# XLANG Compiler, *version-0.0.1*

# Contents

# 1. Introduction

XLANG is an x86 high level language compiler that generate x86 assembly language. The syntax of the language is similar to those of C like languages.
It generates NASM 80x86 assembly language code. Only ELF32 format is supported for compiling, assembling and linking the code. But users are free to assembly and link code in any format after compiling.The compiler is not the fully functional that supports all features provided as per grammar rules at final x86 code generation.Compiler still lack of features such as abstract types such as records handling, better support for pointers, double precision floating points, more than one predicate handling in conditional as well as in loops. So these features can be used using inline assembly.

# 2. Building XLANG

Grab the source from the internet.
Following are the dependencies for building xlang.

```
GCC with C++11 Compiler(g++)
make
```

On Unix like systems, open terminal in directory and type following commands.

```
make
sudo make install
```

To remove xlang type following commands.

```
sudo make remove
```

To clean out object files and build directory, type

```
make cleanobj
make clean
```

# 3. Running XLANG

To compile a file, type following command:

```
xlang <options>  <filename>
```

It will compile, assemble with NASM and will link with GCC. The assemble file format is elf32 and link format with GCC is m32.

Options :

-S  :   Compile a file and generate assembly (.asm) file, but do not assemble it.

-c  :   Compile and assembly a file and generate object (.o) file but do not link it.

-O1  :  Apply optimizations to a code before code generation.

--print-tree  :  Print Abstract Syntax Tree(AST) generated during parsing.

--print-symtab  :  Print Global Symbol table.

--print-record-symtab :  Print record symbol table.

--no-cstdlib  :  Do not use any C standard library while linking.

--omit-frame-pointer  :  Omit stack frame pointer(push ebp, mov ebp, esp).

For more see:  **man xlang**

# 4. XLANG Language

Keywords :

| | | | | |
|---|---|---|---|---|
| asm | break | char | const | continue |
| do | double | else | extern | float |
| for | global | goto | if | int |
| long | record | return | short | sizeof |
| static | void | while | | |

Some examples are given in **examples** directory.
Here’s the HelloWorld program,

```
/* helloworld program in xlang */
extern void printf(char*);

global void main()
{
  printf(“Hello World!”);    // print hello world
}
```

Declaring function/variable as global prefix will make available to outside world.
Also to access other globals from other files, use extern.
For using functions.variables in one file declare them as static or nothing any global/extern.
Also variables must be declared before use in functions.

For more about syntax, see Appendix A.

Variable Declaration :
Supported data types are void, char, double, float, int, short, long and record types.

```
    e.g:  int x, y;
```

Selection statement :

```
 e.g:
  if (x < y){

  }else{
     return;
  }
```

Supported condition operators are <, >, <=, >=, ==, != and literal.

Loops :

Supports while loop, for loop and do-while loops with above condition operators.

e.g:

```
    while(x > 0){
      y++;
   }
```

Jump statement :

Supports goto, return and break;

```
if(x < 10){                    while(y >= 0){
    goto label;                    if(x != 10) {  break; }
 }else{                        }
   goto exit;
 }
label:
  y = 10;
exit:
```

Inline assembly :

See **examples/inline_assembly.x** file.

```
  asm{
      "<template>" [ <output-operand> : <input-operand> ]
   }
```

template :  any assembly instruction, output/input operands are replaced with each % or %n pattern in template.

Input/Output operand : it requires constraint as string in (“”). if has memory then its variable name.Output operand constraint follows ‘=’ sign before any type of constraint.

Constraint :

| | |
|---|---|
| a | eax |
| b | ebx |
| c | ecx |
| d | edx |
| S | esi |
| D | edi |
| m | memory |
| i | immediate value |
| % or %n | where n = 0, 1, ... |

e.g:

```
  asm{
     “mov eax, ebx”,
     “mov %, %” [ “=a”() : “m”(x) ]  // copy value of variable x into eax register
   }
```

Records :

An abstract types can be declared using records.

See **examples/record_test.x**

e.g:

```
record test{
  int x;
  char ch;
  float f;
  double d;
  test next;
  int array[3][3];
}
```

**Example** :
Here’s the factorial program from examples/fact.x

```
extern void scanf(char*, int);
extern void printf(char*, int);

void print_string(char* str)
{
  printf(str, 0);
}

int factorial(int num)
{
  int fact;
  fact = 1;

  if(num <= 0){
    return 1;
  }else{
    while(num > 0){
      fact = fact * num;
      num--;
    }
  }

  return fact;
}

global int main()
{
  int x,fact;

  print_string("Enter a number: ");
  scanf("%d", &x);
  fact = factorial(x);

  printf("factorial %d\n", fact);
}
```

Compile this program and generate assembly(.asm) file.

```
xlang -S examples/fact.x
```

Here’s the output of translated program into assembly language with comments, line numbers and function names with its arguments and types, local variables and their location on stack.

```
section .text
    extern scanf
    extern printf
    global main

; [ function: print_string(char str) ]
print_string:
    push ebp
    mov ebp, esp
    ; str = [ebp + 8], dword
; line: 6, func_call: printf
; line 6
    mov eax, 0
    push eax    ; param 2
; line 6
    mov eax, dword[ebp + 8]  ; assignment str
    push eax    ; param 1
    call printf
    add esp, 8    ; restore func-call params stack frame
._exit_print_string:
    mov esp, ebp
    pop ebp
    ret
; [ function: factorial(int num) ]
factorial:
    push ebp
    mov ebp, esp
    sub esp, 4    ; allocate space for local variables
    ; num = [ebp + 8], dword
    ; fact = [ebp - 4], dword
; line 12
    mov eax, 1
    mov dword[ebp - 4], eax
; condition checking, line 14
    cmp dword[ebp + 8], 0
    jle .if_label1
    jmp .else_label1
.if_label1:
; line 15
    mov eax, 1
    jmp ._exit_factorial    ; return, line 15
    jmp .exit_if1
.else_label1:
; while loop, line 17
```

```
.while_loop1:
; condition checking, line 17
    cmp dword[ebp + 8], 0
    jle .exit_while_loop1
; line 18
    xor eax, eax
    xor edx, edx
    mov eax, dword[ebp - 4]  ; fact
    mov ebx, dword[ebp + 8]  ; num
    mul ebx
    mov dword[ebp - 4], eax
; line 19
    dec dword[ebp + 8]    ; --
    jmp .while_loop1    ; jmp to while loop
.exit_while_loop1:
.exit_if1:
; line 23
    mov eax, dword[ebp - 4]  ; assignment fact
    jmp ._exit_factorial    ; return, line 23
._exit_factorial:
    mov esp, ebp
    pop ebp
    ret
; [ function: main() ]
main:
    push ebp
    mov ebp, esp
    sub esp, 8    ; allocate space for local variables
    ; fact = [ebp - 8], dword
    ; x = [ebp - 4], dword
; line: 30, func_call: print_string
    mov eax, string_val1
    push eax    ; param 1
    call print_string
    add esp, 4    ; restore func-call params stack frame
; line: 31, func_call: scanf
; line 31
    lea eax, [ebp - 4]    ; address of
    push eax    ; param 2
    mov eax, string_val2
    push eax    ; param 1
    call scanf
    add esp, 8    ; restore func-call params stack frame
; line: 32, func_call: factorial
; line 32
    mov eax, dword[ebp - 4]  ; assignment x
    push eax    ; param 1
    call factorial
    add esp, 4    ; restore func-call params stack frame
    mov dword[ebp - 8], eax    ; line: 32, assign
; line: 34, func_call: printf
; line 34
```

```
    mov eax, dword[ebp - 8]  ; assignment fact
    push eax    ; param 2
    mov eax, string_val3
    push eax    ; param 1
    call printf
    add esp, 8    ; restore func-call params stack frame
._exit_main:
    mov esp, ebp
    pop ebp
    ret

section .data
    string_val1 db
0x45,0x6E,0x74,0x65,0x72,0x20,0x61,0x20,0x6E,0x75,0x6D,0x62,0x65,0x72,0x3A,0x20,
0x00    ; 'Enter a number: '
    string_val2 db 0x25,0x64,0x00    ; '%d'
    string_val3 db
0x66,0x61,0x63,0x74,0x6F,0x72,0x69,0x61,0x6C,0x20,0x25,0x64,0x0A,0x00    ; 'factorial
%d\n'
```

# 4. Compiler Internals

Everything has implemented by hand.No tools has been used.
C++ programming language is used to write a compiler.
A lexer that returns a token(consist of numeric value, lexeme, location) using lexical grammar rules. A buffering technique is used for input of characters.(files: lexer.hpp and lexer.cpp, token.hpp)
For parsing, a hand written Recursive Descent Parser is used for parsing using grammar rules that generate an Abstract Syntax Tree(AST) along with the Symbol table.A Reverse Polish Notation technique is used to generate tree for primary expressions.(files: parser.hpp, parser.cpp, tree.hpp, tree.cpp, symtab.hpp, symtab.cpp)
A static sematic analyzer takes this AST from parser, travese it and checks for attribute errors(e.g. invalid types, undefined function-call etc.).(files: analyze.hpp, analyze.cpp)
An optimizer traverses AST and try to optimize it by reducing unused variables, precomputing constant expressions etc.(files: optimize.hpp, optimize.cpp)
An x86 code generation phase generate final NASM assembly code that is written to a file(.asm), that then will be assembled with NASM and will be linked with GCC to form final executable. Code generation uses the types of register supported by intel x86 as well as instructions with their separate types.(files: x86_gen.hpp, x86_gen.cpp, insn.hpp, insn.cpp, regs.hpp, regs.cpp)

# Appendix A – Grammar

keyword : one of
  asm
  break
  char
  const
  continue
  do
  double
  else
  extern
  float
  for
  global
  goto
  if
  int
  long
  record
  return
  short
  sizeof
  static
  void
  while

symbol : one of
  ! % ^ ~ & * ( ) - + = [ ] { } | : ; < > , . / \ ' " $

literal :
  integer-literal
  float-literal
  character-literal
  string-literal

integer-literal :
  decimal-literal
  octal-literal
  hexadecimal-literal
  binary-literal

decimal-literal :
  nonzero-digit
  nonzero-digit sub-decimal-literal

sub-decimal-literal :
  digit
  digit sub-decimal-literal

octal-literal :
  0
  0 sub-octal-literal

sub-octal-literal :
  octal-digit

```
  octal-digit sub-octal-literal

hexadecimal-literal :
  0x sub-hexadecimal-literal
  0X sub-hexadecimal-literal

sub-hexadecimal-literal :
  hexadecimal-digit
  hexadecimal-digit sub-hexadecimal-literal

binary-literal :
  0b sub-binary-literal
  0B sub-binary-literal

sub-binary-literal :
  one of 0 1
  one of 0 1 sub-binary-literal

digit : one of
    0 1 2 3 4 5 6 7 8 9

nonzero-digit : one of
           1 2 3 4 5 6 7 8 9

octal-digit : one of
  0 1 2 3 4 5 6 7

hexadecimal-digit : one of
           0 1 2 3 4 5 6 7 8 9
           a b c d e f
           A B C D E F


float-literal :
           digit-sequence . digit-sequence
           digit-sequence .

digit-sequence :
  digit
  digit digit-sequence

comment :
  / / any character except newline
  / * any character * /

character-literal :
  'c-char-sequence'

c-char-sequence :
  c-char
  c-char c-char-sequence

c-char :
  any character except single quote, backslash and new line
  escape-sequence

escape-sequence :
  \' \" \? \\ \a \b \f \n \r \t \v \0

string-literal :
  "s-char-sequence"

s-char-sequence :
  s-char
  s-char s-char-sequence

s-char :
  any character except double quote, backslash and new line
  escape-sequence

string-literal-sequence :
  string-literal
  string-literal , string-literal-sequence

non-digit : one of
  _ $ a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

identifier :
  non-digit
  non-digit sub-identifier

sub-identifier :
  non-digit
  digit
  non-digit sub-identifier
  digit sub-identifier

token :
  identifier
  keyword
  literal
  assignment-operator
  arithmetic-operator
  comparison-operator
  logical-operator
  bitwise-operator
  pointer-operator
  address-of-operator
  incr-operator
  decr-operator
  .
  ,
  {
  }
  [
  ]
  (
  )
  :
  ;


expression :
  primary-expression
  assignment-expression
  sizeof-expression
  cast-expression
  id-expression
  function-call-expression

primary-expression :
  literal
  identifier
  ( primary-expression )
  ( primary-expression ) primary-expression
  unary-operator primary-expression
  literal binary-operator primary-expression
  id-expression binary-operator primary-expression
  sub-primary-expression

sub-primary-expression :
  binary-operator primary-expression

constant-expression :
  integer-literal
  character-literal

assignment-expression :
  id-expression assignment-operator expression

incr-decr-expression :
  prefix-incr-expression
  postfix-incr-expression
  prefix-decr-expression
  postfix-decr-expression

prefix-incr-expression :
  incr-operator id-expression

postfix-incr-expression :
  id-expression incr-operator

prefix-decr-expression :
  decr-operator id-expression

postfix-decr-expression :
  id-expression decr-operator

incr-operator :
  ++

decr-operator :
  --

id-expression :
  identifier
  identifier . id-expression
  identifier -> id-expression
  identifier subscript-id-access
  pointer-indirection-access
  incr-decr-expression
  address-of-expression

subscript-id-access :
  [ identifier ]
  [ constant-expression ]
  [ id-expression ] subscript-id-access
  [ constant-expression ] subscript-id-access
  [ identifier ] . id-expression
  [ constant-expression ] -> id-expression

pointer-indirection-access :
  pointer-operator-sequence id-expression

pointer-operator-sequence :
  pointer-operator
  pointer-operator pointer-operator-sequence

pointer-operator :
  *

unary-operator : one of
  + - ! ~

binary-operator :
  arithmetic-operator
  logical-operator
  comparison-operator
  bitwise-operator

arithmetic-operator : one of
  + - * / %

logical-operator : one of
  && ||

comparison-operator : one of
  < <= > >= == !=

bitwise-operator : one of
  | & ^ << >> bit_and bit_or bit_xor

assignment-operator : one of
  = += -= *= /= %= |= &= ^= <<= >>=

address-of-expression :
  & id-expression

sizeof-expression :
  sizeof ( simple-type-specifier )
  sizeof ( identifier )

cast-expression :
  ( cast-type-specifier ) identifier

cast-type-specifier :
  simple-type-specifier
  identifier
  simple-type-specifier pointer-operator-sequence
  identifier pointer-operator-sequence

function-call-expression :
  id-expression ( )
  id-expression ( func-call-expression-list )

func-call-expression-list :
  expression
  expression , func-call-expression-list

type-specifier :
  simple-type-specifier
  record-name

simple-type-specifier :
  void
  char
  double
  float
  int
  short
  long

record-specifier :
  record-head { record-member-definition }

record-head :
  global record record-name
  record record-name

record-name :
  identifier

record-member-definition :
  type-specifier rec-id-list

rec-id-list :
  identifier
  identifier rec-subscript-member
  identifier , rec-id-list
  identifier rec-subscript-member , rec-id-list
  pointer-operator-sequence rec-id-list
  rec-func-pointer-member
  pointer-operator-sequence rec-func-pointer-member

rec-subscript-member :
  [ constant-expression ]
  [ constant-expression ] rec-subscript-member

rec-func-pointer-member :
  ( pointer-operator identifier ) ( rec-func-pointer-params )

rec-func-pointer-params :
  type-specifier
  type-specifier pointer-operator-sequence
  type-specifier , rec-func-pointer-params
  type-specifier pointer-operator-sequence , rec-func-pointer-params

declaration :
  simple-declaration
  function-declaration

simple-declaration :
  type-specifier simple-declarator-list
  const type-specifier simple-declarator-list
  extern type-specifier simple-declarator-list
  static type-specifier simple-declarator-list
  global type-specifier simple-declarator-list

simple-declarator-list :
  identifier
  identifier subscript-declarator
  identifier , simple-declarator-list
  identifier subscript-declarator , simple-declarator-list
  pointer-operator-sequence simple-declarator-list

subscript-declarator :
  [ constant-expression ]
  [ constant-expression ] subscript-declarator


function-declaration :
  func-head

func-head :
  type-specifier function-name ( func-params )
  extern type-specifier function-name ( func-params )
  global type-specifier function-name ( func-params )

function-name :
  identifier

func-params :
  type-specifier
  type-specifier identifier
  type-specifier pointer-operator-sequence
  type-specifier pointer-operator-sequence identifier
  type-specifier , func-params
  type-specifier identifier , func-params
  type-specifier pointer-operator-sequence , func-params
  type-specifier pointer-operator-sequence identifier , func-params

asm-statement :
  asm { asm-statement-sequence }

asm-statement-sequence :
  string-literal [ asm-operand : asm-operand ]
  string-literal [ asm-operand : asm-operand ] , asm-statement-sequence

asm-operand :
  string-literal ( expression )
  string-literal ( expression ) , asm-operand
```

```
function-definition :
  func-head { statement }

statement :
  labled-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  simple-declaration

statement-list :
  statement
  statement statement-list

labled-statement :
  identifier :

expression-statement :
  expression

selection-statement :
  if ( condition ) { statement-list }
  if ( condition ) { statement-list } else { statement-list }

condition :
  expression

iteration-statement :
  while ( condition ) { statement-list }
  do { statement-list } while ( condition ) ;
  for ( init-expression ; condition ; update-expression ) { statement-list }

jump-statement :
  break ;
  continue ;
  return expression
  goto identifier

declaration-statement :
  declaration
```

# Appendix B – Contact Information

Mail :- zopepritam444@gmail.com