2/20/2021

Todd Lunde

CSIS 492 – Senior Seminar

Dr. Chen

Project: 05 - Initial Code Base

Initial Code Base:*Von Neumann Computer Architecture*

I will be using some or most of the following programming code examples in my project. The code includes: High-level (C++) and low-level (ARM and MASM) program code examples, as well as compiler program code examples (Lexer, Token, Word and Num classes) in high-level (Java) programming language. Most of this code will be used in the languages portion of my report. My code examples will help to demonstrate the differences between high and low-level languages, as well as how high-level languages are compiled down into assembler and machine code instructions that computers (processors) can understand in order to execute them.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Github repository: (05 - Initial Code Base):

<https://github.com/twlunde/CSIS-492---Senior-Seminar---Project-05-Initial-Code-Base.git>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Iterative Fibonacci Sequence in C++ (high-level) Language

// Author: Todd Lunde

// File: (IterativeFibonacci.cpp - no multiplication - to run equivalent in VisUAL Arm Assembler)

// Class: CSIS 320 - Architecture

// Lab #4: part #4 (high-level language implementation), exercise #3

// Date: 4/4/2019

// Description:

// This file contains function int main(), function void fibonacci(int number)

#include <iostream>

#include <iomanip>

using namespace std;

void fibonacci(int number) // fibonacci function

{

int x = 0, y = 1, z = 0;

for(int i=0; i<=number; i++) // begin for loop

{

cout << x << " ";

z = x + y;

x = y;

y = z;

} // end for

} // end void fibonacci(int number)

int main() // begin program main()

{

int number;

cout << endl << endl;

cout << "Enter the number to output iterative fibonacci sequence for: ";

cin >> number;

cout << "\nThe iterative fibonacci sequence for the number " << number << " is: ";

fibonacci(number); // call function void fibonacci(int number)

cout << endl << endl;

return 0;

} // end main()

[Iterative Fibonacci Sequence in ARM (low-level) Assembly](https://stackoverflow.com/questions/23511377/recursive-fibonacci-in-masm-assembly) Language

; Author: Todd Lunde

; File: (IterativeFibonacci - no MUL for multiplication, using all addition)

; Class: CSIS 320 - Architecture

; Lab #4: part #4 (ARM implementation - VisUAL assembly), exercise #3

; Date: 4/4/2019

; ARM Assembly Code

MAIN

; int x = 0, y = 1, z = 0;

MOV R0, #0

MOV R1, #1

MOV R2, #0

BL FIBONACCI ; call function

MOV R4, R0 ; R4 = result

; void fibonacci(int number)

; {

; fibonacci

; void fibonacci(int number)

;{

; int x = 0, y = 1, z = 0;

FIBONACCI

MOV R1, #0 ; int x = 0

MOV R2, #1 ; y = 1

MOV R3, #0 ; z = 0

FOR

CMP R0, #40 ; R0 == i=1 to 10

BEQ DONE ; if (i == R0==10), exit loop

; cout << x << " ";

ADD R3, R1, R2 ; z = x + y;

MOV R1, R2 ; x = y;

MOV R2, R3 ; y = z;

ADD R0, R0, #1 ; i = i + 1, R0 = R0 + 1

B FOR ; repeat loop

MOV PC, LR ; return to caller

DONE

2/21/2021

Recursive Fibonacci Sequence in C++ (high-level) Language

// Author: Todd Lunde

// File: (RecursiveFibonacci.cpp)

// Class: CSIS 320 - Architecture

// Lab #4: part #4 (high-level language implementation - C++), exercise #4

// Date: 4/4/2019

// Description:

// This file contains function int main(), function int fibonacci(int number)

#include <iostream>

#include <iomanip>

using namespace std;

int fibonacci(int number)

{

if(number <= 1)

return number;

return fibonacci(number-1) + fibonacci(number-2);

} // end int fibonacci(int n)

int main() // begin program main()

{

int number;

cout << endl << endl;

cout << "Enter the number to output recursive fibonacci sequence for: ";

cin >> number;

cout << "\nThe iterative fibonacci sequence for the number " << number << " is: ";

for(int I = 0;I <= number; i++)

cout << fibonacci(i) << " ";

cout << endl << endl;

return 0;

} // end main()

[Recursive Fibonacci Sequence in](https://stackoverflow.com/questions/23511377/recursive-fibonacci-in-masm-assembly) [MASM (low-level) Assembly](https://stackoverflow.com/questions/23511377/recursive-fibonacci-in-masm-assembly) Language

include Irvine32.inc

.code

main PROC

mov ecx, 0

push 10 ; calculate the nth fib

call fib ; calculate fib (eax)

add esp, 4 ; clean up the stack

call WriteDec

call Crlf

exit

main ENDP

fib PROC C

add ecx,1

push ebp

mov ebp,esp

sub esp, 4 ; space for a local dword [ebp-4]

mov eax,[ebp+8] ; get n

; if ((n == 1) || (n == 2)) return 1;

cmp eax,2 ; n == 2?

je exception2

cmp eax,1 ; n == 1?

je exception2

; else return fib(n-1) + fib(n-2);

dec eax

push eax ; Fib(n-1)

call fib

mov [ebp-4], eax ; store first result

dec dword ptr [esp] ; (n-1) on the stack -> (n-2)

call fib

add esp, 4 ; clean up stack

add eax, [ebp-4] ; add result and stored first result

jmp Quit

exception2:

mov eax, 1 ; start values: 1, 1

; dec eax ; start values: 0, 1

Quit:

mov esp, ebp ; restore esp

pop ebp ; restore ebp

ret ; return EAX, stack not cleaned up

fib ENDP

END main

Compiler Segment - Lexical Analyzer “Lexer” Class in Java (high-level) Language

package lexer;

import java.io.\*;

import java.util.\*;

import symbols.\*;

public class Lexer {

   public static int line = 1;

   char peek = ' ';

   Hashtable words = new Hashtable();

   void reserve(Word w) { words.put(w.lexeme, w); }

   public Lexer() {

      reserve( new Word("if",    Tag.IF)    );

      reserve( new Word("else",  Tag.ELSE)  );

      reserve( new Word("while", Tag.WHILE) );

      reserve( new Word("do",    Tag.DO)    );

      reserve( new Word("break", Tag.BREAK) );

      reserve( Word.True );  reserve( Word.False );

      reserve( Type.Int  );  reserve( Type.Char  );

      reserve( Type.Bool );  reserve( Type.Float );

   }

   void readch() throws IOException { peek = (char)System.in.read(); }

   boolean readch(char c) throws IOException {

      readch();

      if( peek != c ) return false;

      peek = ' ';

      return true;

   }

   public Token scan() throws IOException {

      for( ; ; readch() ) {

         if( peek == ' ' || peek == '\t' ) continue;

         else if( peek == '\n' ) line = line + 1;

         else break;

      }

      switch( peek ) {

      case '&':

         if( readch('&') ) return Word.and;  else return new Token('&');

      case '|':

         if( readch('|') ) return Word.or;   else return new Token('|');

      case '=':

         if( readch('=') ) return Word.eq;   else return new Token('=');

      case '!':

         if( readch('=') ) return Word.ne;   else return new Token('!');

      case '<':

         if( readch('=') ) return Word.le;   else return new Token('<');

      case '>':

         if( readch('=') ) return Word.ge;   else return new Token('>');

      }

      if( Character.isDigit(peek) ) {

         int v = 0;

         do {

            v = 10\*v + Character.digit(peek, 10); readch();

         } while( Character.isDigit(peek) );

         if( peek != '.' ) return new Num(v);

         float x = v; float d = 10;

         for(;;) {

            readch();

            if( ! Character.isDigit(peek) ) break;

            x = x + Character.digit(peek, 10) / d; d = d\*10;

         }

         return new Real(x);

      }

      if( Character.isLetter(peek) ) {

         StringBuffer b = new StringBuffer();

         do {

            b.append(peek); readch();

         } while( Character.isLetterOrDigit(peek) );

         String s = b.toString();

         Word w = (Word)words.get(s);

         if( w != null ) return w;

         w = new Word(s, Tag.ID);

         words.put(s, w);

         return w;

      }

      Token tok = new Token(peek); peek = ' ';

      return tok;

   }

}

Compiler Segment - Lexical Analyzer “Token” Class in Java (high-level) Language

package lexer;

public class Token {

    public final int tag;

    public Token(int t) { tag = t; }

    public String toString() {return "" + (char)tag;}

}

Compiler Segment - Lexical Analyzer “Word” Class in Java (high-level) Language

package lexer;

public class Word extends Token {

   public String lexeme = "";

   public Word(String s, int tag) { super(tag); lexeme = s; }

   public String toString() { return lexeme; }

   public static final Word

      and = new Word( "&&", Tag.AND ),  or = new Word( "||", Tag.OR ),

      eq  = new Word( "==", Tag.EQ  ),  ne = new Word( "!=", Tag.NE ),

      le  = new Word( "<=", Tag.LE  ),  ge = new Word( ">=", Tag.GE ),

      minus  = new Word( "minus", Tag.MINUS ),

      True   = new Word( "true",  Tag.TRUE  ),

      False  = new Word( "false", Tag.FALSE ),

      temp   = new Word( "t",     Tag.TEMP  );

}

Compiler Segment - Lexical Analyzer (helper) “Num” Class Extending “Token” Class in Java (high-level) Language

package lexer;

public class Num extends Token {

    public final int value;

    public Num(int v) { super(Tag.NUM); value = v; }

    public String toString() { return "" + value; }

}

2/22/2021

Machine Code

[Machine Code. Under the hood of your program | by Rukshani Athapathu | Coder's Corner | Medium](https://medium.com/coderscorner/machine-code-dd8fcbe3153)

<https://medium.com/coderscorner/machine-code-dd8fcbe3153>

“Hello World!” written in C++ (high-level) Language with ASCII characters

#include <stdio.h>

int main()

{

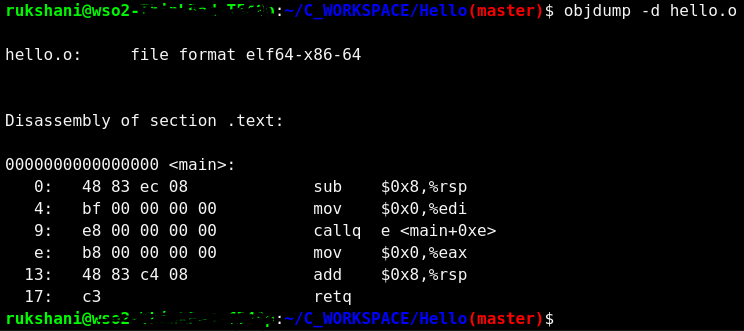
printf("Hello World!\n");

return 0;

}

*“Hello World!”* generated machine code (hello.o) in Linux results in a program called ‘OBJDUMP’, which we can use to view the generated instructions in assembly format

Image for post

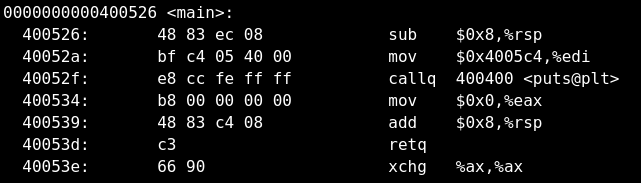


Relocatable object file

Executable code generated using a linker to run on this generated object file (hello.o): *gcc -o output hello.o*

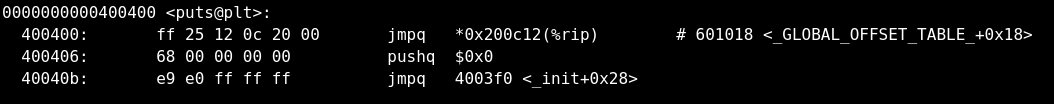
Disassembled generated executable file (output) using *objdump*

Image for post



Executable file content

Image for post



Processor executing executable file (output) loaded into main memory

Image for post

Image for post



Generated files

Machine Instructions

Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium)

Last Updated : 01 Apr, 2019

Machine Instructions are commands or programs written in machine code of a machine (computer) that it can recognize and execute.

* A machine instruction consists of several bytes in memory that tells the processor to perform one machine operation.
* The processor looks at machine instructions in main memory one after another, and performs one machine operation for each machine instruction.
* The collection of machine instructions in main memory is called a **machine language program.**

Machine code or machine language is a set of instructions executed directly by a computer’s central processing unit (CPU). Each instruction performs a very specific task, such as a load, a jump, or an ALU operation on a unit of data in a CPU register or memory. Every program directly executed by a CPU is made up of a series of such instructions.

The general format of a**machine instruction** is

|  |
| --- |
| **[Label:]                 Mnemonic      [Operand, Operand]                  [; Comments]** |

* Brackets indicate that a field is optional
* Label is an identifier that is assigned the address of the first byte of the instruction in which it appears. It must be followed by **“:”**
* Inclusion of spaces is arbitrary, except that at least one space must be inserted; no space would lead to an ambiguity.
* Comment field begins with a semicolon **“ ; ”**

**Example:**

|  |
| --- |
| **Here:                 MOV      R5,#25H                 ;load 25H into R5** |

**Machine instructions used in 8086 microprocessor**

**1. Data transfer instructions**– move, load exchange, input, output.

* MOV :Move byte or word to register or memory .
* IN, OUT: Input byte or word from port, output word to port.
* LEA: Load effective address
* LDS, LES Load pointer using data segment, extra segment .
* PUSH, POP: Push word onto stack, pop word off stack.
* XCHG: Exchange byte or word.
* XLAT: Translate byte using look-up table.

**2. Arithmetic instructions** – add, subtract, increment, decrement, convert byte/word and compare.

* ADD, SUB: Add, subtract byte or word
* ADC, SBB :Add, subtract byte or word and carry (borrow).
* INC, DEC: Increment, decrement byte or word.
* NEG: Negate byte or word   (two’s complement).
* CMP: Compare byte or word (subtract without storing).
* MUL, DIV: Multiply, divide byte or word (unsigned).
* IMUL, IDIV: Integer multiply, divide byte or word (signed)
* CBW, CWD: Convert byte to word, word to double word
* AAA, AAS, AAM,AAD: ASCII adjust for add, sub,  mul, div .
* DAA, DAS: Decimal adjust for addition, subtraction (BCD numbers)

**3. Logic instructions** – AND, OR, exclusive OR, shift/rotate and test

* NOT :   Logical NOT of byte or word (one’s complement)
* AND:  Logical AND of byte or word
* OR: Logical OR of byte or word.
* XOR: Logical exclusive-OR of byte or word
* TEST: Test byte or word (AND without storing).
* SHL, SHR: Logical Shift rotate instruction shift left, right byte or word? by 1or CL
* SAL, SAR: Arithmetic shift left, right byte or word? by 1 or CL
* ROL, ROR: Rotate left, right byte or word? by 1 or CL .
* RCL,  RCR: Rotate left, right through carry byte or word? by 1 or CL.

1. **String manipulation instruction** – load, store, move, compare and scan for byte/word

* MOVS: Move byte or word string
* MOVSB, MOVSW: Move byte, word string.
* CMPS:  Compare byte or word string.
* SCAS S: can byte or word string (comparing to A or AX)
* LODS, STOS:  Load, store byte or word string to AL.

**5. Control transfer instructions** – conditional, unconditional, call subroutine and return from subroutine.

* JMP:Unconditional jump .it includes loop transfer and subroutine and interrupt instructions.
* JNZ:jump till the counter value decreases to zero.It runs the loop till the value stored in CX becomes zero

**6. Loop control instructions-**

* LOOP: Loop unconditional, count in CX, short jump to target address.
* LOOPE (LOOPZ): Loop if equal (zero), count in CX, short jump to target address.
* LOOPNE (LOOPNZ): Loop if not equal (not zero), count in CX, short jump to target address.
* JCXZ: Jump if CX equals zero (used to skip code in loop).
* Subroutine and Intrrupt instructions-
* CALL, RET:  Call, return from procedure (inside or outside current segment).
* INT, INTO:  Software interrupt, interrupt if overflow.IRET: Return from interrupt.

**7. Processor control instructions-**

Flag manipulation:

* STC, CLC, CMC:  Set, clear, complement carry flag.
* STD, CLD:  Set, clear direction flag.STI, CLI: Set, clear interrupt enable flag.
* PUSHF, POPF: Push flags onto stack, pop flags off stack.

**Sample GATE Question**

Consider the sequence of machine instructions given below:

MUL R5, R0, R1

DIV R6, R2, R3

ADD R7, R5, R6

SUB R8, R7, R4

In the above sequence, R0 to R8 are general purpose registers. In the instructions shown, the first register stores the result of the operation performed on the second and the third registers. This sequence of instructions is to be executed in a pipelined instruction processor with the following 4 stages: (1) Instruction Fetch and Decode (IF), (2) Operand Fetch (OF), (3) Perform Operation (PO) and (4) Write back the Result (WB). The IF, OF and WB stages take 1 clock cycle each for any instruction. The PO stage takes 1 clock cycle for ADD or SUB instruction, 3 clock cycles for MUL instruction and 5 clock cycles for DIV instruction. The pipelined processor uses operand forwarding from the PO stage to the OF stage. The number of clock cycles taken for the execution of the above sequence of instructions is \_\_\_\_\_\_\_\_\_\_\_

**(A)** 11  
**(B)** 12  
**(C)** 13  
**(D)** 14

Answer: (C)

Explanation:

1 2 3 4 5 6 7 8 9 10 11 12 13

IF OF PO PO PO WB

IF OF PO PO PO PO PO WB

IF OF PO WB

IF OF PO WB