gcc -Wall -std=c11 -O0
↪ -g3 scan.o
↪ hashmap.o
↪ tidwall_hashmap.o
↪ picoc_x64_mem.o -o
↪ picoc_x64_mem
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$
↪ ./picoc_x64_mem <
↪ example/bpmod.pc >
↪ bpmod.s
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ gcc
↪ bpmod.s
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ time
↪ ./a.out
966

real 0m2.092s
user 0m2.063s
sys 0m0.000s
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$
```

Source Code 16:
picoc_x64_reg の実行結果

```
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ make
↪ picoc_x64_reg
gcc -Wall -std=c11 -O0
↪ -g3    -c -o
↪ picoc_x64_reg.o
↪ picoc_x64_reg.c
gcc -Wall -std=c11 -O0
↪ -g3  scan.o
↪ hashmap.o
↪ tidwall_hashmap.o
↪ picoc_x64_reg.o -o
↪ picoc_x64_reg
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$
↪ ./picoc_x64_reg <
↪ example/bpmod.pc >
↪ bpmod.s
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ gcc
↪ bpmod.s
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ time
↪ ./a.out
966

real    0m1.518s
user    0m1.500s
sys 0m0.016s
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$
```

Source Code 17:
gcc -O0 の実行結果

```
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ gcc -O0
↪ example/bpmod.c
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ time
↪ ./a.out
966

real    0m2.522s
user    0m2.484s
sys 0m0.016s
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$
```

Source Code 18:
gcc -O2 の実行結果

```
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ gcc -O2
↪ example/bpmod.c
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$ time
↪ ./a.out
966

real    0m0.027s
user    0m0.000s
sys 0m0.031s
masaki@DESKTOP-H24FTFV:
↪ ~/jikkenii$
```

表 1 に，それぞれの手法を 3 回行ったときの実実行時間を示す．

表 1: 各コンパイル方法の実行時間（実時間）

手法	1 回目 [s]	2 回目 [s]	3 回目 [s]
picoc -O0	2.416	2.416	2.556
picoc -O2	**0.029**	**0.029**	**0.030**
picoc_x64_mem	2.241	2.194	2.226
picoc_x64_reg	1.623	1.690	1.639
gcc -O0	2.577	2.590	2.614
gcc -O2	**0.030**	**0.029**	**0.028**

図 4 に，それぞれの手法ごとの平均実行時間を示す．ここで，青色が実実行時間，赤色がユーザ実行時間．緑色がシステム実行時間であり，それぞれプログラムの呼び出しから終了までにかかった実時間，プログラム自体の処理時間，プログラムを処理するために、OS が処理をした時間である．

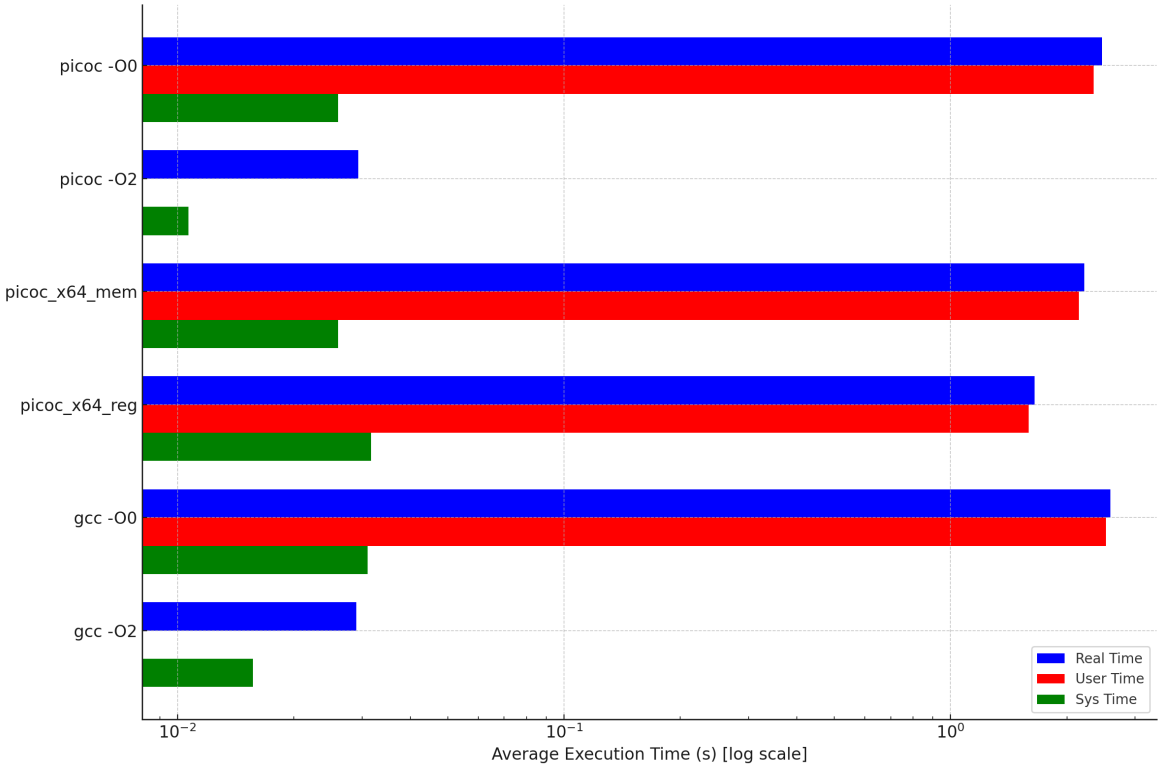

図 4: 手法ごとの実行時間

考察

実験結果から，picoc -O2，gcc -O2 の実行時間が他手法と比べて約 1/10 小さく，目に見えるほど優位な差が生じていることがわかる．これはコンパイラの最適化によるものであり，-O2 最適化を有効にすることでサポートされているほぼ全ての最適化を実行しているためコンパイル時間と生成コードのパフォーマンスの両方が著しく向上する．[2] また，picoc_x64_mem と picoc_x64_reg にも実行時間に約 0.5 秒の差が見られる．これは PC の構造上，数値をメモリに保存するより，レジスタに保存したほうがアクセス時間が短いことにより生じた差である．最後に picoc によるコンパイルを行った場合と gcc によるコンパイルを比較すると，最適化していない場合のみ picoc を用いた実行時間が約 0.1 秒速い．これは，picoc コンパイラが gcc と比べて小さい構成のコンパイラであるため，より今回のコードに最適化されていたためであると考察できる．そのため，最適化を行った場合は gcc もコードに最適化された構成になり，両方の差は見られない．

参考文献

[1] Wikipedia. Pico (programming language), 2023. Accessed: October 7, 2023.

[2] GNU. Optimize options, 2020. Accessed: October 15, 2023.